RABBIT HOLES

Accelerate · Manipal University Jaipur

11

PHASE

Configuration and automation

Aliases, attributes, and hooks that stop a bad commit before it exists.

Part of **git started**, Accelerate's third workshop.
github.com/accelerate-muj/git-started · MIT licensed.

Contents

1 Four levels of config 3

1.1 Different identities for different work 3

2 Settings worth turning on 3

2.1 Line endings 4

3 Aliases 4

4 Ignoring files 5

4.1 Precedence 5

4.2 Patterns 5

4.3 A global ignore 5

5 gitattributes 6

6 Hooks 6

6.1 The useful client-side hooks 7

6.2 Commit message conventions 7

7 Reference 8

1 Four levels of config

Git reads settings from several files and the most specific wins.

Level	File	Applies to
`--system`	`/etc/gitconfig`	Every user on the machine
`--global`	`~/.gitconfig`	You, everywhere
`--local`	`.git/config`	One repository
`--worktree`	`.git/config.worktree`	One worktree

```
git config --list --show-origin
git config --get user.email
git config --global --edit
```

`--show-origin` is the one to remember. When a setting is not doing what you expect, it tells you which file it came from, and the answer is usually a `--local` value you set months ago and forgot.

1.1 Different identities for different work

Personal projects under one email, university work under another:

```
[includeIf "gitdir:~/work/"]
    path = ~/.gitconfig-work
```

Put that in `/.gitconfig`, put the work identity in `/.gitconfig-work`, and git switches automatically based on where the repository lives. The trailing slash matters.

2 Settings worth turning on

```
git config --global init.defaultBranch main
git config --global pull.rebase true
git config --global fetch.prune true
git config --global rebase.autoSquash true
git config --global rerere.enabled true
git config --global merge.conflictStyle zdiff3
git config --global diff.algorithm histogram
git config --global push.autoSetupRemote true
git config --global column.ui auto
```

```
git config --global help.autocorrect 20
```

Three of those deserve a note.

`push.autoSetupRemote` means `git push` works on a brand new branch without `-u origin <name>`. It removes one of the most common small frictions in daily use.

`diff.algorithm histogram` produces noticeably better diffs on code that has been moved or re-indented, at no real cost.

`help.autocorrect 20` makes git run the obvious correction after two seconds when you typo a command. Set it to `0` if you find that alarming rather than helpful.

2.1 Line endings

The classic cross-platform problem: Windows uses CRLF, everything else uses LF, and without configuration every file shows as entirely modified when the other person touches it.

```
git config --global core.autocrlf true
git config --global core.autocrlf input
```

`true` on Windows, `input` on macOS and Linux.

> **NOTE**
>
> Better still, do not rely on each person configuring it. Put a `.gitattributes` file in the repository with `* text=auto eol=lf` and the rule travels with the project, applying to everyone regardless of their settings. This repository does exactly that.

3 Aliases

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.lg "log --graph --oneline --decorate --all"
git config --global alias.last "log -1 HEAD --stat"
git config --global alias.unstage "restore --staged"
git config --global alias.amend "commit --amend --no-edit"
```

Aliases starting with `!` run a shell command instead, from the repository root:

```
git config --global alias.root "!pwd"
git config --global alias.cleanup "!git branch --merged main | grep -v main | xargs -r git branch -
    ↪ d"
git config --global alias.sync "!git fetch --prune && git pull --rebase"
```

That `cleanup` one deletes every local branch already merged into `main`, which after an active month is most of them. Read it before you run it, which is good advice for any alias you copy from a document.

TRY IT

Add this one and use it for a week:

```
git config --global alias.wip "!git add -A && git commit -m 'wip: save point'"
```

A single word that snapshots everything before you try something risky. Phase 4 explains why this makes you recoverable, and Phase 5 explains how to squash the `wip` commits away before anyone sees them.

4 Ignoring files

4.1 Precedence

Git checks several sources, most specific first:

1. `.gitignore` in the same directory, then each parent directory
2. `.git/info/exclude`, local to your clone and never committed
3. The file named by `core.excludesFile`, your global ignore

4.2 Patterns

```
*.log              all .log files, any directory
build/             a directory anywhere
/build             build only at the repository root
!important.log     re-include something a previous rule excluded
doc/**/*.pdf       PDFs at any depth under doc/
temp?.txt          temp1.txt, tempA.txt
```

TRAP

`.gitignore` only affects **untracked** files. Once a file is tracked, git keeps tracking it no matter what you add to the ignore list. This is the single most common confusion with the feature.
To stop tracking a file while keeping it on disk:

```
git rm --cached config.local.json
git commit -m "Stop tracking local config"
```

The `--cached` is essential. Without it, `git rm` deletes the file.

4.3 A global ignore

Editor and OS clutter belongs in your global ignore, not in every project's `.gitignore`. Nobody else should have to accommodate your editor.

```
git config --global core.excludesFile ~/.gitignore_global
```

Put `.DS_Store`, `Thumbs.db`, `.idea/`, `*.swp` in there.

5 gitattributes

`.gitignore` decides which files git sees. `.gitattributes` decides how git treats them. It is committed, so it applies to everyone.

```
* text=auto eol=lf
*.png binary
*.pdf binary
*.md diff=markdown
*.py diff=python
package-lock.json linguist-generated=true
CHANGELOG.md merge=union
docs/ export-ignore
*.psd filter=lfs diff=lfs merge=lfs -text
```

Three of those are worth explaining.

`diff=python` tells git the language, so diff hunk headers show the enclosing function name instead of a meaningless nearby line. Small change, and it makes reviewing much easier.

`linguist-generated=true` collapses generated files in GitHub's diff view and excludes them from the language statistics. Lock files are the usual case.

`merge=union` on a changelog makes git keep *both* sides of a conflict automatically instead of stopping. For an append-only list of lines this is usually right, and it would have removed the conflicts from Phase 3, which is exactly why we did not use it there.

6 Hooks

Scripts git runs at specific moments. They live in `.git/hooks` and there are samples there already.

```
ls .git/hooks
```

Remove the `.sample` suffix and make it executable to enable one.

6.1 The useful client-side hooks

Hook	Runs
`pre-commit`	Before the message is written. Exit non-zero to cancel. Linting and secret scanning go here.
`prepare-commit-msg`	Before the editor opens. Use it to insert a template or a ticket number.
`commit-msg`	After the message is written. Enforce a format.
`pre-push`	Before a push. Run the test suite here, not in `pre-commit`.
`post-checkout`	After switching branches. Reinstall dependencies.

A `pre-commit` hook that blocks obvious secrets:

```
#!/bin/sh
if git diff --cached | grep -nE "(api[_-]?key|secret|password|BEGIN.*PRIVATE KEY)"; then
  echo "Possible secret in staged changes. Commit blocked."
  echo "If this is a false positive: git commit --no-verify"
  exit 1
fi
```

TRAP

Hooks live in `.git/hooks`, which is not part of the repository and does not get cloned. A hook you write protects you and nobody else.
Two ways round it. Point git at a directory that *is* committed:

```
git config core.hooksPath .githooks
```

Or use a framework that installs them on `npm install` or equivalent: Husky for JavaScript, `pre-commit` for Python and general use.
Also remember `--no-verify` skips every hook. Hooks are a convenience that catches mistakes, never a security control. Anything that genuinely must hold has to run in CI, where the contributor cannot skip it. Phase 14 covers that.

6.2 Commit message conventions

Conventional Commits is the widely used format:

```
feat(parser): support blank lines in sonnets
fix: correct off-by-one in line counter
docs: explain the force-with-lease difference
refactor!: drop support for the old config format
```

The prefix is machine readable, which is the point: tools derive the next version number and generate a changelog from it. `feat` bumps MINOR, `fix` bumps PATCH, and `!` or a `BREAKING CHANGE:` footer bumps MAJOR.

Enforce it in `commit-msg` if your team wants it. Consider whether you want it first, because on a small project it can be ceremony without benefit.

7 Reference

Goal	Command
Where did this setting come from?	`git config --list --show-origin`
Different identity per directory	`includeIf "gitdir:~/work/"`
Push a new branch with no flags	`git config --global push.autoSetupRemote true`
Make an alias	`git config --global alias.lg "log --graph ..."`
Alias that runs a shell command	prefix the value with `!`
Stop tracking a committed file	`git rm --cached <file>`
Ignore editor clutter everywhere	`git config --global core.excludesFile ...`
Share hooks with the team	`git config core.hooksPath .githooks`
Skip hooks once	`git commit --no-verify`

CHECK YOURSELF

1. Four config levels. Which wins?
2. You added `config.json` to `.gitignore` and git still tracks it. Why?
3. What does `.gitattributes` do that `.gitignore` cannot?
4. Why can a `pre-commit` hook never be a security control?
5. Which hook should run your test suite, and why not `pre-commit`?

Next: Phase 12, *GitHub as a platform*, which leaves git behind entirely and covers the things GitHub added on top: issues, projects, pages, and a search syntax worth learning properly.