

Git Tutorial: 30-Minute Essentials and Research Reference

2026-10-18

Contents

Part I: The 30-Minute Essentials	2
Part II: Research-Oriented Reference	6
Course home · 简体中文 · PDF	

Item	Details
Applicability	Git 2.x; Ubuntu terminal; GitHub workflow examples
Documentation review date	2026-10-18
License	CC BY-NC-SA 4.0 · Xiong Zeng and contributors

Part I: The 30-Minute Essentials

Time	Topic
0–5 minutes	The four areas and key commands
5–10 minutes	Installation and identity
10–22 minutes	One complete GitHub workflow
22–26 minutes	Diffs and history
26–30 minutes	Sync and safety

1. Git Basics: Four Areas and Essential Commands

Git is a distributed version control system (VCS). It records the history of changes to files so that you can compare versions, restore older code, experiment in independent branches, and integrate changes from collaborators.

1.1 What problems does Git solve?

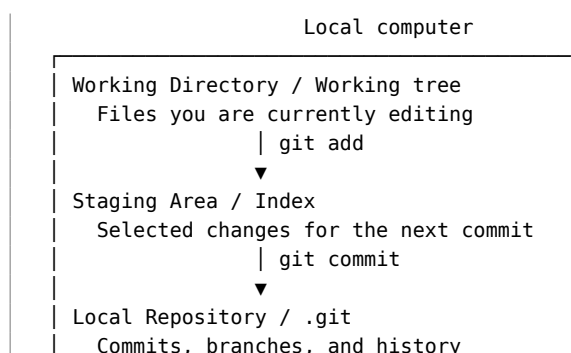
- **Version control:** Each commit identifies a traceable state of your code.
- **Safe experimentation:** Try a different reward function or training algorithm without disturbing a stable version.
- **Recovery:** Retrieve accidentally changed code or undo a buggy commit.
- **Collaboration:** Work independently on PPO, SAC, perception, and sim-to-real modules, then integrate them.
- **Cross-machine synchronization:** Use GitHub or GitLab to synchronize source code between a workstation and GPU servers.
- **Reproducibility:** Record the Commit ID used for a training run together with its configuration, seed, and dataset version.

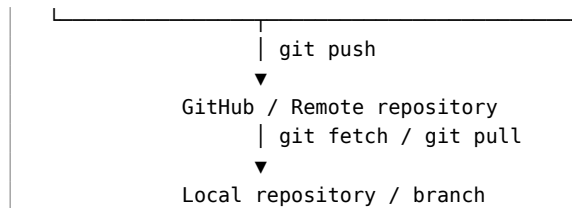
1.2 Git versus GitHub, GitLab, and VS Code

Tool	Purpose
Git	Local version control, commits, branches, and merging; most operations work offline.
GitHub	Online Git hosting with pull requests, code review, and automation.
GitLab	Another platform for hosting Git repositories and collaborating.
VS Code	A code editor with built-in Source Control support for Git.

Git is not GitHub. You can run `git init`, `git add`, `git commit`, and `git branch` without any GitHub account or network connection.

1.3 Working tree, staging area, local repository, remote repository





The following commands perform **three different operations**:

```
git add .
git commit -m "Implement PPO training"
git push
```

- **git add** places **selected file changes** in the staging area.
- **git commit** records the **staged content** in a local commit.
- **git push** sends **local commits** to a remote repository.

A commit is effectively a snapshot of the project at a particular moment, identified by a hash. Uncommitted working-tree edits do **not** get uploaded just because you run **git push**.

1.4 Ten essential commands

Command	Purpose
git init	Initialize a local repository
git clone URL	Clone a remote repository
git status	Inspect the current state
git add PATH	Stage file changes
git commit -m "..."	Create a local commit
git push	Publish commits
git pull	Fetch and integrate remote changes
git log --oneline	View compact commit history
git switch -c NAME	Create and switch to a branch
git merge NAME	Merge a specified branch

2. Install Git and Configure Your Identity

2.1 Install Git

Open an Ubuntu terminal and run:

```
sudo apt update
sudo apt install git
git --version
```

You should see a version such as **git version 2.x.y**; the exact version depends on your Ubuntu release and repositories.

2.2 Configure the commit author identity

```
git config --global user.name "Your Name"
git config --global user.email "your_email@example.com"
```

These values are written into the author identity of new commits. Use an email address associated with GitHub if you want GitHub to attribute the contributions correctly. Inspect the global configuration:

```
git config --global --list
```

You can also configure a different identity for **one repository** by omitting **--global**:

```
git config user.name "Lab Account"
git config user.email "lab@example.com"
```

2.3 Set the initial branch name

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

This affects repositories initialized with `git init` in the future. It does not rename existing branches.

3. The One End-to-End Workflow: Clone to Pull Request

This is the tutorial's **only fully sequenced day-to-day workflow**. It replaces overlapping local-repository, collaboration, research-workflow, and beginner-workflow examples in the previous edition.

3.1 Before you begin

Assume you have push access to an existing GitHub repository named `robot-policy`, and its default branch is `main`. If you do not have a remote repository yet, see Chapter 6 in Part II for `git init` and remote setup. HTTPS cloning can be anonymous for public repos; pushing requires GitHub authentication (Credential Manager, a suitable token, or SSH).

3.2 Clone, change, inspect, commit, and open a PR

```
# Clone once
git clone https://github.com/YOUR_USERNAME/robot-policy.git
cd robot-policy

# Start one isolated task from the latest main
git switch main
git pull --ff-only
git switch -c feature/pose-estimation

# Edit an existing source file, then inspect it
git status
git diff
git add train.py
git diff --staged
git commit -m "Improve visual pose estimation"

# First push of this feature branch
git push -u origin feature/pose-estimation
```

On GitHub, open a Pull Request from `feature/pose-estimation` into `main` for checks and review. After it is merged, run `git switch main` and `git pull --ff-only` locally. **Do not practice pushing to upstream open-source repositories where you lack write access; fork first when necessary.**

3.3 Local versus remote changes

`git add` modifies the index, `git commit` records local history, and only `git push` publishes local commits. A GitHub PR is a review/merge request, not a commit. Uncommitted working-tree changes are never sent by `git push`.

4. Inspect Changes with Status, Diff, and Log

When debugging a training run, knowing what the code **actually changed** is often more valuable than remembering what you *intended* to change.

4.1 Inspect the working tree

```
| git status
```

The output identifies your current branch, staged edits, unstaged edits, and untracked files.

4.2 Compare code

```
git diff          # Working tree vs staging area
git diff --staged # Staging area vs latest commit
git diff -- train.py # Changes in one file
```

A diff might look like:

```
- learning_rate = 3e-4
+ learning_rate = 1e-4
```

The - line is removed; the + line is added.

4.3 Visualize the commit graph

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

For example:

```
* a34bd21 (HEAD -> main) Merge SAC implementation
|\
| * d72f8a1 Implement SAC algorithm
| * f61b2c9 Add replay buffer
|/
* b83fc92 Implement PP0
* 28ad613 Initial commit
```

Make a habit of combining `git status`, `git diff`, and `git log --oneline --graph --all`. Before each commit, confirm that you have not staged temporary artifacts, model weights, or secrets.

5. Syncing Remote Code and Pre-Commit Safety

5.1 Update local code: Fetch versus Pull

```
cd robot-policy
git fetch origin
```

`fetch` retrieves remote objects and updates remote-tracking references. It **does not automatically merge them into the current local branch**.

If your local `main` tracks `origin/main`, you may then run:

```
git merge origin/main
```

A common shorthand is:

```
git pull
```

`pull` performs `fetch` followed by integration, using merge or rebase according to the command options and Git configuration. To avoid an unexpected merge commit when updating a stable branch, beginners may prefer:

```
git pull --ff-only
```

This succeeds only when the update can be fast-forwarded; if the histories have diverged, Git stops and lets you decide how to integrate them.

Note: You normally cannot push directly to another team's open-source repository unless you have permission. You can Fork the repository first and work in your own Fork, or contribute through the project's pull request process.

5.2 Three checks before committing

1. `git status`: confirm the active branch, staged files, and untracked files.
2. `git diff --staged`: review exactly what the next commit will contain.
3. Never commit passwords, private keys, model checkpoints, large datasets, or training outputs. See Chapter 9 for `.gitignore` and Git LFS/DVC.

5.3 Minimal safe undo

If you staged a file accidentally, `git restore --staged FILE` only unstages it and preserves working-tree edits. Before discarding work or undoing a shared commit, consult the risk table in Chapter 8.

Part II: Research-Oriented Reference

Use this part as a reference when a real need arises; beginners do not need to memorize it.

6. New Repositories, SSH, and GitHub Authentication

6.1 Initialize a new research project

If no remote repository exists yet, create a local project:

```
mkdir robot-policy && cd robot-policy
git init
echo '# Robot Policy' > README.md
git add README.md
git commit -m "Initial commit"
```

Create an **empty repository without starter files** on GitHub, then configure `origin` and push using Section 6.3. Use `git clone` for existing repositories; do not run `git init` again.

6.2 Configure an SSH key

SSH authentication is convenient for long-term GitHub development on Ubuntu.

First check for existing keys:

```
ls -al ~/.ssh
```

If you do not already have a suitable Ed25519 key, generate one:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

Follow the prompts to choose the key location and passphrase. If a key file already exists, **do not overwrite it without checking**.

Start an SSH agent and load the private key:

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519
```

Display and copy the **public key**:

```
cat ~/.ssh/id_ed25519.pub
```

In GitHub, navigate to **Settings** → **SSH and GPG keys** → **New SSH key**, paste the public key, and save. The file `id_ed25519` is your **private key**: never publish or upload it. The `.pub` file is the one to share.

Test connectivity:

```
ssh -T git@github.com
```

On the first connection, verify GitHub's host fingerprint before accepting it. A successful result generally confirms authentication while noting that shell access is not available.

Reference: [GitHub's official SSH documentation](#).

6.3 Push a local project to GitHub

Create an **empty repository** named `robot-policy` on GitHub. If your local repository already has an initial commit, avoid adding a `README`, `.gitignore`, or License on the website during repository creation; doing so would create a separate initial history.

Then, locally:

```
cd robot-policy
git remote add origin git@github.com:YOUR_USERNAME/robot-policy.git
git remote -v
git branch -M main
git push -u origin main
```

Interpretation:

- **origin** is the conventional **name** of a remote; it is not a special server.
- `git remote -v` shows the fetch and push URLs.
- `git branch -M main` names the current branch **main**.
- `-u` establishes upstream tracking so later pushes usually need only `git push`.

6.4 Clone an open-source project

For example:

```
git clone https://github.com/ajwagen/dsrl.git
cd robot-policy
```

With SSH configured and appropriate access, you can instead use the repository's SSH URL:

```
git clone git@github.com:ajwagen/dsrl.git
```

`clone` downloads the repository history, creates the project directory, and normally configures an **origin** remote. Do **not** run `git init` again inside a cloned repository.

HTTPS	SSH
Public repositories can typically be cloned anonymously	Typically requires SSH authentication
Uses an <code>https://...</code> URL	Uses a <code>git@github.com:... URL</code>
Pushing needs appropriate account credentials	Key-based authentication makes pushing convenient

7. Branches, Team Collaboration, and Conflicts

Branches let you maintain a stable implementation and experimental features side by side. For example, **main** can keep stable PPO code while **feature/sac** contains a new SAC implementation:

```
A --- B ----- M      main
      \           /
      C --- D -----/      feature/sac
```

Here **M** is a possible merge commit. If the merge can be fast-forwarded, Git may not create a separate merge commit.

7.1 List, create, and switch branches

```
git branch          # List local branches
git branch feature/sac  # Create only
git switch -c feature/sac  # Create and switch (alternative)
git switch main      # Switch to main
git switch feature/sac  # Return to the feature branch
```

If you already created the branch using `git branch feature/sac`, switch with `git switch feature/sac`; do not run `switch -c` with the same name again.

Switching branches updates files in your working tree. Git may block the switch if an uncommitted modification would be overwritten. Commit the work or use `git stash` first.

7.2 Merge into the main branch

```
git switch main
git merge feature/sac
```

If there are no conflicts, Git completes the merge. Otherwise, see Section 7.5.

7.3 Delete a finished branch

```
git branch -d feature/sac
```

The `-d` option checks that the branch has been safely integrated. To delete the corresponding remote branch:

```
git push origin --delete feature/sac
```

For shared repositories, it is generally better to merge feature branches via a reviewed GitHub Pull Request into a protected `main` branch.

A research group might organize work as follows:

```
main
├─ feature/ppo
├─ feature/sac
├─ feature/pose-estimation
└─ feature/sim2real
```

A recommended workflow is **Feature Branch → Push → Pull Request → Review → Merge**.

7.4 Push the branch and open a PR

```
git push -u origin feature/pose-estimation
```

Create a Pull Request on GitHub, typically from `feature/pose-estimation` into `main`. Other team members can inspect changes, review code, discuss implementation details, run CI checks, and approve the integration.

A useful team rule: **do not make large, untested algorithm changes directly on a shared stable branch**. Validate them in isolation first.

A merge conflict may arise when two branches modify overlapping lines and Git cannot choose an automatic resolution.

Suppose two branches changed `train.py`:

```
<<<<<<< HEAD
learning_rate = 0.0001
=====
learning_rate = 0.0003
>>>>>>> feature/sac
```

- Between `<<<<<<< HEAD` and `=====` is the current branch's version.
- Between `=====` and `>>>>>>> feature/sac` is the incoming branch's version.

7.5 Resolve a conflict

1. Run `git status` to identify conflicting files.
2. Open the file and understand the intent of both changes.
3. Keep one side, or write a result that correctly combines both behaviors.
4. **Remove all conflict markers** (`<<<<<<<, =====, >>>>>>>`).
5. Run tests and validate the behavior.
6. Stage the resolved file and complete the merge commit.

For example, choose:

```
learning_rate = 0.0001
```

Then run:

```
git add train.py
git commit
```

To cancel an in-progress merge:

```
git merge --abort
```

For conflicts during a rebase, use `git rebase --continue` or `git rebase --abort` instead. `git merge --abort` is for a merge operation.

7.6 Resolve conflicts using VS Code

VS Code's Merge Editor compares the current version, the incoming version, and the merged result. Regardless of whether you use the editor or terminal, run tests afterward—especially for reward functions, action scaling, robot reference frames, and other changes whose semantics are not obvious from a text diff.

8. Undoing, Correcting, and Recovering History

Recovery depends on whether changes are **unstaged**, **staged**, **committed**, or **already shared**. The following commands are not interchangeable.

8.1 Unstaged changes: Discard one file's working-tree edits

```
| git restore train.py
```

This restores `train.py` in the working tree from the staging area. **Unbacked working-tree edits may be lost.**

8.2 Staged changes but no commit: Unstage

```
| git restore --staged train.py
```

The edits remain in the working tree, but they will not be included in the next commit unless staged again.

8.3 A local, unshared commit: Undo the commit but keep edits

Assume the latest commits are:

```
| a82bc31 Bad reward modification
| b41d2ab Working PPO version
```

Undo the last commit while retaining its staged changes:

```
| git reset --soft HEAD~1
```

Undo the last commit and unstage the changes, but retain the working-tree edits:

```
| git reset HEAD~1
```

Undo the last commit and discard the corresponding working-tree changes:

```
| git reset --hard HEAD~1
```

reset --hard is destructive. Verify that you do not need the affected code, experiment results, or uncommitted work before using it.

8.4 A pushed/shared commit: Use Revert

```
| git revert a82bc31
```

revert creates a new commit that attempts to cancel the changes introduced by an earlier commit, preserving an auditable shared history. Conflicts are possible if later commits have modified the same code.

Operation	Effect	Typical use
<code>git restore FILE</code>	Discard unstaged file edits	Abandon temporary local changes
<code>git restore --staged FILE</code>	Unstage, keep edits	Accidental staging
<code>git reset --soft HEAD~1</code>	Move branch pointer, keep staging	Refine unshared commits
<code>git reset HEAD~1</code>	Move pointer, keep working edits	Restage selected changes
<code>git reset --hard HEAD~1</code>	Move pointer and discard edits	Only when no recovery is needed
<code>git revert HASH</code>	Add a reverse commit	Undo an already shared commit

General principle: use **reset** cautiously for private history; prefer **revert** for shared history.

9. Ignoring Files, Git LFS, and DVC

Machine learning projects routinely produce gigabytes of datasets, checkpoints, videos, and logs. Such files are usually **not suitable for an ordinary source-code Git repository**.

9.1 A useful template

Create `.gitignore` in the project root:

```
touch .gitignore
nano .gitignore
```

Example:

```
# Python cache
__pycache__/
*.pyc

# Local environments
.venv/
venv/

# Secrets
.env
*.pem

# Model artifacts
*.pt
*.pth
*.ckpt
checkpoints/

# Training outputs
outputs/
logs/
wandb/

# Large datasets
data/
datasets/

# ROS recordings
*.bag
*.db3

# Optional IDE state
.vscode/
```

Adapt these rules to your project. Some projects intentionally version tiny test weights or share certain `.vscode/` settings, so blanket exclusions may not be appropriate.

9.2 What `.gitignore` does—and does not—do

Ignore rules mostly affect **untracked** files. If `checkpoint.pt` is already tracked, adding `*.pt` to `.gitignore` does not automatically stop tracking it.

To remove a file from Git tracking while keeping the local copy:

```
git rm --cached checkpoint.pt
git commit -m "Stop tracking checkpoint"
```

This does not remove old copies from existing commits. If you accidentally published an API key, password, or other secret, revoke/rotate the affected credentials promptly and evaluate whether public history must be cleaned.

9.3 Separate source code, data, and models

Recommended division:

- Git: source code, configurations, scripts, documentation, and small test assets.
- Object storage or experiment tracking: checkpoints, videos, logs, and large datasets.

- Experiment metadata: Commit ID, dataset version, weight checksum, and training parameters.

For versioned large artifacts, separately consider Git LFS, DVC, or a specialized model/data management solution.

9.4 Choosing Git LFS, DVC, or Object Storage

Option	Best suited for	Important caveat
Plain Git + <code>.gitignore</code>	Code, config, docs; ignore transient outputs	Ignore rules do not remove previous history
Git LFS	Larger binary assets that need Git-associated versions	Git stores pointers; the binary lives in LFS storage with provider support and quotas
DVC + remote storage	Large datasets and versioned pipeline metadata	Git stores <code>.dvc</code> metadata; the actual data needs configured remote storage
Object storage / experiment platform	Very large videos, checkpoints, and logs	Record versions, paths, and checksums rather than a mutable URL alone

Minimal Git LFS example (install Git LFS first and ensure your host supports it):

```
git lfs install
git lfs track "*.onnx"
git add .gitattributes models/policy.onnx
git commit -m "Track policy model via LFS"
```

With DVC, use `dvc add datasets/trajectories` on a data directory and commit the generated `.dvc` metadata and ignore rules to Git. You must also configure and synchronize a DVC remote so other machines can obtain the actual data. **Neither LFS nor DVC automatically provides unlimited remote storage.**

10. Stash, Rebase, Show, and Blame

10.1 git stash: Temporarily shelve uncommitted work

Suppose you are implementing SAC but must switch to `main` to fix an urgent issue:

```
git stash
```

By default, this mainly saves changes to tracked files. Include untracked files by running:

```
git stash push -u -m "WIP SAC training"
```

Then:

```
git switch main
# Fix the issue and commit it
git switch feature/sac
git stash pop
```

Command	Purpose
<code>git stash list</code>	List saved stashes
<code>git stash show -p</code>	Inspect the most recent stash's patch
<code>git stash pop</code>	Restore a stash and remove it if successful
<code>git stash apply</code>	Restore a stash but keep the stash record

Stash application can also conflict; it is not a substitute for reliable long-term backups.

10.2 **git rebase**: Move experimental work onto a newer base

Original history:

```
main:    A -- B -- C
feature: A -- B -- D -- E
```

Run:

```
git switch feature
git rebase main
```

Conceptually, the result is:

```
main:    A -- B -- C
feature: A -- B -- C -- D' -- E'
```

rebase replays changes and creates new commit IDs. It is useful for organizing **your own private branches**, but do not casually rewrite shared history that other developers are using.

10.3 Inspect a commit or trace a changed line

```
git show <commit-id>      # Full details of one commit
git blame train.py         # Last modifying commit for each line
```

git blame can help answer “when was reward scaling changed?” or “which commit changed the control frequency?” It is a historical diagnostic tool, not a judgment about individual responsibility.

10.4 Inspect local and remote branches

```
git branch -a
```

Example:

```
* main
  feature/sac
remotes/origin/main
remotes/origin/feature/ppo
```

Remote-tracking references (`remotes/origin/...`) represent Git’s locally recorded state of remote branches. Use **git fetch** to refresh them.

11. Version Tracking and Reproducible Research

11.1 Name branches by the research task

Research scenario	Suggested branch
Implement PPO or SAC	feature/ppo / feature/sac
Explore a different reward	experiment/reward-v2
Adapt an online residual velocity field	experiment/residual-velocity
Update pose estimation or camera calibration	feature/pose-estimation
Fix deployed robot control latency	fix/control-latency
Reproduce a paper’s experiment	Pin the code using a Commit ID or a Tag

11.2 Record the Commit ID with every training run

```
git rev-parse HEAD
```

Save it alongside the experiment configuration, for example:

```
experiment:
  name: residual_velocity_v3
  git_commit: a82bc31
  random_seed: 42
```

```
training:
  learning_rate: 0.0003
  batch_size: 256
```

The short hash above is illustrative; for production experiment records, store the **full hash** returned by the command.

Important: A Commit ID identifies committed code only. If you launch an experiment with uncommitted modifications, the hash alone cannot fully reproduce that run. Check `git status` before official runs, and record a patch or create a dedicated experiment commit if needed.

11.3 Separate code, configuration, and large files

- **Git repository:** Source code, configs, entry points, evaluation scripts, README, environment documentation.
- **Experiment output storage:** TensorBoard/W&B logs, videos, weights, and large caches—usually ignored by Git.
- **Reproducibility metadata:** Commit ID, environment/dependency versions, dataset version, seed, simulator version, and GPU/driver information.
- **Robot deployment:** Tag a verified policy release (e.g., `v1.0-deploy`) and store compatible controller settings and hardware versions.

Example: tag a verified deployment version:

```
git tag -a v1.0-deploy -m "Validated robot deployment"
git push origin v1.0-deploy
```

11.4 What metadata must accompany a research run?

Bind every simulation or robot run to the **full Commit ID**, **clean/dirty working-tree state**, **config files**, **random seeds**, **dependency and simulator versions**, **dataset versions/checksums**, and **checkpoint hash**. For real robots also record consequential URDF, firmware, camera extrinsic, and controller-rate settings.

A Commit ID identifies only committed code. If `git status` shows uncommitted edits, retain a patch or make a traceable commit. The Commit ID does not replace data and environment versioning.

12. Common Errors and Safe Troubleshooting

12.1 fatal: not a git repository

Git cannot find a valid repository in the current directory or its parents. Check:

```
pwd
ls -a
```

Make sure you are in the project directory. Run `git init` only if you actually intend to create a **new** repository.

12.2 Permission denied (publickey)

SSH authentication failed. Check:

```
ssh -T git@github.com
```

Verify that the SSH agent has your correct private key loaded and that its corresponding public key is attached to the intended GitHub account.

12.3 Updates were rejected

This usually means that the remote has commits missing from your local history, so the push cannot fast-forward. Inspect first:

```
git fetch origin
git status
```

For your own feature branch with a correctly configured upstream, you can choose this workflow if it matches your team's conventions:

```
git pull --rebase
# Resolve potential conflicts and run tests
```

```
| git push
```

Do not blindly use `git push --force` just to make the error disappear; it can overwrite collaborators' commits.

12.4 Your local changes would be overwritten

Uncommitted local changes might be overwritten by a switch or update. Choose one of the following before retrying:

```
| # Option A: Commit the current work
| git add .
| git commit -m "Save work in progress"
|
| # Option B: Temporarily stash it, including untracked files
| git stash push -u
```

12.5 A general troubleshooting rule

When Git reports an unfamiliar error, start with `git status`, read its suggested next steps, and determine whether you are in a merge, rebase, or ordinary working-tree state. Avoid chains of `reset --hard`, `clean -fd`, or force-push commands when you do not understand the current state.

13. Git in VS Code

Open the repository with:

```
| code .
```

Open the **Source Control** pane, commonly with `Ctrl + Shift + G`.

Action	VS Code workflow
View changed files	Source Control file list
Compare versions	Click a modified file in the list
Stage edits	+ / Stage Changes
Commit	Enter a message and choose Commit
Push	Push or Sync Changes
Switch branches	Status bar branch label or Command Palette
Resolve conflicts	Merge Editor

For example, change:

```
| reward = success_reward
```

to:

```
| reward = success_reward + progress_reward
```

VS Code's Diff Editor clearly shows the removed and added lines. For RL experiments with many configuration files, reviewing the complete diff before each commit often reveals accidental hyperparameter changes.

Recommended combination: Use the terminal for reproducible commands and precise control, and use VS Code to review large diffs and resolve conflicts.

14. Advanced Command Reference and Official Docs

Category	Command	Purpose
Setup	<code>git init</code>	Create a repository
Clone	<code>git clone URL</code>	Download a repository
Status	<code>git status</code>	Inspect current state
Diff	<code>git diff</code>	Inspect unstaged changes

Category	Command	Purpose
History	<code>git log --oneline</code>	Browse commit history
Stage	<code>git add .</code>	Stage changes in the current directory
Commit	<code>git commit -m "msg"</code>	Create a commit
Publish	<code>git push</code>	Send local commits to a remote
Fetch	<code>git fetch</code>	Retrieve remote state without merging
Sync	<code>git pull --ff-only</code>	Fast-forward-only update
Branch	<code>git branch</code>	List local branches
Branch	<code>git switch -c name</code>	Create and switch branch
Branch	<code>git switch name</code>	Switch to an existing branch
Merge	<code>git merge name</code>	Merge a named branch
Undo	<code>git restore file</code>	Discard unstaged file edits
Undo	<code>git restore --staged file</code>	Unstage a file
Undo	<code>git revert HASH</code>	Add a reversing commit
Shelf	<code>git stash</code>	Temporarily save edits
Shelf	<code>git stash pop</code>	Restore a stash
Inspect	<code>git show HASH</code>	Examine a commit
Inspect	<code>git blame file</code>	Inspect per-line commit attribution

14.1 Official references

- [Pro Git \(Chinese\)](#)
- [Pro Git \(English\)](#)
- [Git Command Reference](#)
- [GitHub: Connect with SSH](#)
- [GitHub: Pull Request Documentation](#)

Conclusion: Git’s main contribution to research is not simply “uploading code to GitHub.” It is connecting **source-code versions, experimental changes, and reproducible results**. Start with small commits, separate experiment branches, review diffs, and keep large artifacts out of the main source repository. Add team collaboration practices as your projects grow.