The Foundation - System Mission, Ubuntu, Terminal, and GitHub

Dr. TNOURJI, Abdellah
Research Scientist
The UM6P Vanguard Center

Autonomous Drone Engineering Internship

Mohammed VI Polytechnic University, UM6P , September 8, 2026

Outline

1 Mission and Foundations

2 Understanding Ubuntu and the Linux Environment

3 The Terminal and Essential Bash Foundations

4 Git and GitHub Foundations

1 Mission and Foundations

2 Understanding Ubuntu and the Linux Environment

3 The Terminal and Essential Bash Foundations

4 Git and GitHub Foundations

First, Observe the Destination

The Internship Mission

Build a reproducible autonomous-drone inspection system in simulation

Engineering Need

Inspection tasks may involve large, repetitive, difficult-to-access, or potentially hazardous areas. Aerial robots can help collect visual evidence efficiently and consistently.

The Internship Mission

Build a reproducible autonomous-drone inspection system in simulation

Engineering Need

Inspection tasks may involve large, repetitive, difficult-to-access, or potentially hazardous areas. Aerial robots can help collect visual evidence efficiently and consistently.

Internship Mission

Configure, program, simulate, test, and document a quadcopter that:

- follows a planned inspection mission;
- observes a simulated scene with a camera;
- detects selected objects;
- records useful mission evidence.

First, Observe the Destination

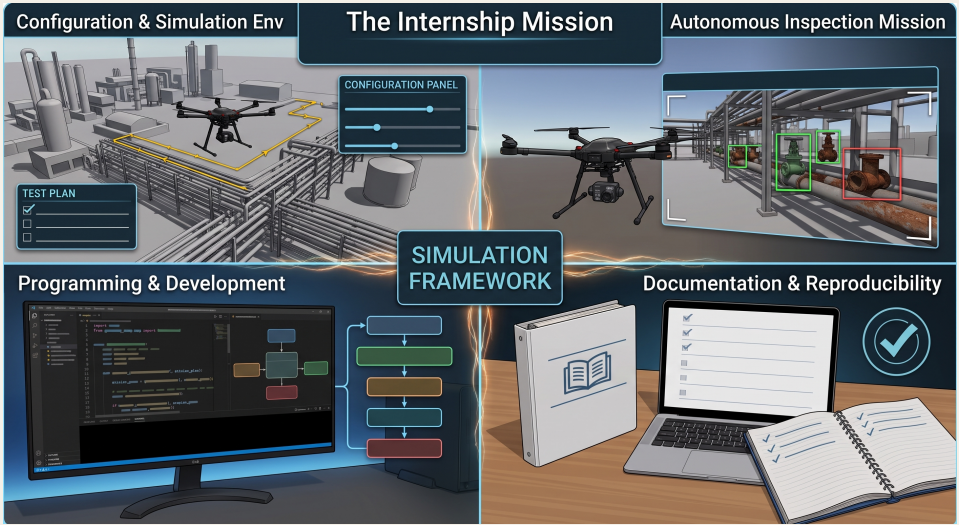
Observe the complete demonstration before studying its components.

Gazebo world $\longrightarrow$ PX4-controlled flight $\longrightarrow$ ROS 2 data exchange $\longrightarrow$ Camera perception $\longrightarrow$ Mission evidence

Important Boundary

A successful simulation is evidence of software integration, not proof that a real aircraft is ready or safe to fly.

First, Observe the Destination



The Engineering Challenge

One visible mission depends on several connected technical layers

Why Is the Project Technically Interesting?

A drone inspection mission is a **system-of-systems problem**. Motion, sensing, communication, software, perception, testing, and documentation must work together.

Operating environment → **Simulation** → **Flight control** → **Communication** → **Perception** → **Evidence**

The Engineering Challenge

One visible mission depends on several connected technical layers

Why Is the Project Technically Interesting?

A drone inspection mission is a **system-of-systems problem**. Motion, sensing, communication, software, perception, testing, and documentation must work together.

Operating environment → **Simulation** → **Flight control** → **Communication** → **Perception** → **Evidence**

A Chain Is Verified Link by Link

A detector cannot process a missing camera stream. A mission node cannot guide the vehicle through an incorrect interface. A working local setup cannot be reproduced without controlled dependencies and documentation.

One Project, Several Foundations

A complete robotic system is built layer by layer

The drone demonstration is the visible result.

Behind it, several technical foundations must work together.

One Project, Several Foundations

A complete robotic system is built layer by layer

The drone demonstration is the visible result.

Behind it, several technical foundations must work together.

Prepare

Configure the computer and create a reproducible project environment.

Build

Program, simulate, connect, and test the robotic components.

Deliver

Document the work, preserve the results, and share a working project.

One Project, Several Foundations

A complete robotic system is built layer by layer

The drone demonstration is the visible result.

Behind it, several technical foundations must work together.

Prepare

Configure the computer and create a reproducible project environment.

Build

Program, simulate, connect, and test the robotic components.

Deliver

Document the work, preserve the results, and share a working project.

Think of the Project as a Team

Ubuntu provides the workplace, Bash executes instructions, Git preserves progress, Python implements logic, ROS 2 connects components, Gazebo provides the simulated world, and AI interprets camera data.

You are not expected to master everything on Day 1.
You will learn each foundation when the project needs it.

Technology Ecosystem I

Preparing the environment and preserving your work

Ubuntu: The Workplace

Ubuntu is the Linux distribution used to run the project. It provides the files, processes, packages, and services required by the robotics tools.

Terminal and Bash: The Control Desk

The **terminal** is the window where commands are entered. **Bash** is the shell that reads and executes those commands, variables, and scripts.

Git: The Project Timeline

Git records changes on the computer. It allows students to create checkpoints, compare versions, and recover a previous working state.

GitHub: The Shared Project Home

GitHub hosts the Git repository online. It supports collaboration, documentation, issue tracking, review, and delivery of the internship project.

Technology Ecosystem I

Preparing the environment and preserving your work

Ubuntu: The Workplace

Ubuntu is the Linux distribution used to run the project. It provides the files, processes, packages, and services required by the robotics tools.

Terminal and Bash: The Control Desk

The **terminal** is the window where commands are entered. **Bash** is the shell that reads and executes those commands, variables, and scripts.

Git: The Project Timeline

Git records changes on the computer. It allows students to create checkpoints, compare versions, and recover a previous working state.

GitHub: The Shared Project Home

GitHub hosts the Git repository online. It supports collaboration, documentation, issue tracking, review, and delivery of the internship project.

How These Foundations Support the Drone Project

Ubuntu runs the tools → **Bash** launches the workflow → **Git** preserves the work → **GitHub** makes the project reproducible and shareable.

Technology Ecosystem II

From programming logic to simulated drone intelligence

Python: The Logic

Implements readable robotics programs, data processing, mission utilities, tests, and AI integration.

ROS 2: The Communication Network

Connects software components using nodes, topics, messages, services, and parameters. ROS 2 is not an operating system.

Gazebo: The Virtual Test Environment

Simulates the drone, camera, objects, environment, and physics before any real-world deployment.

PX4: The Flight Controller

Estimates the drone state, stabilizes the vehicle, and follows flight references inside the simulation.

AI and YOLOv8: The Perception

Processes camera images and detects objects of interest. Detection results provide observations, not direct flight control.

Important Engineering Boundary

A successful simulation helps us develop and test the system, but it does not guarantee that the drone is ready for safe real-world flight.

Internship Deliverables

What students are expected to build and submit

Working Engineering Assets

- structured Git repository and clear history;
- reproducible environment and configuration;
- ROS 2 packages, Python source code, and launch files;
- working Gazebo/PX4 inspection simulation;
- camera and object-detection integration.

Evidence and Communication

- layer-level and end-to-end tests;
- logs, screenshots, and mission results;
- installation and execution instructions;
- final demonstration;
- technical report with results, limitations, and future work.

Delivery Standard

A reviewer should be able to clone the repository, follow the documentation, reproduce the demonstration, and understand what was verified.

Learning Roadmap

Each foundation unlocks one part of the final GitHub project

Foundation	Contribution to the Project	Practical Outcome
Ubuntu	Provides the Linux engineering environment	Prepare the workstation
Terminal and Bash	Operate, configure, launch, inspect, and troubleshoot	Run the documented workflow
Git and GitHub	Preserve and share a reproducible project	Clone, manage, document, and deliver
Python for robotics	Express logic and process project data	Read and modify project programs
ROS 2	Connect software components through defined interfaces	Build and verify communication
Gazebo and PX4	Simulate the world and controlled drone motion	Execute the inspection mission
AI integration	Convert camera data into perception results	Produce inspection evidence

Goal of the Tutorials

End-to-End Objective

The tutorials are not isolated courses. Together, they provide the foundations required to reproduce, understand, test, and improve the autonomous-drone project hosted on GitHub.

The Student Journey

Progress from guided reproduction to engineering ownership

1. **Orient:** understand the mission, architecture, repository, and expected evidence.
2. **Prepare:** configure Ubuntu, tools, dependencies, workspace, and project files.
3. **Learn the foundations:** practise Bash, Git, Python, ROS 2, Gazebo, PX4, and perception concepts in context.
4. **Integrate:** connect mission logic, simulated flight, camera data, and object detection.
5. **Verify:** test one layer at a time before testing the complete pipeline.
6. **Deliver:** publish reproducible code, configuration, evidence, documentation, and a technical report.

Transition to the Technical Foundations

One foundation completed, the next one begins

Internship Learning Roadmap

Ubuntu **Current**	$\longrightarrow$	**Terminal and Bash**	$\longrightarrow$	Git and GitHub
Python for Robotics	$\longrightarrow$	ROS 2	$\longrightarrow$	Gazebo and PX4
		AI Integration		

1 Mission and Foundations

2 Understanding Ubuntu and the Linux Environment

3 The Terminal and Essential Bash Foundations

4 Git and GitHub Foundations

What Is an Operating System?

The coordinator between the computer, applications, and user

User and applications $\longleftrightarrow$ **Operating system** $\longleftrightarrow$ **Hardware**

What Is an Operating System?

The coordinator between the computer, applications, and user

User and applications $\longleftrightarrow$ Operating system $\longleftrightarrow$ Hardware

Intuitive Analogy

Think of the operating system as the manager of a laboratory. It assigns resources, organizes equipment, and controls access.

What It Manages

- files and storage;
- memory and running programs;
- users and permissions;
- devices, networks, and services.

Project Connection

When Gazebo uses the GPU, PX4 runs as a process, or ROS 2 reads a camera, the operating system coordinates the required resources.

Who Manages the Computer?

Programs need controlled access to physical resources

Active Programs (The Demand)

During the drone project, multiple heavy programs run simultaneously:

- **Gazebo:** simulates physics;
- **PX4:** controls flight logic;
- **ROS 2:** exchanges messages;
- **YOLOv8:** detects objects.

Who Manages the Computer?

Programs need controlled access to physical resources

Active Programs (The Demand)

During the drone project, multiple heavy programs run simultaneously:

- **Gazebo:** simulates physics;
- **PX4:** controls flight logic;
- **ROS 2:** exchanges messages;
- **YOLOv8:** detects objects.

Shared Resources (The Supply)

All these programs must compete for the same physical hardware:

- **Processor** (CPU time);
- **Memory** (RAM allocation);
- **Files** (Storage access);
- **Devices** (GPU and Network).

Who Manages the Computer?

Programs need controlled access to physical resources

Active Programs (The Demand)

During the drone project, multiple heavy programs run simultaneously:

- **Gazebo:** simulates physics;
- **PX4:** controls flight logic;
- **ROS 2:** exchanges messages;
- **YOLOv8:** detects objects.

Shared Resources (The Supply)

All these programs must compete for the same physical hardware:

- **Processor** (CPU time);
- **Memory** (RAM allocation);
- **Files** (Storage access);
- **Devices** (GPU and Network).

The Fundamental Problem & Key Intuition

These programs cannot safely share resources independently. If Gazebo and YOLOv8 both demand 100% of the GPU at the same time, the system crashes.

Solution: Between the software and the physical computer, we need a trusted, central traffic controller. This low-level manager is the **kernel**.

The Kernel

The low-level manager between software and hardware

Applications and Robotics Programs

Gazebo PX4 ROS 2 Python YOLOv8

$\Downarrow$

Linux Kernel

Controls and coordinates access to computer resources

$\Downarrow$

Processor, Memory, Storage, Network, and Devices

The Kernel

The low-level manager between software and hardware

Applications and Robotics Programs

Gazebo PX4 ROS 2 Python YOLOv8

⇓

Linux Kernel
Controls and coordinates access to computer resources

⇓

Processor, Memory, Storage, Network, and Devices

What Does the Kernel Manage?

- **Processes:** which programs are running and when they use the processor;
- **Memory:** which memory regions belong to each running program;
- **Files and storage:** how data is read, written, and protected;
- **Devices:** how software communicates with hardware through drivers;

Linux Is More Than a Kernel

From the technical core to a complete operating environment

1. The Linux Kernel

The low-level engine that talks directly to the hardware:

- processes;
- memory;
- files & networks;
- hardware devices.

2. User Tools & Ecosystem

Software that makes the kernel actually usable by humans:

- a shell (Bash);
- terminal commands;
- software libraries;
- a package manager (`apt`).

3. Linux Distribution

A complete, ready-to-use operating system assembled from both parts.

Ubuntu 22.04

Linux Is More Than a Kernel

From the technical core to a complete operating environment

1. The Linux Kernel

The low-level engine that talks directly to the hardware:

- processes;
- memory;
- files & networks;
- hardware devices.

2. User Tools & Ecosystem

Software that makes the kernel actually usable by humans:

- a shell (Bash);
- terminal commands;
- software libraries;
- a package manager (`apt`).

3. Linux Distribution

A complete, ready-to-use operating system assembled from both parts.

Ubuntu 22.04

The Operating System Equation

Kernel + Tools, Shell & Apps = Distribution (Ubuntu)

Linux Is More Than a Kernel

From the technical core to a complete operating environment

The Operating System Equation

Kernel + Tools, Shell & Apps = Distribution (Ubuntu)

Operating System Comparison

Why don't we just program the drone on Windows or macOS?

Ubuntu (Linux)

The Robotics Standard

- **Native:** ROS 2 and Gazebo are built primarily for this OS.
- **Open:** Complete, unfiltered access to hardware.
- **Focus:** Command-line automation.

Windows

The General Desktop

- **Familiar:** Broadest consumer support.
- **Focus:** Graphical User Interfaces (GUI).
- **Barrier:** Requires Virtual Machines or WSL to run Linux tools, adding overhead.

macOS

The Premium Workspace

- **Unix-based:** Excellent developer environment.
- **Closed:** Locked to Apple hardware.
- **Barrier:** Compiling complex robotics binaries natively is often unsupported.

Operating System Comparison

Why don't we just program the drone on Windows or macOS?

Ubuntu (Linux)

The Robotics Standard

- **Native:** ROS 2 and Gazebo are built primarily for this OS.
- **Open:** Complete, unfiltered access to hardware.
- **Focus:** Command-line automation.

Windows

The General Desktop

- **Familiar:** Broadest consumer support.
- **Focus:** Graphical User Interfaces (GUI).
- **Barrier:** Requires Virtual Machines or WSL to run Linux tools, adding overhead.

macOS

The Premium Workspace

- **Unix-based:** Excellent developer environment.
- **Closed:** Locked to Apple hardware.
- **Barrier:** Compiling complex robotics binaries natively is often unsupported.

The Engineering Decision

While Windows and macOS are excellent for daily tasks, forcing them to run advanced robotics software requires workarounds (like Virtual Machines) that drain CPU and RAM. **Ubuntu provides the native, zero-overhead environment required for the drone's digital twin.**

Why One Exact Ubuntu Version Matters

Compatibility is part of engineering, not an installation detail

Ubuntu version $\rightarrow$ available packages $\rightarrow$ compatible robotics stack

Why One Exact Ubuntu Version Matters

Compatibility is part of engineering, not an installation detail

Ubuntu version → available packages → compatible robotics stack

Example Compatibility Pairs

- ROS 2 Humble: Ubuntu 22.04
- ROS 2 Jazzy: Ubuntu 24.04

Internship Rule

Follow the version declared in the project documentation. Do not replace it only because another release is newer.

Why This Matters

A tutorial written for another Ubuntu or ROS 2 release may contain different package names, dependencies, or setup commands.

The Filesystem: One Organized Tree

Linux organizes files from a single starting point

$$\texttt{/} \rightarrow \texttt{/home} \rightarrow \texttt{/home/student} \rightarrow \texttt{project workspace}$$

The Filesystem: One Organized Tree

Linux organizes files from a single starting point

`/` → `/home` → `/home/student` → `project workspace`

Filesystem

The rule and structure used to organize files and directories. The top of the Linux tree is called **root** and is written as `/`.

Home Directory

Each user normally works inside a personal directory such as `/home/student`. The shortcut `~` means "my home directory."

Project Connection

File paths tell scripts where to find worlds, models, source code, configuration files, logs, and datasets.

Users, Administrator, and Root

Permissions protect the system from accidental changes

User

A normal account for daily work, coding, simulation, and documents.

Administrator

A trusted user allowed to perform selected system-level tasks.

Root

The system's all-powerful administrative account. Root can modify almost anything.

Users, Administrator, and Root

Permissions protect the system from accidental changes

User

A normal account for daily work, coding, simulation, and documents.

Administrator

A trusted user allowed to perform selected system-level tasks.

Root

The system's all-powerful administrative account. Root can modify almost anything.

What sudo Means

sudo requests permission to run one command with administrative privileges. It is not a decoration to add when a command fails.

Safe Working Habit

Work as a normal user. Use administrative privileges only when the command is understood and the trusted documentation requires them.

Packages, Dependencies, and Repositories

How Ubuntu obtains and maintains software

Package

A prepared unit of software with metadata needed for installation and maintenance.

Dependency

Another package or library required for the software to work.

Repository

A configured software source from which packages and updates can be obtained.

Terminal, Shell, and Bash

The window, the interpreter, and the specific tool

Terminal

The application window that displays text and accepts keyboard input.

Shell

The command interpreter that reads instructions and starts programs.

Bash

A widely used shell that supports commands, variables, scripts, redirection, and automation.

Terminal, Shell, and Bash

The window, the interpreter, and the specific tool

Terminal

The application window that displays text and accepts keyboard input.

Shell

The command interpreter that reads instructions and starts programs.

Bash

A widely used shell that supports commands, variables, scripts, redirection, and automation.

Restaurant Analogy

The terminal is the counter, the shell is the person interpreting the order, and Bash is one specific language and rule set used to express that order.

Next tutorial: we will use the terminal and Bash safely, one command at a time.

Open Source Is More Than Free Access

It is a collaborative engineering model

Open-Source Software

Source code is available under a license that defines how it may be inspected, used, modified, and shared.

Modular Software

Complex systems are divided into components with clear responsibilities and interfaces.

Community Collaboration

Developers contribute code, reviews, documentation, tests, bug reports, and technical discussions.

Documentation

Installation steps, assumptions, versions, and known problems are recorded so that others can repeat the work.

Professional Habit

Reusing open-source software does not remove engineering responsibility. We still verify versions, licenses, interfaces, behavior, and assumptions.

Reproducibility: “It Works on My Computer” Is Not Enough

Another student should be able to obtain the same result

Same repository + same versions + documented steps → comparable result

Reproducibility: “It Works on My Computer” Is Not Enough

Another student should be able to obtain the same result

Same repository + same versions + documented steps → comparable result

Record

- Ubuntu and ROS 2 versions;
- required packages and dependencies;
- configuration and environment variables;
- launch and test commands.

Verify

- start from a clean terminal;
- follow the documented sequence;
- compare expected and observed outputs;
- report missing assumptions.

Error Messages Are Engineering Evidence

Troubleshooting begins with observation, not random commands

1. **Stop and read** the complete message.
2. **Locate** the first useful error, file, package, or process name.
3. **Check context**: directory, command, versions, and environment.
4. **Form one hypothesis** and test one change at a time.
5. **Document the verified fix** and how it was validated.

Error Messages Are Engineering Evidence

Troubleshooting begins with observation, not random commands

1. **Stop and read** the complete message.
2. **Locate** the first useful error, file, package, or process name.
3. **Check context**: directory, command, versions, and environment.
4. **Form one hypothesis** and test one change at a time.
5. **Document the verified fix** and how it was validated.

Beginner-Friendly Question

Instead of asking "Why does nothing work?", ask: "Which component failed, what evidence identifies it, and what changed since the last working state?"

Avoid

Blindly adding `sudo`, repeatedly reinstalling everything, or copying commands written for another software version.

What You Should Now Be Able to Explain

Conceptual checkpoint before using the command line

- An **operating system** coordinates applications, users, and hardware.
- **Ubuntu** is a Linux distribution; **Linux** provides the kernel foundation.
- Ubuntu is selected because it offers a supported and reproducible robotics environment.
- A **filesystem** organizes files; users and permissions control access.
- Packages come from repositories and may require dependencies.
- Running programs are processes that receive settings from their environment.
- Open-source robotics depends on collaboration, modularity, documentation, reproducibility, and disciplined troubleshooting.

What You Should Now Be Able to Explain

Conceptual checkpoint before using the command line

- An **operating system** coordinates applications, users, and hardware.
- **Ubuntu** is a Linux distribution; **Linux** provides the kernel foundation.
- Ubuntu is selected because it offers a supported and reproducible robotics environment.
- A **filesystem** organizes files; users and permissions control access.
- Packages come from repositories and may require dependencies.
- Running programs are processes that receive settings from their environment.
- Open-source robotics depends on collaboration, modularity, documentation, reproducibility, and disciplined troubleshooting.

Next: Terminal and Bash Foundations

Transition to the Technical Foundations

One foundation completed, the next one begins

Internship Learning Roadmap

Ubuntu Completed	**Terminal and Bash** Current	Git and GitHub
Python for Robotics	ROS 2	Gazebo and PX4
	AI Integration	

References and Further Reading I

Ubuntu and Linux foundations

Ubuntu, *About the Ubuntu Project*.
https://ubuntu.com/about

Canonical, *Official Ubuntu Documentation*.
https://help.ubuntu.com/

Canonical, *Ubuntu Desktop Documentation*.
https://documentation.ubuntu.com/desktop/

The Linux Kernel Organization, *Linux Kernel Documentation*.
https://docs.kernel.org/

Linux Foundation, *Filesystem Hierarchy Standard*.
https://refspecs.linuxfoundation.org/fhs.shtml

References and Further Reading II

Packages, permissions, and robotics compatibility

Ubuntu, *Ubuntu Packages Search*.
https://packages.ubuntu.com/

Debian Project, *APT User's Guide*.
https://www.debian.org/doc/manuals/apt-guide/

Ubuntu Community Help Wiki, *RootSudo*.
https://help.ubuntu.com/community/RootSudo

1 Mission and Foundations

2 Understanding Ubuntu and the Linux Environment

3 The Terminal and Essential Bash Foundations

4 Git and GitHub Foundations

From Ubuntu to the Terminal

The operating environment is ready; now we learn how to operate it

Completed Foundation

Ubuntu
The Linux environment in which the robotics tools run.

$\Longrightarrow$

Current Foundation

Terminal and Bash
The interface and language used to operate the project.

$\Longrightarrow$

Later Foundations

Git, Python, ROS 2, Gazebo, PX4, and AI

From Ubuntu to the Terminal

The operating environment is ready; now we learn how to operate it

Completed Foundation

Ubuntu
The Linux environment in which the robotics tools run.

$\Longrightarrow$

Current Foundation

Terminal and Bash
The interface and language used to operate the project.

$\Longrightarrow$

Later Foundations

Git, Python, ROS 2, Gazebo, PX4, and AI

Central Question

How can we communicate reliably with Ubuntu to organize, configure, launch, observe, and troubleshoot the autonomous-drone project?

Learning Principle

Do not memorize commands as isolated words. For every command, identify its purpose, inputs, effect on the system, expected output, and possible risks.

Graphical Interface or Command Line?

Two complementary ways to interact with the same computer

Graphical User Interface

A **GUI** uses windows, menus, buttons, and visual file browsers. It is intuitive for exploration and applications that require visual interaction.

Example: opening a folder by clicking its icon.

Command-Line Interface

A **CLI** accepts precise text commands. It is efficient for repetition, automation, remote access, configuration, and diagnostics.

Example: entering `cd /quard_um6p_intership`.

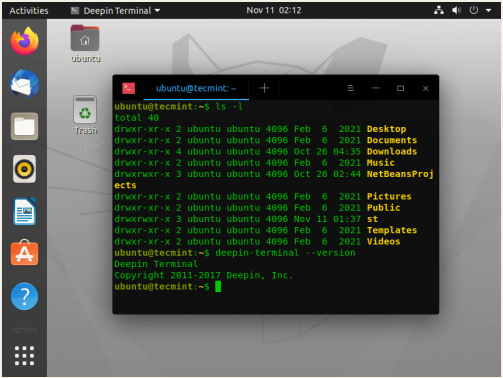
Graphical Interface or Command Line?

Two complementary ways to interact with the same computer

Command-Line Interface

A **CLI** accepts precise text commands. It is efficient for repetition, automation, remote access, configuration, and diagnostics.

Example: entering `cd /quard_um6p_intership`.



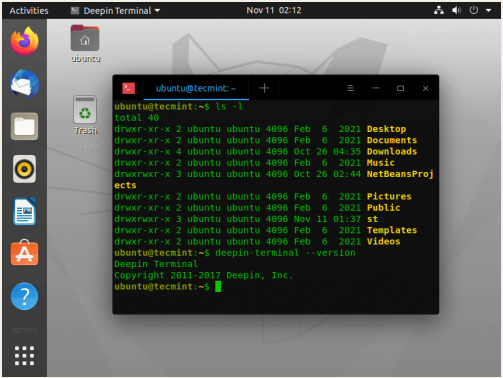
Graphical Interface or Command Line?

Two complementary ways to interact with the same computer

Command-Line Interface

A **CLI** accepts precise text commands. It is efficient for repetition, automation, remote access, configuration, and diagnostics.

Example: entering `cd /quard_um6p_intership`.



Engineering Perspective

The terminal does not replace every graphical tool. Engineers select the most appropriate interface: visual tools for observation and text commands for repeatable, inspectable, and automatable workflows.

Terminal vs. Shell vs. Bash

Untangling the command-line vocabulary

1. The Terminal

The Window (The Phone)

- The graphical black box on your screen.
- It has no brain. It only captures your keyboard typing and displays text back to you.

2. The Shell

The Job Category (Translator)

- A type of software that acts as a middleman.
- It reads the text from the terminal and translates it into instructions the Operating System understands.

3. Bash

The Specific Brand

- **B**ourne **A**gain **SH**ell.
- The default shell installed on Ubuntu.
- Other popular shells include **Zsh** (default on macOS) and **Fish**.

Terminal vs. Shell vs. Bash

Untangling the command-line vocabulary

1. The Terminal

The Window (The Phone)

- The graphical black box on your screen.
- It has no brain. It only captures your keyboard typing and displays text back to you.

2. The Shell

The Job Category (Translator)

- A type of software that acts as a middleman.
- It reads the text from the terminal and translates it into instructions the Operating System understands.

3. Bash

The Specific Brand

- **B**ourne **A**gain **SH**ell.
- The default shell installed on Ubuntu.
- Other popular shells include **Zsh** (default on macOS) and **Fish**.

The Information Flow

You type into the **Terminal** → The Terminal hands the text to **Bash** → **Bash** realizes cd means "Change Directory" and asks the **Kernel** to move you.

How to Read a Command

A command normally combines a program, options, and arguments

```
ls   -la    /quard_um6p_intership
```

Command

`ls`
The program or shell command to execute.

Options

`-la`
Modify behavior: long format and hidden files.

Argument

`/quard_um6p_intership`
The path on which the command operates.

How to Read a Command

A command normally combines a program, options, and arguments

```
ls   -la    /quard_um6p_intership
```

Command

`ls`
The program or shell command to execute.

Options

`-la`
Modify behavior: long format and hidden files.

Argument

`/quard_um6p_intership`
The path on which the command operates.

General Pattern

```
command [options] [arguments]
```

Important

Spaces separate parts of a command. Paths containing spaces normally require quotation marks or escaped spaces.

The Filesystem Is a Tree

Every file and folder has a location called a path

Important Locations

- `/`: root of the filesystem;
- `~`: current user's home directory;
- `.`: current directory;
- `..`: parent directory.

Project Example

`~/quard_um6p_intership`

The tilde expands to the home directory. The remaining name identifies the internship repository inside that home directory.

The Filesystem Is a Tree

Every file and folder has a location called a path

Important Locations

- /: root of the filesystem;
- : current user's home directory;
- .: current directory;
- ..: parent directory.

Project Example

`/quard_um6p_intership`

The tilde expands to the home directory. The remaining name identifies the internship repository inside that home directory.

`/home/student` ⟶ `quard_um6p_intership` ⟶ `gazebo` ⟶ `models`

Mental Model

A path is a route through the filesystem tree. Navigation commands change the current location; they do not move the project files themselves.

Absolute and Relative Paths

Two ways to describe the location of the same resource

Absolute Path

Starts at the filesystem root / and describes the complete route.

`/home/student/quard_um6p_intership/docs`

It does not depend on the current directory.

Relative Path

Starts from the current directory.

```
docs
./docs
../another_folder
```

Its meaning depends on where the shell currently is.

Common Mistake

If a command reports "No such file or directory," first run `pwd`, inspect the spelling and capitalization, and verify the path with `ls`. Linux paths are case-sensitive.

Locate Yourself and Inspect a Directory

Begin every workflow by confirming context

Purpose and Syntax

```
pwd                  # print the current directory
ls [options] [path] # list directory contents
```

Simple Examples

```
pwd
ls
ls -la
ls ~/Downloads
```

Project Examples

```
pwd
ls -la ~/quard_um6p_intership
ls gazebo/models
```

Common Mistakes

Do not assume that a new terminal opens in the project directory. Hidden files such as `.git` appear only when an option such as `-a` is used.

Navigate with cd

Change the shell's current working directory

Purpose and Syntax

```
cd [path]
```

Without an argument, cd returns to the home directory.

Navigation Examples

```
cd ~
cd ..
cd -
cd ~/Documents
```

Project Workflow

```
cd ~/quard_um6p_intership
pwd
ls
cd gazebo/models
```

Common Mistakes

cd requires a directory, not a regular file. Use tab completion to avoid spelling errors, and run `pwd` whenever the current location is uncertain.

Create Files and Directories

Build a safe practice workspace before modifying the repository

Purpose and Syntax

```
mkdir [options] directory   # create a directory
touch file                  # create an empty file or update its time
```

Simple Examples

```
mkdir bash_practice
touch notes.txt
mkdir -p logs/session_01
```

Project Exercise

```
cd ~/quard_um6p_intership
mkdir -p student_work/logs
touch student_work/observations.txt
```

Common Mistakes

Without `-p`, `mkdir` cannot create missing parent directories. Do not create random files inside source or configuration folders before understanding their purpose.

Copy and Move Safely

Preserve originals before reorganizing or editing important files

Purpose and Syntax

```
cp [options] source destination
mv source destination
```

cp duplicates data. mv relocates or renames it.

Simple Examples

```
cp notes.txt notes_backup.txt
cp -r config config_backup
mv notes.txt terminal_notes.txt
```

Project Examples

```
cp config/project.env \
   student_work/project.env.backup
mv student_work/observations.txt \
   student_work/day1_notes.txt
```

Safety Check

A destination file may be overwritten. Inspect both locations first, use clear backup names, and remember that copying a directory normally requires -r.

Remove Files with Care

Deletion from the terminal may bypass the graphical trash folder

Purpose and Syntax

```
rm [options] file
rm -r directory
```

Safe Practice Example

```
mkdir ~/bash_practice_remove
cd ~/bash_practice_remove
touch temporary.txt
ls
rm temporary.txt
ls
```

Remove Files with Care

High-Risk Command

`rm` does not provide a general undo operation. Before deletion:

1. run `pwd`;
2. inspect the target with `ls`;
3. verify the exact path;
4. avoid recursive or forced deletion unless it is truly necessary.

Never copy and execute an unfamiliar deletion command from the internet.

Read Text Files

Select the viewer according to the amount of information

Commands and Purposes

```
cat file        # print the complete file
less file       # inspect interactively
head file       # show the first lines
tail file       # show the last lines
tail -f file    # follow appended output
```

Project Examples

```
cat README.md
less PHASE_3_PIPELINE_README.md
head -n 20 config/project.env
tail -n 30 student_work/run.log
tail -f student_work/run.log
```

Practical Choice

Use `cat` for short files, `less` for long documents, `head` for introductions, and `tail` for recent logs. Press `q` to leave `less` or stop `tail -f` with `Ctrl+C`.

Common Mistake

Printing a very large log with `cat` may hide the useful information in thousands of terminal lines. Choose the smallest view that answers the question.

Produce Output and Keep a Record

Use echo and redirection to create simple evidence

Purpose and Syntax

```
echo "text"          # display text
command > file       # replace file content with output
command >> file      # append output to the file
```

Project Evidence Example

```
echo "Day 1 terminal verification" > student_work/run.log
pwd >> student_work/run.log
ls -la >> student_work/run.log
echo "Inspection complete" >> student_work/run.log
cat student_work/run.log
```

Find Files and Search Their Content

Locate the resource first, then inspect the information inside it

Purpose and Syntax

```
find [start_path] [tests]          # search filesystem entries
grep [options] pattern [file]      # search text content
```

Simple Examples

```
find . -name "*.py"
grep "camera" README.md
grep -n "error" student_work/run.log
```

Project Examples

```
find . -name "package.xml"
find gazebo/models -name "model.sdf"
grep -Rni "camera" docs
```

Common Mistakes

Quote wildcard patterns such as `"*.py"` so the shell does not expand them too early. Recursive searches can produce large outputs; begin from the smallest relevant directory.

Chain Steps with &&

Execute the next command only if the previous one succeeds

Syntax and Meaning

```
command_1 && command_2 && command_3
```

Bash runs the chain from left to right and stops when one command returns a failure status.

Project Example

```
cd ~/quard_um6p_intership && \
source config/project.env && \
pwd && \
ls gazebo/models
```

Why This Is Safer Than Unconditional Execution

If changing directory fails, the later commands are not executed in an unexpected location. However, && does not make a destructive command safe; every command in the chain must still be reviewed.

Environment Variables

Named values used to configure processes and tools

Intuition

An environment variable is a named configuration value inherited by programs started from the shell. It can describe paths, selected middleware, a world name, or another runtime setting.

Create and Read

```
export WORLD_NAME="default"
echo "$WORLD_NAME"
```

`export` makes the variable available to child processes.

Project Configuration

```
export GZ_SIM_RESOURCE_PATH="$HOME/.gz/models"
echo "$GZ_SIM_RESOURCE_PATH"
```

Common Mistakes

Do not place spaces around =. Quote values containing spaces. Use `$NAME` to read a variable, and verify the value before launching a process that depends on it.

Why We Use `source`

Apply configuration to the current shell session

Purpose and Syntax

`source` reads and executes a file **inside the current shell** rather than starting an isolated child shell.

Syntax:

```
source filename
. filename
```

Why This Matters

Setup files modify environment variables. If you forget to source a file, your terminal remains "blind"—ROS 2 packages, executables, or Gazebo resources will not be discoverable.

Security Note: Source only files you trust.

Typical Project Sequence

Every time you open a new terminal, you must "wake up" the robotics tools:

```
# 1. Load base ROS 2 tools
source /opt/ros/humble/setup.bash

# 2. Navigate to your workspace
cd ~/quard_um6p_intership

# 3. Load drone-specific variables
source config/project.env

# 4. Load your compiled code
source install/setup.bash
```

Comments and Readable Command Sequences

A command should explain its intention as well as its action

Comments with

In Bash, text following # is normally treated as a comment and is not executed. Comments record purpose, assumptions, and expected observations.

Project-Oriented Example

```
# Load ROS 2 commands into this terminal
source /opt/ros/humble/setup.bash

# Enter the internship repository
cd ~/quard_um6p_intership

# Verify the selected Gazebo world
export WORLD_NAME="default"
echo "$WORLD_NAME"
```

Permissions and chmod

Control who may read, modify, or execute a file

Read, Write, and Execute

- `r`: read the content;
- `w`: modify the content;
- `x`: execute the file as a program or script.

Inspect and Modify

```
ls -l launch_project.sh
chmod +x launch_project.sh
ls -l launch_project.sh
```

Execute Explicitly

```
./launch_project.sh
```

The `./` path tells Bash to use the script in the current directory.

Safety Concern

Executable permission does not make a script trustworthy. Read unfamiliar scripts before executing them, especially if they request elevated privileges.

Administrative Commands and sudo

Elevated privileges should be temporary, explicit, and justified

What sudo Means

`sudo command` runs one authorized command with elevated privileges. It is used when a normal user is not permitted to change a protected system resource.

Reasonable Use

```
sudo apt update
sudo apt install package_name
```

The password prompt may not display characters while typing. This is normal.

Safe Use Rules

- verify the package name and installation source;
- do not add sudo merely to silence a permission error;
- avoid running project scripts as root unless documentation gives a valid reason;
- never execute an unknown internet command with elevated privileges.

Install Software Safely with APT

Use Ubuntu's package manager and verify each installation

Two Different Operations

```
sudo apt update
sudo apt install package_name
```

`apt update` refreshes the package index. It does not upgrade every installed package. `apt install` installs the requested package and dependencies.

Safe Installation Workflow

```
# 1. Update the list of available software
sudo apt update

# 2. Find the exact package name
apt search package_name

# 3. Install the package
sudo apt install package_name

# 4. Verify it was installed correctly
which installed_command
installed_command --help
```

Project Rule: No Guessing

- **Trust the Docs:** Install only dependencies explicitly required by trusted project documentation.
- **Document Everything:** Record the exact package names and commands in your logbook so your teammates can reproduce the environment.
- **Stay Consistent:** Do not guess package names or mix unrelated installation methods (like `apt` vs. `pip` vs. `snap`) without understanding them.

Programs and Processes

A program is stored code; a process is a running instance

Program

A file containing instructions, such as a Python script, ROS 2 executable, or Gazebo binary.

Process

A running instance with a process identifier, memory, open files, environment variables, and operating state.

Inspect Running Processes

```
ps
ps aux
ps aux | grep "gz"
ps aux | grep "python"
```

Project Connection

Gazebo, PX4, ROS 2 nodes, bridges, and AI programs run as separate processes. A visible window is not the only evidence that a process exists, and an existing process is not automatically a healthy

Stop a Process Deliberately

Prefer graceful interruption before sending stronger signals

Normal Interactive Stop

Press `Ctrl+C` in the terminal running the foreground process. This sends an interrupt request and often allows the program to shut down cleanly.

Stop by Process Identifier

```
ps aux | grep "process_name"
kill PID
```

First identify the correct process. Replace `PID` with its numeric identifier.

Safety and Diagnosis

Do not kill processes by guesswork. Confirm the process owner and command first. A forced stop can lose data or hide the original fault. Use stronger termination only when graceful methods fail and the consequences are understood.

Discover Commands and Documentation

Ask the system before searching for random solutions

Locate a Command

```
which python3
which ros2
which gz
```

`which` shows the executable selected through the shell's search path.

Read Usage Information

```
ls --help
find --help
man ls
man chmod
```

Press `q` to leave a manual page.

Interpretation

`-help` usually provides concise command-specific usage. `man` provides structured reference documentation. `which` answers which executable would run, not whether that program is correctly configured.

Terminal Productivity Habits

Work faster without sacrificing accuracy

Completion and History

- **Tab:** complete a path or command;
- **Up/Down:** browse previous commands;
- **Ctrl+R:** search command history;
- `history`: display numbered history;
- `clear`: clear the visible terminal.

Editing and Control

- **Ctrl+C:** interrupt the current process;
- **Ctrl+L:** clear the screen;
- **Ctrl+A:** move to line start;
- **Ctrl+E:** move to line end;
- **Ctrl+U:** erase to line start.

Safety Habit

History saves time, but never re-run a recalled command automatically. Read the entire line and verify paths, variables, and options before pressing Enter.

Read Error Messages as Evidence

A Disciplined Diagnostic Sequence

1. identify the command that failed;
2. read the first meaningful error and its context;
3. classify the problem: path, permission, command, dependency, syntax, or process;
4. change one factor and execute again.

Typical Messages and First Checks

```
command not found            -> which command; check setup
No such file or directory    -> pwd; ls; inspect spelling
Permission denied            -> ls -l; inspect ownership and mode
package not found            -> apt update; verify package name
```

Do Not Hide the Problem

Adding sudo, reinstalling everything, or killing every process may change the symptom without identifying the cause. Preserve the original command and output before troubleshooting.

Common Command-Line Mistakes

Small syntax errors can change the meaning of an operation

Context and Paths

- running a command from the wrong directory;
- confusing absolute and relative paths;
- ignoring capitalization;
- forgetting quotation marks around spaces;
- assuming means the project root.

Syntax and System State

- inserting spaces around variable assignment;
- forgetting to source a setup file;
- confusing > with »;
- using `sudo` as a universal repair;
- copying commands without reading their outputs.

Reliable Recovery

Return to known facts: `pwd`, `ls -la`, `echo "$VARIABLE"`, `which command`, `ps aux`, and the complete error message.

Terminal Sandbox (Part 1/3)

Environment Initialization and Navigation

1. Access the Sandbox

Open your browser to the free online terminal:

webterm.app/en/free-play

Mission 1: The Workspace

Before writing code, an engineer must orient themselves and build a clean folder structure.

1. Print your current working directory.
2. List the existing files.
3. Create a new folder named `drone_ws`.
4. Move inside `drone_ws`.
5. Verify that the folder is currently empty.

Terminal Sandbox (Part 1/3)

Environment Initialization and Navigation

1. Access the Sandbox

Open your browser to the free online terminal:

webterm.app/en/free-play

Mission 1: The Workspace

Before writing code, an engineer must orient themselves and build a clean folder structure.

1. Print your current working directory.
2. List the existing files.
3. Create a new folder named `drone_ws`.
4. Move inside `drone_ws`.
5. Verify that the folder is currently empty.

Mission 1: Solution Workflow

Try to complete the mission first. Use this solution if you get stuck:

```
# 1. Print Working Directory
pwd
# 2. List current files
ls -la
# 3. Create the workspace directory
mkdir drone_ws
# 4. Change directory into it
cd drone_ws
# 5. List files (should be empty)
ls
```

Mission 2: Configuration Files

A drone needs configuration parameters. Let's create a file, write data into it, and back it up.

1. Use `echo` to create a file named `config.env` containing the text `ROS_DOMAIN_ID=42`.
2. Read the contents of `config.env` directly in the terminal to verify.
3. Create a new folder named `backup`.
4. Copy `config.env` into the `backup` folder.

Terminal Sandbox (Part 2/3)

Creating, Reading, and Copying Files

Mission 2: Configuration Files

A drone needs configuration parameters. Let's create a file, write data into it, and back it up.

1. Use `echo` to create a file named `config.env` containing the text `ROS_DOMAIN_ID=42`.
2. Read the contents of `config.env` directly in the terminal to verify.
3. Create a new folder named `backup`.
4. Copy `config.env` into the `backup` folder.

Mission 2: Solution Workflow

```
# 1. Write text into a new file
echo "ROS_DOMAIN_ID=42" > config.env
# 2. Read (concatenate) the file
cat config.env
# 3. Create a backup folder
mkdir backup
# 4. Copy the file into the folder
cp config.env backup/
# Verify the copy was successful
ls backup/
```

Terminal Sandbox (Part 3/3)

Moving, Searching, and Cleanup

Mission 3: System Administration

Engineers must regularly restructure projects and search through massive log files.

1. Rename your `backup` folder to `archive`.
2. Use the `grep` command to search for the word `ROS` inside your archived configuration file.
3. Finally, clean up your workspace by permanently deleting the entire `archive` folder.

Danger Zone: `rm -rf`

In Linux, there is no "Recycle Bin" for the terminal. The `rm` command deletes files instantly and permanently. Use it with extreme caution!

Terminal Sandbox (Part 3/3)

Moving, Searching, and Cleanup

Mission 3: System Administration

Engineers must regularly restructure projects and search through massive log files.

1. Rename your `backup` folder to `archive`.
2. Use the `grep` command to search for the word `ROS` inside your archived configuration file.
3. Finally, clean up your workspace by permanently deleting the entire `archive` folder.

Danger Zone: `rm -rf`

In Linux, there is no "Recycle Bin" for the terminal. The `rm` command deletes files instantly and permanently. Use it with extreme caution!

Mission 3: Solution Workflow

```
# 1. Rename (Move) backup to archive
mv backup archive
# Verify the name changed
ls
# 2. Search for "ROS" inside the file
grep "ROS" archive/config.env
# 3. Delete the archive folder
# (-r = recursive, -f = force)
rm -rf archive/
# Verify it is gone forever
ls
```

References and Further Reading

Essential documentation for Day 1 foundations

William E. Shotts Jr.
The Linux Command Line.
linuxcommand.org

Canonical Ltd.
Ubuntu Command Line for Beginners.
ubuntu.com/tutorials

Scott Chacon & Ben Straub.
Pro Git Book.
git-scm.com/book

Transition to the Technical Foundations

One foundation completed, the next one begins

Internship Learning Roadmap

Ubuntu Completed	**Terminal and Bash** Completed	Git and GitHub Current
Python for Robotics	ROS 2	Gazebo and PX4
	AI Integration	

1 Mission and Foundations

2 Understanding Ubuntu and the Linux Environment

3 The Terminal and Essential Bash Foundations

4 Git and GitHub Foundations

From Bash to Version Control

The terminal operates files; Git manages their engineering history

Ubuntu
Completed

$\Longrightarrow$

Terminal and Bash
Completed

$\Longrightarrow$

Git and GitHub
Current foundation

New Engineering Question

How can several students modify the same robotics project while preserving its history, identifying each contribution, and keeping the GitHub repository reproducible?

Central Principle

Git is not a replacement for careful engineering. It makes careful work visible, traceable, reviewable, and recoverable.

Why Robotics Needs Version Control

Robotics systems evolve through many connected files and contributors

Without Version Control

- files become `final`, `final2`, and `final_fixed`;
- working configurations are overwritten;
- changes cannot be attributed or reviewed;
- two contributors may replace each other's work.

With Version Control

- each meaningful change becomes a recorded checkpoint;
- experiments can be isolated on branches;
- collaborators synchronize controlled changes;
- earlier states remain available for investigation.

Project Connection

A change to a ROS 2 node may depend on a launch file, model, environment variable, or README instruction. Version control preserves these relationships as the system evolves.

What Is Version Control?

A structured history of a project's meaningful states

Working files $\longrightarrow$ Selected changes $\longrightarrow$ Recorded checkpoint $\longrightarrow$ Shared history

Intuitive Definition

Version control records how a collection of files changes over time. It helps engineers answer: *What changed? Who changed it? Why was it changed? When was it changed?*

Git and GitHub

A local version-control tool and an online collaboration platform

Git

- distributed version-control software;
- runs on the local computer;
- records commits, branches, and merges;
- can work locally without GitHub.

GitHub

- online hosting and collaboration platform;
- stores shared Git repositories;
- provides browsing, issues, reviews, and documentation;
- connects contributors through remote repositories.

Git and GitHub

Git: The Local Camera

Think of Git as a digital camera for your project.

- When you write a good piece of code, you take a "snapshot" (**commit**).
- If your next experiment breaks the drone, you can simply rewind to an older snapshot.
- **Limitation:** These snapshots live *only* on your physical laptop. If the laptop breaks, the code is gone.

GitHub: The Shared Gallery

Think of GitHub as an online photo album.

- You upload your local snapshots to the internet (**push**).
- Your 9 teammates can browse your code and download it to their own laptops (**pull**).
- **Advantage:** It acts as the ultimate backup. It is the "single source of truth" for the team.

Git and GitHub

Git: The Local Camera

Think of Git as a digital camera for your project.

- When you write a good piece of code, you take a "snapshot" (**commit**).
- If your next experiment breaks the drone, you can simply rewind to an older snapshot.
- **Limitation:** These snapshots live *only* on your physical laptop. If the laptop breaks, the code is gone.

GitHub: The Shared Gallery

Think of GitHub as an online photo album.

- You upload your local snapshots to the internet (**push**).
- Your 9 teammates can browse your code and download it to their own laptops (**pull**).
- **Advantage:** It acts as the ultimate backup. It is the "single source of truth" for the team.

The Daily Project Workflow

1. **Write Code** $\longrightarrow$ 2. **Snap Photo** (`git commit`) $\longrightarrow$ 3. **Upload** (`git push`)

Essential Git Vocabulary

A project-centered mental model before using commands

Repository	Project files together with the Git history and metadata.
Commit	A recorded snapshot of selected changes with author, time, and message.
Branch	A movable line of development used to isolate related work.
Remote	A named connection to another repository, commonly `origin` on GitHub.
Clone	A local copy of a repository, including its history and remote connection.
Pull	Retrieve remote changes and integrate them into the current local branch.
Push	Publish local commits to a remote branch.
Merge	Combine histories from different lines of development.
`.gitignore`	Rules describing untracked files that Git should normally ignore.

Before Cloning: Verify the Destination

Clone the repository once and into the intended parent directory

Pre-Clone Checks

```
cd ~
pwd
ls -la
which git
git --version
```

Question Before Execution

Will `git clone` create `quard_um6p_intership` inside the current directory, or does a folder with that name already exist?

Common Mistake

Do not clone the repository repeatedly inside itself or inside another accidental clone. If the target folder already exists, inspect it before taking action.

Access the Internship Repository on GitHub

Understand the platform before copying commands

Repository Address

`https://github.com/abdtnourji/quard_um6p_intership`

Inspect on GitHub

- repository name and owner;
- default branch;
- README and documentation;
- folders and recent commits;
- access permissions.

Obtain the URL

Select **Code**, choose the documented authentication method, and copy the repository URL exactly. HTTPS is the simplest starting point when no SSH key workflow has been configured.

Clone the Internship Repository

Create one local working copy with history and remote configuration

Purpose and Syntax

```
git clone <repository-url>
```

Internship Command

```
cd ~
git clone https://github.com/abdtnourji/quard_um6p_intership.git
cd quard_um6p_intership
```

What the Command Creates

The project directory, working files, commit history, branches known at clone time, and a remote normally named `origin`.

Verify That the Clone Is Correct

A successful-looking download still requires engineering checks

Verification Sequence

```
cd ~/quard_um6p_intership
pwd
ls -la
git status
git branch
git remote -v
git log --oneline -5
```

Expected Evidence

The path identifies the intended repository, `.git` exists, Git recognizes a branch, the working tree is initially clean, `origin` points to the expected GitHub URL, and recent commits are visible.

Explore Before Modifying

Understand the project structure and follow its documentation

Filesystem View

```
ls -la
find . -maxdepth 2 -type d | sort
find . -name "README*"
find . -name "package.xml"
```

Git-Aware View

```
git status
git branch
git log --oneline --graph -8
git remote -v
```

Professional Rule

The README is part of the engineering interface. Read setup instructions, folder responsibilities, prerequisites, and known limitations before executing the pipeline.

Use `git status` as the Dashboard

Ask Git what it sees before and after every important action

Purpose and Syntax

```
git status
git status --short
```

Interpret the Categories

- **untracked**: Git has not been asked to include the file;
- **modified**: tracked content differs from the last commit;
- **staged**: content is selected for the next commit;
- **clean**: no uncommitted change is detected.

Habit

Run `git status` before editing, before staging, before committing, before pulling, and before pushing. It provides context, not just success or failure.

Create a Small, Safe Change

Practise the workflow with student-owned documentation

Project Exercise

```
cd ~/quard_um6p_intership
mkdir -p student_work
echo "Git workflow observations" > student_work/git_notes.txt
git status
```

Engineering Intention

This creates a visible, low-risk change without modifying flight logic, simulation models, or shared configuration.

Repository Policy

If the project defines a specific student folder, naming convention, or contribution process, follow that documentation instead of inventing a parallel structure.

Inspect Changes with `git diff`

Review content before selecting it for a commit

Purpose and Syntax

```
git diff              # unstaged tracked changes
git diff --staged     # changes selected for commit
```

Project Workflow

```
echo "Repository inspected from the project root." \
  >> student_work/git_notes.txt
git diff
git status
```

Important Detail

An untracked file may not appear in ordinary `git diff`. Use `git status` to discover it, then stage it before reviewing with `git diff -staged`.

Stage Only the Intended Change

Build the next commit deliberately

Purpose and Syntax

```
git add <file-or-directory>
```

Preferred Beginner Workflow

```
git add student_work/git_notes.txt
git status
git diff --staged
```

Why Not Always Use `git add .`?

Staging everything may include logs, generated files, credentials, unrelated edits, or large assets. Add explicit paths until you can explain every staged change.

Create a Meaningful Commit

Record one coherent engineering change with a useful message

Purpose and Syntax

```
git commit -m "concise description of the change"
```

Useful Message

```
git commit -m \
  "docs: add student Git workflow notes"
```

It states the area and result.

Weak Messages

`update`, `changes`, `final`, and `fix stuff` do not explain engineering intent.

After Committing

Run `git status` and `git log -oneline -5`. The commit exists locally until it is pushed.

Read the Project History

Commits form an engineering narrative, not merely a backup list

Purpose and Useful Views

```
git log
git log --oneline
git log --oneline --graph --decorate --all
```

Questions the History Can Answer

- Which change introduced a launch instruction?
- When was a model or configuration updated?
- Which branch contains a student's contribution?
- What commit preceded a newly observed problem?

Branches Isolate Lines of Work

Experiment without immediately changing the shared main line

Inspect and Create

```
git branch
git switch -c docs/student-git-notes
```

`git branch` lists branches. `git switch -c` creates and enters a new branch.

Move Between Existing Branches

```
git switch main
git switch docs/student-git-notes
```

Before Switching

Check `git status`. Uncommitted changes may follow you, conflict with the destination, or prevent the switch. Branch names should describe the task, not the contributor's mood.

What Does Merge Mean?

Combine compatible histories after review and verification

$$
\begin{array}{l}
\texttt{main:}\ A \longrightarrow B \longrightarrow C \\
\searrow D \longrightarrow E \quad \texttt{feature branch} \\
\qquad\qquad\quad \Downarrow \\
\texttt{main:}\ A \longrightarrow B \longrightarrow C \longrightarrow M
\end{array}
$$

Intuition

A merge combines changes from different lines of development. If both lines modify incompatible parts of the same content, Git asks an engineer to resolve the conflict.

Responsibility

A conflict is not Git failing. It is Git refusing to guess which engineering intention is correct. Resolve it by understanding both changes, then test the combined result.

Inspect the Remote Connection

Know where pulls come from and where pushes go

Purpose and Syntax

```
git remote -v
```

Expected Interpretation

`origin` is the conventional name created by cloning. The output normally shows fetch and push URLs for the GitHub repository.

Wrong-Remote Prevention

Before your first push, confirm that the owner and repository name are correct and that you have authorized contribution access. Do not replace `origin` randomly to bypass a permission problem.

Synchronize Before Publishing

Bring in remote work before sending your own commits

Safe Beginner Sequence

```
git status
git branch
git pull origin main
```

Meaning

`git pull` retrieves remote changes and integrates them into the current local branch. Read the output to determine whether the branch was already current, fast-forwarded, or requires conflict resolution.

Before Pulling

Start from a clean, understood working state. If local edits are unfinished, commit them appropriately or follow the instructor's recovery procedure rather than forcing the pull.

Publish Local Commits with `git push`

Synchronize verified history to GitHub

Purpose and Syntax

```
git push <remote> <branch>
```

Examples

```
git push origin main
# For a new contribution branch:
git push -u origin docs/student-git-notes
```

Rejected Push

A non-fast-forward rejection usually means the remote contains commits missing locally. Do not force-push. Retrieve and understand upstream changes, resolve any conflict, test, and then push normally.

The Everyday Internship Workflow

Observe, synchronize, modify, review, record, and share

1. **Enter and inspect:** move to the repository, then run `git status` and `git branch`.
2. **Synchronize:** pull the approved branch before beginning new work.
3. **Modify:** make one focused code, configuration, test, or documentation change.
4. **Review:** inspect `git status` and `git diff`.
5. **Stage:** add only the files intended for the next checkpoint.
6. **Verify staged content:** inspect `git diff -staged`.
7. **Commit:** use a concise message that explains the result.
8. **Synchronize again:** pull if necessary, resolve carefully, test, and push.
9. **Confirm on GitHub:** verify the branch, commit, and documentation online.

Team Rule

The exact branch and review policy is defined by the internship repository. Follow the documented collaboration workflow rather than assuming every contributor pushes directly to `main`.

Protecting the Project: Forking

Creating your own workspace without impacting the main repository

The "Golden Copy" Rule

The supervisor's main GitHub repository is the **Golden Copy**. If 10 students push their experimental code directly into it at the same time, the project will instantly break.

The Solution: "Forking"

A **Fork** is a completely separate, personal copy of the repository that lives on *your* GitHub account.

- You can freely modify it.
- You can completely break it.
- It **never** affects the supervisor's original code.

Your Safe Setup Workflow

Instead of cloning the main project directly, you will:

1. **Fork (In Browser):** Go to the supervisor's GitHub page and click the "Fork" button to create your copy.
2. **Clone (In Terminal):** Run `git clone` using the URL of *your* new fork, not the supervisor's URL.
3. **Push (In Terminal):** When you type `git push`, your code goes safely to your personal cloud repository.

Use .gitignore Deliberately

Keep generated, local, and sensitive artifacts out of history

The Purpose

A .gitignore file contains patterns telling Git to ignore specific untracked files, preventing them from accidentally being committed and pushed to GitHub.

Important Boundaries

- **Too Late:** Ignoring a file does *not* remove it from history if it was already committed previously.
- **Security:** Never rely on .gitignore as your only protection for credentials. Never commit passwords.
- **Bloat:** Use it to prevent uploading massive ROS 2 logs or compiled build files that crash repositories.

Typical Robotics Patterns

```
# Python cache
__pycache__/
*.pyc
# Local ROS 2 workspaces
build/
install/
log/
# Local environments
.venv/
# Sensitive secrets
.env
*.pem
```

Never Commit Secrets

Repository history can outlive the working file

Do Not Commit

- passwords, access tokens, API keys, or private SSH keys;
- private certificates or machine-specific credentials;
- confidential datasets or personal information;
- environment files containing secret values.

If a Secret Is Exposed

Stop sharing it, notify the responsible instructor or repository maintainer, revoke or rotate the credential, and follow the approved history-cleaning procedure. Merely deleting the file in a later commit may leave the secret in earlier history.

Better Pattern

Commit a documented template such as `.env.example` with placeholder names, then keep real secret values outside version control.

Keep the Repository Engineering-Friendly

Version the source of truth, not every generated artifact

Good Candidates

- source code and launch files;
- human-maintained configuration;
- small test fixtures;
- README files and reports;
- scripts needed to reproduce outputs.

Review Before Adding

- build and cache directories;
- large videos, models, bags, or datasets;
- temporary logs and screenshots;
- machine-specific settings;
- files obtainable from documented sources.

Reason

Large binary files make cloning, reviewing, and history management harder. Use the repository's approved storage and release strategy for necessary large assets.

Recover: Staged the Wrong File

Correct the selection without destroying the working change

Situation

You staged a file that should not be part of the next commit.

```
git status
git restore --staged path/to/file
git status
```

Effect

The file is removed from the staging area, but its working-tree modification is preserved for later review.

Do Not Guess Recovery Commands

Always inspect `git status` before and after recovery. Commands involving `-hard`, forced pushes, or history rewriting are outside the beginner workflow unless supervised.

Recover: Discard an Unwanted Local Edit

Use only when the last committed version is truly the desired state

For One Tracked, Unstaged File

```
git diff -- path/to/file
git restore path/to/file
git status
```

Irreversible Working-Tree Loss

`git restore path/to/file` discards the selected uncommitted edit. Copy valuable experimental content elsewhere or commit it on an appropriate branch before restoring.

Safer Question

Do I want to remove the file from the next commit, or destroy the actual edit? These are different intentions and require different commands.

Recover: Push Rejected or Pull Conflicted

Preserve evidence and resolve the real synchronization problem

Push Rejected

```
git status
git branch
git remote -v
git pull origin main
```

Read the output, integrate approved remote work, test, and push again. Never use force as the automatic response.

Merge Conflict

1. run `git status`;
2. open each conflicted file;
3. understand both alternatives;
4. remove conflict markers after deciding;
5. test the integrated result;
6. stage and commit the resolution.

Escalation Rule

If the correct engineering intention is unclear, stop and ask the file owner or instructor. A syntactically resolved conflict can still create incorrect robot behavior.

Two Computers, One Shared Repository

At the Computer Where Work Was Completed

```
git status
git add <intended-files>
git commit -m "describe the completed change"
git pull origin main
git push origin main
```

At the Other Computer

```
cd ~/quard_um6p_intership
git status
git pull origin main
```

Precondition

Each computer must be on the intended repository and branch. Uncommitted local edits must be reviewed before pulling, or they may conflict with the incoming work.

End-to-End Practice

Obtain, modify, record, synchronize, and verify

Guided Workflow

```
cd ~
git clone https://github.com/abdtnourji/quard_um6p_intership.git
cd quard_um6p_intership

git status
git remote -v
git log --oneline -5

mkdir -p student_work
echo "Git practice completed" > student_work/git_practice.txt

git status
git add student_work/git_practice.txt
git diff --staged
git commit -m "docs: add Git practice evidence"
git pull origin main
git push origin main
```

References

Primary documentation for continued practice

Git Project, *Git Documentation*.
https://git-scm.com/doc

Git Project, *A Tutorial Introduction to Git*.
https://git-scm.com/docs/gittutorial

GitHub Docs, *Git Basics*.
https://docs.github.com/en/get-started/git-basics

GitHub Docs, *Cloning a Repository*.
https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository

GitHub Docs, *Pushing Commits to a Remote Repository*.
https://docs.github.com/en/get-started/using-git/pushing-commits-to-a-remote-repository

Transition to the Technical Foundations

One foundation completed, the next one begins

Internship Learning Roadmap

Ubuntu Completed	**Terminal and Bash** Completed	Git and GitHub Completed
Python for Robotics Current	ROS 2	Gazebo and PX4
	AI Integration	