Git & Version Control

Guida Completa al Controllo di Versione

15 novembre 2025

Indice

Prefazione

A chi si rivolge questo manuale

Questi appunti sono dedicati a studenti, sviluppatori e chiunque voglia comprendere a fondo Git e il controllo di versione. Il percorso è strutturato per accompagnare progressivamente dalle basi teoriche del version control alla gestione avanzata di repository collaborativi, workflow moderni e best practices professionali.

Perché Git è fondamentale

Git è lo standard de facto per il controllo di versione nel mondo dello sviluppo software. Non si tratta solo di un tool tecnico, ma di una competenza professionale essenziale che:

- Permette di tracciare ogni modifica al codice nel tempo
- Facilita la collaborazione tra team distribuiti geograficamente
- Abilita workflow complessi di sviluppo (feature branches, hotfix, release)
- Fornisce sicurezza contro perdite di dati e regressioni
- È richiesto praticamente da ogni azienda tech nel mondo

Struttura del corso

Il corso è organizzato in 10 capitoli che coprono l'intero ecosistema Git:

Parte I - Fondamenti (Capitoli 1-2)

- Storia e filosofia del version control
- Differenze tra VCS centralizzati e distribuiti
- Concetti fondamentali: repository, commit, staging area
- Primi comandi Git: init, add, commit, status, log
- File .gitignore e gestione file da escludere

Parte II - Branching e Merging (Capitolo 3)

- Creazione e gestione branch
- Switching tra branch con checkout/switch
- Merge: fast-forward e three-way merge
- Rebase per linearizzare la storia

- Risoluzione conflitti

Parte III - Repository Remoti (Capitolo 4)

- Configurazione remote
- Clone di repository esistenti
- Fetch, pull e push
- Tracking branches
- Best practices per sincronizzazione

Parte IV - Workflow Collaborativi (Capitolo 5)

- Git Flow: workflow strutturato per release
- GitHub Flow: workflow semplificato per CI/CD
- Trunk-Based Development
- Fork e Pull Request
- Code review e collaborazione

Parte V - Argomenti Avanzati (Capitoli 6-10)

- Git avanzato: stash, cherry-pick, bisect, reflog
- GitHub e GitLab: piattaforme collaborative
- Best practices professionali
- Troubleshooting e recovery
- Introduzione a CI/CD con Git

Prerequisiti

Per affrontare questo corso è consigliabile avere:

- Familiarità con la riga di comando (terminal/bash/cmd)
- Conoscenza base di un linguaggio di programmazione
- Comprensione dei concetti di file e directory
- (Opzionale) Esperienza con sviluppo software collaborativo

Strumenti necessari

Software indispensabile:

- **Git**: Sistema di controllo versione (2.30+)
- **Terminal/Bash**: Shell per eseguire comandi Git
- **Editor di testo**: VS Code, Vim, Nano, Sublime Text

Piattaforme Git (consigliato):

- **GitHub**: Piattaforma leader per hosting repository
- **GitLab**: Alternativa completa con CI/CD integrato
- **Bitbucket**: Soluzione Atlassian per team enterprise

GUI Git (opzionale):

- **GitKraken**: Client grafico potente e intuitivo
- **SourceTree**: GUI gratuita di Atlassian
- **GitHub Desktop**: Integrazione nativa GitHub
- **VS Code Git**: Integrazione Git in VS Code

Come installare Git

Linux (Ubuntu/Debian):

```
sudo apt update
sudo apt install git
```

macOS:

```
brew install git
```

Windows: Scarica Git for Windows da https://git-scm.com/download/win
Verifica installazione:

```
git --version
# Output: git version 2.39.0 (o superiore)
```

Come studiare

Per ottenere il massimo da questi appunti:

1. **Leggi attentamente la teoria**: Ogni concetto è spiegato con diagrammi e esempi
2. **Pratica ogni comando**: Crea repository di prova e sperimenta
3. **Disegna i diagrammi**: Visualizza branch, merge e workflow su carta
4. **Commetti errori**: Sbagliare è parte dell'apprendimento (usa repository di test!)
5. **Risolvi gli esercizi**: Prova autonomamente prima di vedere le soluzioni
6. **Collabora**: Trova un partner e praticate workflow collaborativi
7. **Leggi i log**: Analizza la storia dei commit per capire il flusso

Nota Importante

Git ha una curva di apprendimento ripida inizialmente. **Non scoraggiarti!** La maggior parte degli sviluppatori usa quotidianamente solo 10-15 comandi. Questo manuale copre tutto lo spettro, ma inizierai con i comandi base e acquisirai padronanza gradualmente.

Convenzioni tipografiche

Nel testo vengono utilizzate le seguenti convenzioni:

- `Comandi Git`: In carattere monospace (`git commit`)
- **Concetti chiave**: In grassetto (repository, branch, commit)
- *Parametri/opzioni*: In corsivo (-m, –amend)
- Box colorati: Best Practices, Attenzioni, Errori Comuni, Esempi

Struttura comandi:

```
$ git comando [opzioni] <argomenti>
# Commento esplicativo
```

Il simbolo `$` indica il prompt della shell (non va digitato).

Filosofia di Git

Git è stato creato da Linus Torvalds nel 2005 per gestire lo sviluppo del kernel Linux. I principi fondamentali sono:

Distribuito Ogni sviluppatore ha una copia completa della storia

Veloce Operazioni locali senza latenza di rete

Integrità Checksum SHA-1 garantisce integrità dei dati

Immutabilità La storia non viene riscritta (salvo operazioni esplicite)

Branching leggero Creare branch è veloce e incoraggiato

Sito web e risorse

Materiale aggiuntivo disponibile su:

- Repository GitHub: https://github.com/campionluca/Appunti
- Documentazione ufficiale Git: https://git-scm.com/doc
- Pro Git Book (gratuito): https://git-scm.com/book/it/v2
- GitHub Learning Lab: https://lab.github.com
- Visualizing Git: https://git-school.github.io/visualizing-git/

Ringraziamenti

Si ringrazia:

- Linus Torvalds e la community Git per aver creato questo straordinario tool
- La community open source per innumerevoli tutorial e risorse
- L'Istituto Tecnico Antonio Scarpa per il supporto nella realizzazione

- Tutti gli studenti che con domande e feedback hanno migliorato questo materiale

Prof. Luca Campion
Novembre 2025

Note sulla versione

Versione 1.0 - Novembre 2025

- Prima release completa
- 10 capitoli + 2 appendici
- Coverage: Git base, branching, remote, workflow, advanced
- Esempi pratici con Git 2.30+
- Diagrammi TikZ per visualizzare branch e merge
- Esercizi progressivi per ogni capitolo
- Best practices professionali aggiornate

Convenzioni usate negli esempi

Repository di esempio: Useremo repository di prova chiamati:

- `mio-progetto`: Repository locale base
- `website`: Progetto web di esempio
- `app`: Applicazione esempio per branching

Utenti di esempio:

- Alice: Sviluppatrice senior
- Bob: Sviluppatore junior
- Charlie: DevOps engineer

Branch comuni:

- `main/master`: Branch principale di produzione
- `develop`: Branch di sviluppo
- `feature/nome`: Branch per nuove funzionalità
- `hotfix/nome`: Branch per fix urgenti

Prima di iniziare

Configura Git con il tuo nome e email (richiesto per commit):

```
$ git config --global user.name "Tuo Nome"
```

```
$ git config --global user.email "tua.email@example.com"
```

Verifica la configurazione:

```
$ git config --list
```

Licenza e utilizzo

Buono studio e buon Git!

Capitolo 1

Introduzione a Git e Version Control

Introduzione

Il version control (controllo di versione) è una pratica fondamentale nello sviluppo software moderno. Git rappresenta lo standard de facto per tracciare modifiche al codice, collaborare con altri sviluppatori e gestire progetti di qualsiasi dimensione. Questo capitolo introduce i concetti fondamentali del version control, la storia di Git e le differenze tra sistemi centralizzati e distribuiti.

Obiettivi di apprendimento

- Comprendere cos'è il version control e perché è necessario
- Conoscere la storia e l'evoluzione di Git
- Distinguere tra VCS centralizzati e distribuiti
- Comprendere l'architettura di Git
- Conoscere i vantaggi di Git rispetto ad altri VCS
- Configurare Git per la prima volta

1.1 Cos'è il Version Control

1.1.1 Definizione

Un **Version Control System (VCS)** è un software che registra le modifiche ai file nel tempo, permettendo di richiamare versioni specifiche in un secondo momento. È come una macchina del tempo per il codice sorgente.

1.1.2 Il problema senza version control

Immagina di lavorare su un progetto senza version control:

Listing 1.1: Cartella progetto senza VCS

```
mio-progetto/
  main.py
  main_v2.py
  main_v2_final.py
  main_v2_final_REALMENTE_FINALE.py
  main_v2_final_REALMENTE_FINALE_fix.py
  main_backup_20241110.py
```

Problemi evidenti:

- Impossibile sapere cosa è cambiato tra versioni
- Nessuna traccia di chi ha fatto modifiche
- Difficile collaborare (chi ha la versione corretta?)
- Nessun modo di tornare indietro facilmente
- Naming chaos totale

Errore Comune: Version Control Manuale

Creare copie manuali dei file con suffissi come `_v1`, `_final`, `_backup` è inefficiente, error-prone e non scala. Il version control automatizza questo processo in modo robusto e professionale.

1.1.3 La soluzione: Version Control System

Con un VCS come Git:

- **Tracking automatico**: Ogni modifica è tracciata con timestamp e autore
- **Commit messages**: Descrizioni di cosa è stato modificato e perché
- **Branching**: Lavorare su feature diverse in parallelo senza conflitti
- **Rollback**: Tornare a versioni precedenti istantaneamente
- **Collaborazione**: Più persone lavorano sugli stessi file senza sovrascriversi
- **Storia completa**: Audit trail di tutto il progetto

Esempio Pratico

Stai sviluppando una funzionalità ma scopri un bug critico in produzione. Con Git:

1. Salvi il lavoro corrente (`git stash`)
2. Torni alla versione di produzione (`git checkout main`)
3. Crei un hotfix branch (`git checkout -b hotfix/critical-bug`)
4. Fixxi il bug e fai push
5. Torni alla tua feature (`git checkout feature/nuova-funzione`)
6. Recuperi il lavoro salvato (`git stash pop`)

Tutto in pochi secondi, senza perdere nulla!

1.2 Storia di Git

1.2.1 Pre-Git: BitKeeper e la crisi del 2005

Il kernel Linux (1991-2005) inizialmente usava patch e archivi tar per gestire contributi. Nel 2002, il progetto adottò BitKeeper, un VCS proprietario che offriva licenze gratuite ai sviluppatori open source.

La crisi del 2005: BitKeeper revocò le licenze gratuite dopo controversie con la community. Linus Torvalds si trovò senza VCS adeguato per gestire migliaia di contributi al kernel.

1.2.2 Nascita di Git (Aprile 2005)

Linus Torvalds decise di creare un nuovo VCS con questi obiettivi:

- **Velocità**: Operazioni locali istantanee
- **Design semplice**: Architettura comprensibile
- **Supporto branching non lineare**: Migliaia di branch paralleli
- **Completamente distribuito**: Nessun server centrale obbligatorio
- **Gestione progetti grandi**: Kernel Linux ha 25+ milioni linee di codice

Timeline:

- **3 Aprile 2005**: Linus inizia a scrivere Git
- **7 Aprile 2005**: Git diventa self-hosting (gestisce il proprio codice)
- **16 Giugno 2005**: Kernel Linux 2.6.12 rilasciato usando Git
- **21 Luglio 2005**: Linus passa la manutenzione a Junio Hamano
- **2008-oggi**: Git diventa lo standard de facto

Fun Fact

Linus Torvalds ha scritto la prima versione funzionante di Git in **10 giorni**. Il nome "git" è slang britannico per "persona testarda/antipatica" - Linus scherzosamente dice: "I name all my projects after myself: first Linux, now Git."

1.2.3 Evoluzione e adozione

2008 GitHub lanciato - hosting gratuito per repository Git

2011 GitLab lanciato - alternativa self-hosted

2018 Microsoft acquisisce GitHub per 7.5 miliardi di dollari

2020+ Git è usato da 90%+ dei progetti software professionali

1.3 VCS Centralizzati vs Distribuiti

1.3.1 Version Control System Centralizzati (CVCS)

Esempi: **SVN (Subversion)**, CVS, Perforce

Architettura:

Caratteristiche CVCS:

- Un server centrale contiene tutta la storia
- I client hanno solo l'ultima versione (working copy)
- Ogni operazione richiede connessione al server
- Commit = push immediato al server

Vantaggi CVCS:

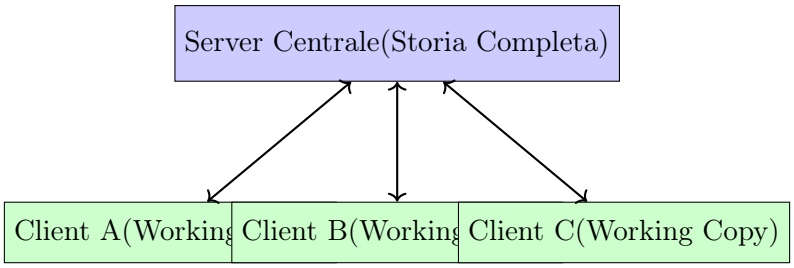


Figura 1.1: Architettura CVCS - Server Centrale

- Amministrazione centralizzata (controllo accessi)
- Modello mentale semplice
- Backup centralizzato

Svantaggi CVCS:

- **Single Point of Failure**: Server down = nessuno lavora
- **Richiede rete**: Impossibile commit offline
- **Lento**: Ogni diff/log richiede query al server
- **Branching costoso**: Creare branch è pesante
- **Rischio perdita dati**: Se il server crasha, si perde tutto

1.3.2 Version Control System Distribuiti (DVCS)

Esempi: **Git**, Mercurial, Bazaar

Architettura:

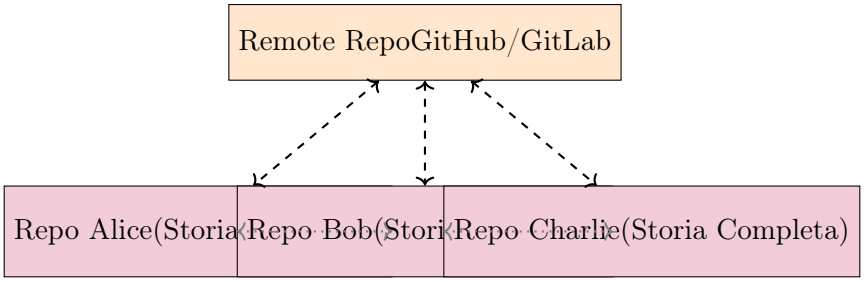


Figura 1.2: Architettura DVCS - Distribuita

Caratteristiche DVCS:

- Ogni clone è una copia completa del repository (storia inclusa)
- Operazioni locali (commit, branch, merge) istantanee
- Sincronizzazione esplicita (push/pull)
- Possibilità di lavorare offline

Vantaggi DVCS:

- **Backup naturale**: Ogni clone è un backup completo

- **Velocità**: Operazioni locali senza latenza di rete
- **Offline-first**: Lavoro completo senza connessione
- **Branching leggero**: Creare branch è istantaneo
- **Flessibilità workflow**: Supporta workflow complessi
- **Integrità**: Checksum garantisce nessuna corruzione

Svantaggi DVCS:

- Curva di apprendimento più ripida
- Concetti più complessi (staging area, rebase, etc.)
- Storage locale maggiore (tutta la storia)

Confronto CVCS vs DVCS

Caratteristica	CVCS (SVN)	DVCS (Git)
Velocità operazioni	Lenta (rete)	Veloce (locale)
Lavoro offline	Limitato	Completo
Branching	Pesante	Leggero
Backup	Centralizzato	Distribuito
Curva apprendimento	Bassa	Media-Alta
Integrità dati	Buona	Eccellente

1.4 Architettura e Concetti Fondamentali di Git

1.4.1 I tre stati di Git

Git ha tre stati principali in cui i file possono trovarsi:

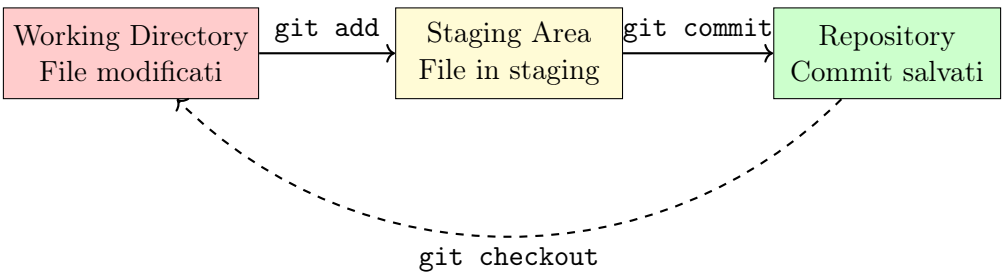


Figura 1.3: I tre stati di Git

Working Directory La directory di lavoro dove modifichi i file

Staging Area (Index) Area intermedia dove prepari i file per il commit

Repository (.git directory) Database dove Git salva snapshot (commit)

1.4.2 Workflow base di Git

1. **Modifichi** file nella working directory
2. **Staging**: Aggiungi snapshot dei file alla staging area (`git add`)
3. **Commit**: Salvi permanentemente snapshot nel repository (`git commit`)

Listing 1.2: Workflow tipico Git

```
# Modifica file
$ echo "Hello Git" > README.md

# Visualizza stato
$ git status
# Output: README.md modificato (modified)

# Aggiungi alla staging area
$ git add README.md

# Visualizza stato
$ git status
# Output: README.md pronto per commit (staged)

# Commit con messaggio
$ git commit -m "Add greeting to README"

# Visualizza stato
$ git status
# Output: Working directory clean
```

1.4.3 Snapshot, non differenze

Differenza fondamentale: La maggior parte dei VCS (SVN, CVS) memorizza informazioni come lista di modifiche ai file (delta-based).

Git invece pensa ai dati come **snapshot** del filesystem. Ogni commit è una foto completa di tutti i file in quel momento.

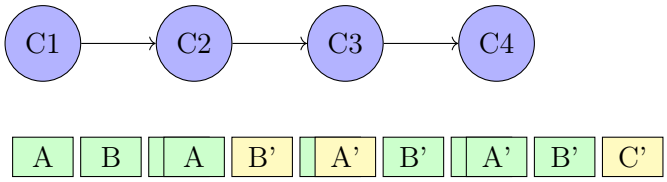


Figura 1.4: Git: Snapshot completi ad ogni commit (giallo = modificato)

Vantaggi snapshot:

- Checkout velocissimo (copia file, non ricostruisce da delta)
- Branching istantaneo (solo puntatori)
- Integrità garantita (checksum SHA-1 su tutto)

1.4.4 Integrità e checksum SHA-1

Ogni oggetto in Git è identificato da un **hash SHA-1** (160 bit, 40 caratteri esadecimali).

Listing 1.3: Esempio commit hash

```
$ git log --oneline
a3f5b21 Add user authentication feature
b8c9d34 Fix navbar responsive layout
c7e2a19 Initial commit
```

Proprietà SHA-1:

- Univoco: Probabilità collisione ≈ 0
- Immutabile: Modificare un byte cambia completamente l'hash
- Verificabile: Corruzione dati è immediatamente rilevata

Best Practice: Integrità

Git garantisce che non puoi modificare contenuti senza che Git lo sappia. È praticamente impossibile corrompere dati o avere modifiche non tracciate. Questo rende Git estremamente affidabile per progetti critici.

1.5 Perché Git è lo standard

1.5.1 Confronto con alternative

Git vs SVN Git è più veloce, supporta branching leggero, lavoro offline completo

Git vs Mercurial Sintassi simile, Git ha ecosistema più grande (GitHub)

Git vs Perforce Git è gratuito e open source, Perforce migliore per asset binari enormi

1.5.2 Adozione nel mondo reale

Statistiche:

- **90%+** dei progetti software usano Git
- **100 milioni+** repository su GitHub (2023)
- Progetti famosi: Linux Kernel, Android, Windows, VS Code, React, TensorFlow

1.6 Configurazione Iniziale di Git

1.6.1 Configurazione utente (obbligatoria)

Ogni commit in Git contiene informazioni sull'autore. Configurazione richiesta:

Listing 1.4: Configurazione base Git

```
# Configura nome (apparirà nei commit)
$ git config --global user.name "Mario Rossi"

# Configura email (apparirà nei commit)
$ git config --global user.email "mario.rossi@example.com"

```

```
# Verifica configurazione
$ git config --list
user.name=Mario Rossi
user.email=mario.rossi@example.com
```

1.6.2 Livelli di configurazione

Git ha tre livelli di configurazione:

`-system` Configurazione globale per tutti gli utenti del sistema
File: `/etc/gitconfig`

`-global` Configurazione per l'utente corrente
File: `~/.gitconfig` o `~/.config/git/config`

`-local` Configurazione per il repository specifico (default)
File: `.git/config` nel repository

Le configurazioni local sovrascrivono global, che sovrascrive system.

1.6.3 Configurazioni utili

Listing 1.5: Altre configurazioni consigliate

```
# Editor di default per commit message (esempio: vim, nano, code)
$ git config --global core.editor "code --wait"

# Colori output Git
$ git config --global color.ui auto

# Default branch name (main invece di master)
$ git config --global init.defaultBranch main

# Autocorrect comandi (attende 1.5 secondi prima di eseguire)
$ git config --global help.autocorrect 15

# Alias utili
$ git config --global alias.st status
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.cm commit
$ git config --global alias.lg "log --oneline --graph --all"

# Dopo configurazione alias:
$ git st  # Equivalente a: git status
$ git lg  # Log grafico compatto
```

1.6.4 Visualizzare e modificare configurazione

Listing 1.6: Gestione configurazione

```
# Visualizza tutte le configurazioni
$ git config --list

# Visualizza configurazione specifica
$ git config user.name
```

```
Mario Rossi

# Visualizza origine configurazione
$ git config --list --show-origin
file:/home/mario/.gitconfig user.name=Mario Rossi
file:/home/mario/.gitconfig user.email=mario.rossi@example.com

# Rimuovi configurazione
$ git config --global --unset user.email

# Modifica direttamente file configurazione
$ git config --global --edit
```

Riepilogo concetti chiave

Concetti fondamentali

- **Version Control** traccia modifiche ai file nel tempo
- **Git** è un DVCS (Distributed Version Control System)
- Ogni clone Git contiene la **storia completa** del progetto
- Git usa **snapshot**, non differenze tra file
- I tre stati: **Working Directory**, **Staging Area**, **Repository**
- **SHA-1 checksum** garantisce integrità dei dati
- DVCS è superiore a CVCS per velocità, offline work, branching
- Configurare `user.name` e `user.email` è **obbligatorio**

Esercizi

1. Spiega con un esempio pratico perché il version control è necessario in un progetto software.
2. Descrivi le differenze tra VCS centralizzati (SVN) e distribuiti (Git). Quali sono i vantaggi di Git?
3. Disegna un diagramma che mostra i tre stati di Git (Working Directory, Staging Area, Repository) e i comandi per passare da uno all'altro.
4. Installa Git sul tuo computer e verifica la versione installata con `git -version`.
5. Configura Git con il tuo nome ed email usando `git config -global`.
6. Crea almeno 3 alias Git utili (esempio: `st` per status, `lg` per log grafico).
7. Ricerca la storia di Git: perché Linus Torvalds ha creato Git nel 2005? Quali erano i requisiti che doveva soddisfare?
8. Confronta Git con un altro VCS (Mercurial o SVN). Quali sono le differenze architetturali?
9. Spiega cosa significa che Git usa "snapshot" invece di "differenze". Qual è il vantaggio?
10. Visualizza tutte le tue configurazioni Git con `git config -list -show-origin` e identifica dove sono salvate.

Verifica

Domande Vero/Falso:

1. Git è un sistema di version control centralizzato. (V/F)
2. Ogni clone Git contiene la storia completa del repository. (V/F)
3. La staging area è opzionale in Git. (V/F)
4. SHA-1 garantisce l'integrità dei dati in Git. (V/F)
5. Git richiede connessione internet per fare commit. (V/F)
6. Linus Torvalds ha creato Git nel 2005. (V/F)

Domande a Risposta Multipla:

1. Quale comando configura il nome utente globalmente?
   - `git config user.name "Nome"`
   - `git config -global user.name "Nome"`
   - `git set -global user.name "Nome"`
   - `git user.name "Nome"`
2. Cosa memorizza Git ad ogni commit?
   - Solo le differenze tra file
   - Snapshot completo di tutti i file
   - Solo i nomi dei file modificati
   - Compressione dei file

Laboratorio pratico

Lab 1: Setup ambiente Git

1. Installa Git sul tuo sistema operativo
2. Verifica l'installazione: `git -version`
3. Configura nome ed email globalmente
4. Configura editor di default (es: VS Code, Vim, Nano)
5. Crea 5 alias personalizzati
6. Visualizza tutte le configurazioni e salva l'output in un file
7. Sperimenta con `git config -global -edit`

Riferimenti

- Pro Git Book: https://git-scm.com/book/it/v2
- Git Official Documentation: https://git-scm.com/doc
- Git History (Linus Torvalds): https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-
- Visualizing Git: https://git-school.github.io/visualizing-git/
- GitHub Learning Lab: https://lab.github.com

Capitolo 2

Repository, Staging Area e Commit

Introduzione

Questo capitolo copre le operazioni fondamentali di Git: inizializzare repository, gestire la staging area, creare commit significativi, visualizzare la storia e configurare file da ignorare. Questi sono i comandi che userai quotidianamente nel lavoro con Git.

Obiettivi di apprendimento

- Inizializzare repository Git (`git init`)
- Comprendere il ruolo della staging area
- Creare commit con messaggi descrittivi
- Visualizzare stato e storia del repository
- Configurare .gitignore per escludere file
- Navigare la storia dei commit con git log
- Comprendere le best practices per commit atomici

2.1 Inizializzare un Repository Git

2.1.1 Creare un nuovo repository

Per iniziare a tracciare un progetto con Git, devi inizializzare un repository:

Listing 2.1: Inizializzare repository Git

```
# Crea directory progetto
$ mkdir mio-progetto
$ cd mio-progetto

# Inizializza repository Git
$ git init
Initialized empty Git repository in /home/user/mio-progetto/.git/

# Verifica creazione directory .git
$ ls -la
drwxr-xr-x  .git/
```

Cosa succede:

- Git crea directory nascosta `.git/`
- Contiene database, configurazione, refs, hooks
- La directory `.git/` è il repository vero e proprio
- La directory di lavoro è ora tracciata da Git

Attenzione: Directory .git

Mai eliminare o modificare manualmente la directory `.git/`. Contiene tutta la storia del progetto. Eliminarla significa perdere tutto il version control (i file rimarranno, ma senza storia).

2.1.2 Struttura directory .git

Listing 2.2: Contenuto directory .git

```
$ tree .git -L 1
.git/
          HEAD                    # Puntatore al branch corrente
          config                  # Configurazione repository-specific
          description             # Descrizione repository (GitWeb)
          hooks/                  # Script hook personalizzati
          index                   # Staging area (file binario)
          info/                   # Exclude patterns globali
          objects/                # Database oggetti Git (blob, tree,
   commit)
          refs/                   # Puntatori a commit (branches, tags)
```

File/Directory chiave:

`HEAD` Puntatore simbolico al branch corrente (es: `ref: refs/heads/main`)

`index` Staging area (snapshot file pronti per commit)

`objects/` Database content-addressable con tutti gli oggetti

`refs/heads/` Puntatori ai commit di ogni branch

2.1.3 Clonare repository esistente

Alternativa a `git init`: clonare repository esistente.

Listing 2.3: Clonare repository remoto

```
# Clona da GitHub
$ git clone https://github.com/username/repo.git

# Clona con nome directory custom
$ git clone https://github.com/username/repo.git mio-nome

# Clona solo ultimo commit (shallow clone)
$ git clone --depth 1 https://github.com/username/repo.git
```

Clone crea:

- Copia completa del repository (storia inclusa)
- Configurazione automatica remote "origin"
- Checkout automatico branch di default

2.2 La Staging Area (Index)

2.2.1 Concetto di staging area

La **staging area** (o index) è una caratteristica distintiva di Git. È un'area intermedia tra working directory e repository.

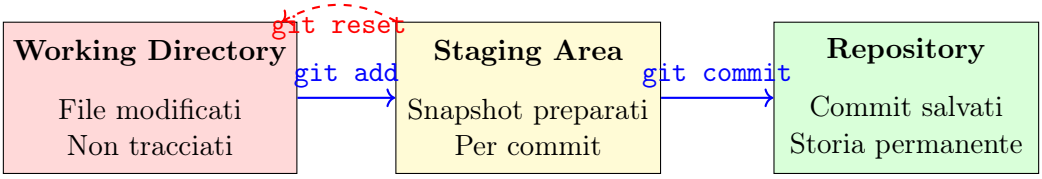


Figura 2.1: Flusso attraverso staging area

Perché staging area?

- **Controllo granulare**: Scegli esattamente cosa committare
- **Commit parziali**: Committi solo parte delle modifiche
- **Review pre-commit**: Verifica cosa stai per committare
- **Commit atomici**: Raggruppa modifiche logicamente correlate

Esempio: Staging Parziale

Hai modificato 5 file:

- 3 file implementano feature A
- 2 file implementano feature B

Con staging area puoi fare:

1. `git add` solo i 3 file di feature A
2. `git commit -m "Add feature A"`
3. `git add` i 2 file di feature B
4. `git commit -m "Add feature B"`

Risultato: Due commit separati e logici, non un commit confuso con entrambe le feature.

2.2.2 Stati dei file in Git

File in Git hanno quattro stati possibili:

Untracked File nuovo, Git non lo sta tracciando

Unmodified File tracciato, non modificato dall'ultimo commit

Modified File tracciato e modificato, non in staging

Staged File in staging area, pronto per commit

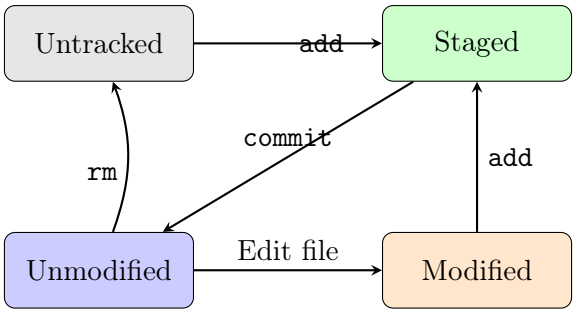


Figura 2.2: Ciclo di vita degli stati dei file

2.3 Visualizzare lo Stato: git status

2.3.1 Comando git status

`git status` è il comando più usato: mostra stato di working directory e staging area.

Listing 2.4: Esempio git status

```
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md
        main.py

nothing added to commit but untracked files present (use "git add" to
    track)
```

Output spiega:

- Branch corrente: `main`
- File untracked: `README.md`, `main.py`
- Suggerimento: usa `git add` per tracciare

2.3.2 git status dopo modifiche

Listing 2.5: Status con file modificati

```
$ echo "Hello World" > README.md
$ git add README.md
$ echo "print('Hello')" > main.py

$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md

```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        main.py
```

Sezioni output:

- **Changes to be committed**: File in staging (README.md)
- **Untracked files**: File non tracciati (main.py)

2.3.3 Status compatto

Listing 2.6: git status in formato breve

```
$ git status -s
A  README.md    # Staged (Added)
?? main.py      # Untracked

$ git status --short --branch
## main [ahead 2]
A  README.md
M  config.py    # Modified
?? main.py
```

Codici status:

- `??` - Untracked
- `A` - Added (staged)
- `M` - Modified
- `D` - Deleted
- `R` - Renamed
- `MM` - Modified sia in staging che in working directory

2.4 Aggiungere File: git add

2.4.1 Staging di file

Listing 2.7: git add - aggiungere file

```
# Aggiungi file singolo
$ git add README.md

# Aggiungi più file
$ git add main.py utils.py config.py

# Aggiungi tutti i file con estensione
$ git add *.py

# Aggiungi tutti i file nella directory
$ git add src/

# Aggiungi tutti i file modificati e nuovi
$ git add .

```

```
# Aggiungi tutti (inclusi file eliminati)
$ git add -A
# Equivalente a:
$ git add --all
```

2.4.2 Staging interattivo

Listing 2.8: git add interattivo

```
# Modalità interattiva (permette scelta selettiva)
$ git add -i

# Staging patch-by-patch (scegli singoli chunk)
$ git add -p

# Esempio output git add -p:
diff --git a/main.py b/main.py
@@ -1,3 +1,4 @@
 def hello():
     print("Hello")
+    print("World")

Stage this hunk [y,n,q,a,d,s,e,?]?
# y = yes, n = no, s = split, e = edit, ? = help
```

Best Practice: git add -p

`git add -p` è estremamente utile quando hai fatto molte modifiche ma vuoi committare solo parte di esse. Ti permette di creare commit atomici anche quando hai lavorato su più cose contemporaneamente.

2.4.3 Unstaging: rimuovere da staging

Listing 2.9: Rimuovere file da staging area

```
# Rimuovi file da staging (Git 2.23+)
$ git restore --staged README.md

# Metodo old-school (funziona sempre)
$ git reset HEAD README.md

# Rimuovi tutti i file da staging
$ git reset HEAD .
```

2.5 Creare Commit: git commit

2.5.1 Commit base

Un **commit** è uno snapshot permanente dei file in staging area.

Listing 2.10: Creare commit

```
# Commit con messaggio inline
$ git commit -m "Add README with project description"
[main (root-commit) a3f5b21] Add README with project description
```

```
 1 file changed, 3 insertions(+)
 create mode 100644 README.md

# Commit apre editor per messaggio multi-line
$ git commit
# Editor si apre, scrivi messaggio, salva e chiudi
```

Output commit spiega:

- Branch: `main`
- Hash commit: `a3f5b21` (abbreviato da 40 caratteri)
- File cambiati: 1
- Inserimenti: 3 righe

2.5.2 Commit con add automatico

Listing 2.11: Commit skip staging (solo file tracciati)

```
# Aggiungi e committi in un comando (solo modified files)
$ git commit -a -m "Update all tracked files"

# Equivalente a:
$ git add -u   # Aggiungi solo file già tracciati modificati
$ git commit -m "Update all tracked files"
```

Attenzione: git commit -a

`git commit -a` aggiunge automaticamente solo file **già tracciati**. File nuovi (untracked) sono ignorati. È comodo ma può causare commit accidentali. Meglio usare `git add` esplicitamente.

2.5.3 Anatomia di un buon commit message

Listing 2.12: Formato commit message

```
# Formato standard:
Short summary (50 chars max)

Detailed explanation if needed. Wrap at 72 characters.
Explain WHAT changed and WHY, not HOW (code shows how).

- Bullet points are ok
- Use imperative mood: "Add feature" not "Added feature"

Fixes #123
```

Best practices commit message:

1. **Prima riga**: Summary breve (50 caratteri max)
2. **Riga vuota**: Separa summary da body
3. **Body**: Spiegazione dettagliata (72 caratteri per riga)
4. **Imperative mood**: "Add", "Fix", "Update" (non "Added", "Fixed")

5. **Perché, non come**: Spiega motivazione, non implementazione
6. **References**: Link issue/ticket se presenti

Esempio Commit Message

Bad:

```
Fixed bug
```

Good:

```
Fix null pointer exception in user login

The login function didn't check if username was null
before accessing length property. This caused crashes
when users submitted empty form.

Now we validate input before processing.

Fixes #456
```

2.5.4 Commit atomici

Commit atomico: Contiene modifiche logicamente correlate a un singolo scopo.

Vantaggi:

- Storia leggibile e comprensibile
- Facile fare revert di singole feature
- Code review più semplice
- Bisect più efficace per trovare bug

Best Practice: Commit Atomici

Do:

- Un commit = una feature/fix
- Commit compila e passa test
- Message descrive il commit completamente

Don't:

- Commit con "Fix A and B and refactor C"
- Commit che rompe la build
- Message generici tipo "Updates" o "WIP"

2.5.5 Ammendare commit

Listing 2.13: Modificare ultimo commit

```
1 # Hai fatto commit ma dimenticato un file
```

```
$ git add forgotten_file.py
$ git commit --amend

# Amend senza cambiare messaggio
$ git commit --amend --no-edit

# Amend cambiando solo messaggio
$ git commit --amend -m "Better commit message"
```

Cosa fa `-amend`:

- Sostituisce l'ultimo commit con nuovo commit
- Combina staging corrente con commit precedente
- Cambia hash commit (tecnicamente nuovo commit)

Attenzione: Amend dopo Push

Mai fare amend di commit già pushati su branch condivisi! Amend riscrive la storia. Se altri hanno già pullato quel commit, creerai conflitti. Regola: amend solo commit locali non pushati.

2.6 Visualizzare Storia: git log

2.6.1 Log base

Listing 2.14: git log - storia commit

```
$ git log
commit a3f5b21e8c9d34f7a2b19c5e6d8f1a4b3c7e9f2d
Author: Mario Rossi <mario@example.com>
Date:   Mon Nov 11 10:30:45 2024 +0100

    Add user authentication feature

commit b8c9d34f7a2b19c5e6d8f1a4b3c7e9f2d3e5a1c
Author: Mario Rossi <mario@example.com>
Date:   Mon Nov 11 09:15:22 2024 +0100

    Initial commit
```

2.6.2 Log formattato

Listing 2.15: Formati git log utili

```
# Log compatto (una riga per commit)
$ git log --oneline
a3f5b21 Add user authentication feature
b8c9d34 Initial commit

# Log con grafo branch
$ git log --oneline --graph --all
* a3f5b21 (HEAD -> main) Add user authentication
* b8c9d34 Initial commit

# Log dettagliato con file modificati
```

```
$ git log --stat
commit a3f5b21...
    auth.py    | 45 ++++++++++++++++++
    login.html | 23 +++++++++
    2 files changed, 68 insertions(+)

# Log con diff completo
$ git log -p
commit a3f5b21...
diff --git a/auth.py b/auth.py
+def login(username, password):
+    # Implementation

# Ultimi N commit
$ git log -n 5      # Ultimi 5 commit
$ git log -3        # Ultimi 3 commit
```

2.6.3 Filtrare log

Listing 2.16: Filtrare storia commit

```
# Commit di autore specifico
$ git log --author="Mario"

# Commit in range di date
$ git log --since="2024-11-01"
$ git log --since="2 weeks ago"
$ git log --until="2024-11-30"

# Commit che toccano file specifico
$ git log -- README.md
$ git log --follow -- README.md  # Segui rename

# Commit con messaggio contenente keyword
$ git log --grep="bug"
$ git log --grep="feature" --grep="fix"  # OR
$ git log --grep="feature" --all-match   # AND

# Commit che modificano codice specifico
$ git log -S "function_name"  # Pickaxe: cerca nei diff
```

2.6.4 Log personalizzato

Listing 2.17: Custom format git log

```
# Pretty format personalizzato
$ git log --pretty=format:"%h - %an, %ar : %s"
a3f5b21 - Mario Rossi, 2 hours ago : Add user auth
b8c9d34 - Mario Rossi, 1 day ago : Initial commit

# Formato custom avanzato
$ git log --pretty=format:"%C(yellow)%h%Creset %C(blue)%an%Creset: %s"

# Salva come alias
$ git config --global alias.lg "log --graph --pretty=format:'%Cred%h%
   Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset
   ' --abbrev-commit"
```

```
$ git lg
* a3f5b21 - (HEAD -> main) Add user auth (2 hours ago) <Mario Rossi>
* b8c9d34 - Initial commit (1 day ago) <Mario Rossi>
```

Format placeholders utili:

- `%h` - Hash abbreviato
- `%H` - Hash completo
- `%an` - Author name
- `%ae` - Author email
- `%ad` - Author date
- `%ar` - Author date, relative (2 hours ago)
- `%s` - Subject (prima riga messaggio)
- `%b` - Body (resto messaggio)
- `%d` - Ref names (branch, tag)

2.7 File .gitignore

2.7.1 Perché .gitignore

Alcuni file non devono essere tracciati da Git:

- File temporanei (`*.tmp`, `*.swp`)
- Build artifacts (`*.o`, `*.pyc`, `dist/`)
- Dipendenze (`node_modules/`, `venv/`)
- File IDE (`.vscode/`, `.idea/`)
- Credenziali (`.env`, `secrets.json`)
- File sistema operativo (`.DS_Store`, `Thumbs.db`)

2.7.2 Sintassi .gitignore

Listing 2.18: Esempio .gitignore

```
# Commento

# File specifico
secrets.json

# Pattern wildcard
*.log
*.tmp

# Intere directory
node_modules/
build/
dist/
```

```
14
15 # Negazione (! = NON ignorare)
16 *.log
17 !important.log    # Traccia questo .log specifico
18
19 # Ignorare file solo in root
20 /TODO             # Solo /TODO, non src/TODO
21
22 # Ignorare file in qualsiasi subdirectory
23 **/temp           # Ignora temp ovunque
24
25 # Pattern avanzati
26 **/*.pyc          # .pyc in qualsiasi directory
27 **/dist/*         # Contenuto di qualsiasi dist/
```

2.7.3 .gitignore per linguaggi comuni

Listing 2.19: .gitignore Python

```
1  # Python
2  __pycache__/
3  *.py[cod]
4  *$py.class
5  *.so
6  .Python
7  venv/
8  env/
9  ENV/
10 .pytest_cache/
11 *.egg-info/
12 dist/
13 build/
```

Listing 2.20: .gitignore Node.js

```
1 # Node.js
2 node_modules/
3 npm-debug.log
4 yarn-error.log
5 .env
6 dist/
7 build/
8 .cache/
9 *.tgz
```

Listing 2.21: .gitignore Java

```
1 # Java
2 *.class
3 *.jar
4 *.war
5 *.ear
6 target/
7 .mvn/
8 .gradle/
9 build/
```

2.7.4 Gitignore globale

Listing 2.22: Configurare gitignore globale

```
# Crea file gitignore globale
$ touch ~/.gitignore_global

# Configura Git per usarlo
$ git config --global core.excludesfile ~/.gitignore_global

# Aggiungi pattern OS/IDE comuni
# ~/.gitignore_global
.DS_Store         # macOS
Thumbs.db         # Windows
.vscode/          # VS Code
.idea/            # IntelliJ
*.swp             # Vim
*~                # Backup files
```

2.7.5 Ignorare file già tracciati

Listing 2.23: Rimuovere file tracciato da Git

```
# Se hai committato file per errore (es: .env)
$ echo ".env" >> .gitignore

# Rimuovi da Git ma mantieni file locale
$ git rm --cached .env

$ git commit -m "Remove .env from tracking"
```

Errore Comune: .gitignore non funziona

Se aggiungi pattern a .gitignore ma il file è già tracciato, Git continuerà a tracciarlo. Devi prima fare `git rm -cached file` per rimuoverlo dall'index.

2.8 Visualizzare Differenze: git diff

2.8.1 Diff working directory

Listing 2.24: git diff - vedere modifiche

```
# Differenze non staged (working vs staging)
$ git diff
diff --git a/main.py b/main.py
index a3f5b21..b8c9d34 100644
--- a/main.py
+++ b/main.py
@@ -1,3 +1,4 @@
 def hello():
     print("Hello")
+    print("World")

# Differenze staged (staging vs ultimo commit)
$ git diff --staged
$ git diff --cached  # Equivalente
```

```
15
16 # Differenze completo (working vs ultimo commit)
17 $ git diff HEAD
```

2.8.2 Diff tra commit

Listing 2.25: Diff tra commit specifici

```
1 # Diff tra due commit
2 $ git diff a3f5b21 b8c9d34
3
4 # Diff ultimo commit vs precedente
5 $ git diff HEAD HEAD~1
6
7 # Diff su file specifico
8 $ git diff main.py
9 $ git diff a3f5b21 b8c9d34 -- main.py
```

Riepilogo concetti chiave

Concetti fondamentali

- `git init` inizializza repository nella directory corrente
- **Staging area** permette controllo granulare su cosa committare
- `git status` mostra stato working directory e staging
- `git add` aggiunge file alla staging area
- `git commit` crea snapshot permanente dei file in staging
- **Commit atomici**: Un commit = una modifica logica
- **Commit message**: Summary breve + body dettagliato
- `git log` visualizza storia commit
- `.gitignore` specifica file da non tracciare
- `git diff` mostra differenze tra versioni

Esercizi

1. Crea nuovo repository, aggiungi 3 file e fai primo commit.
2. Modifica 2 file, usa `git add -p` per fare commit separati.
3. Scrivi .gitignore per progetto Python che ignora `venv/`, `*.pyc`, `.env`.
4. Usa `git log -oneline -graph` per visualizzare storia.
5. Fai commit con messaggio multi-line usando editor.
6. Usa `git commit -amend` per correggere commit message.

7. Configura alias `git lg` per log grafico colorato.
8. Crea file .gitignore_global per OS/IDE.
9. Usa `git diff` per vedere modifiche prima di committare.
10. Fai commit di file per errore, usa `git rm -cached` per rimuoverlo.

Riferimenti

- Git Basics: https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository
- Gitignore templates: https://github.com/github/gitignore
- Commit message guide: https://chris.beams.io/posts/git-commit/

Capitolo 3

Branching e Merging

Introduzione

Il branching è una delle feature killer di Git. Permette di divergere dalla linea principale di sviluppo per lavorare su feature, esperimenti o fix senza impattare il codice stabile. Questo capitolo copre creazione branch, switching, merging, rebase e risoluzione conflitti.

Obiettivi di apprendimento

- Comprendere cosa sono i branch e perché sono fondamentali
- Creare e gestire branch (`git branch`)
- Switching tra branch (`git checkout`/`git switch`)
- Merge: fast-forward e three-way merge
- Rebase per linearizzare la storia
- Risolvere conflitti di merge
- Eliminare branch obsoleti
- Best practices per branching strategy

3.1 Cosa Sono i Branch

3.1.1 Definizione

Un **branch** (ramo) in Git è semplicemente un puntatore mobile a un commit. Il branch di default è chiamato `main` (o `master` in repo più vecchi).

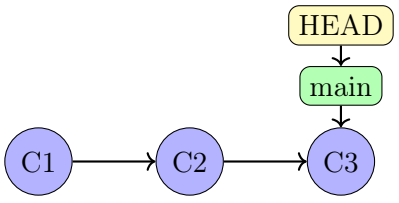


Figura 3.1: Branch come puntatore a commit

Concetti chiave:

- **Branch**: Puntatore a un commit specifico
- **HEAD**: Puntatore al branch corrente (dove sei ora)
- Creare branch è **istantaneo** (solo creare puntatore)
- Nessun overhead di copia file

3.1.2 Perché usare i branch

Vantaggi del Branching

- **Isolamento**: Lavora su feature senza toccare codice stabile
- **Sperimentazione**: Prova idee senza rischio
- **Collaborazione**: Team lavora su feature diverse in parallelo
- **Code review**: Feature branch + pull request
- **Hotfix**: Fixa bug in produzione senza feature incomplete
- **Release management**: Branch per versioni diverse

Scenario Tipico

Stai sviluppando feature complessa (richiederà 2 settimane). Nel frattempo:

1. Bug critico scoperto in produzione
2. Altro dev aggiunge piccola feature
3. Testing richiede modifiche

Senza branch: chaos totale, conflitti continui, impossibile deployare.
Con branch:

- `main`: codice stabile in produzione
- `feature/nuova-ui`: tua feature (isolata)
- `hotfix/critical-bug`: fix urgente
- `feature/small-feature`: altra feature

Ogni branch evolve indipendentemente, merge quando pronto!

3.2 Creare e Gestire Branch

3.2.1 Visualizzare branch esistenti

Listing 3.1: Listare branch

```
# Lista branch locali
$ git branch
* main
  feature/login
  hotfix/navbar

```

```
# Branch con ultimo commit
$ git branch -v
* main            a3f5b21 Add user authentication
  feature/login   b8c9d34 WIP: login form
  hotfix/navbar   c7e2a19 Fix navbar on mobile

# Lista branch locali e remoti
$ git branch -a
* main
  feature/login
  remotes/origin/main
  remotes/origin/develop

# Branch merged/non-merged
$ git branch --merged         # Branch già mergiati in corrente
$ git branch --no-merged      # Branch non ancora mergiati
```

3.2.2 Creare branch

Listing 3.2: Creare nuovo branch

```
# Crea branch (non switcha)
$ git branch feature/nuova-feature

# Verifica creazione
$ git branch
* main
  feature/nuova-feature

# Crea e switcha in un comando (metodo classico)
$ git checkout -b feature/altra-feature
Switched to a new branch 'feature/altra-feature'

# Crea e switcha (Git 2.23+, raccomandato)
$ git switch -c feature/moderna-feature
Switched to a new branch 'feature/moderna-feature'

# Crea branch da commit specifico
$ git branch hotfix/bug a3f5b21

# Crea branch da tag
$ git branch release/v2.0 v2.0
```

3.2.3 Switching tra branch

Listing 3.3: Cambiare branch

```
# Metodo classico
$ git checkout feature/nuova-feature
Switched to branch 'feature/nuova-feature'

# Metodo moderno (Git 2.23+)
$ git switch feature/nuova-feature
Switched to branch 'feature/nuova-feature'

# Torna al branch precedente
```

```
$ git switch -

# Switcha e crea se non esiste
$ git switch -c feature/auto-create
```

Nota: checkout vs switch

Git 2.23 (2019) ha introdotto `git switch` per rendere più chiara la separazione:

- `git switch`: Cambiare branch
- `git restore`: Ripristinare file
- `git checkout`: Fa entrambi (confusionario)

Raccomandazione: usa `switch` e `restore` nei nuovi progetti.

3.2.4 Rinominare branch

Listing 3.4: Rinominare branch

```
# Rinomina branch corrente
$ git branch -m nuovo-nome

# Rinomina branch specifico
$ git branch -m vecchio-nome nuovo-nome

# Esempio: rinomina master in main
$ git branch -m master main
```

3.2.5 Eliminare branch

Listing 3.5: Eliminare branch

```
# Elimina branch (safe: solo se merged)
$ git branch -d feature/completata
Deleted branch feature/completata (was a3f5b21).

# Force delete (anche se non merged)
$ git branch -D feature/abbandonata
Deleted branch feature/abbandonata (was b8c9d34).

# Elimina branch remoto
$ git push origin --delete feature/vecchia
```

Attenzione: Branch Deletion

`-d` è safe: Git impedisce eliminazione se branch non merged (protezione da perdita lavoro). `-D` è force: elimina anche branch non merged. Usa `-D` solo se sei sicuro di voler perdere quel lavoro.

3.3 Merging: Unire Branch

3.3.1 Concetto di merge

Merge integra modifiche da un branch in un altro. Tipicamente: merge feature branch in `main`.

3.3.2 Fast-Forward Merge

Quando non ci sono commit divergenti, Git fa **fast-forward**: sposta semplicemente puntatore avanti.

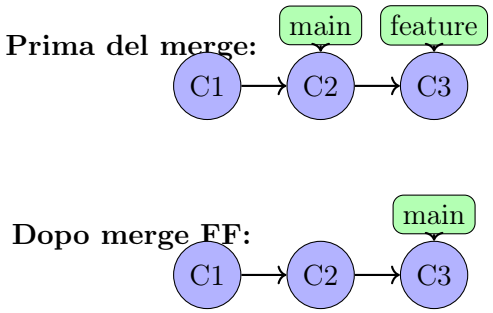


Figura 3.2: Fast-Forward Merge

Listing 3.6: Esempio Fast-Forward Merge

```
# Situazione iniziale
$ git log --oneline --graph --all
* c3 (feature) Add feature C
* c2 Add feature B
* c1 (HEAD -> main) Initial commit

# Merge feature in main
$ git switch main
$ git merge feature
Updating c1..c3
Fast-forward
 feature.py | 10 ++++++++++
 1 file changed, 10 insertions(+)

# Storia dopo merge
$ git log --oneline --graph
* c3 (HEAD -> main, feature) Add feature C
* c2 Add feature B
* c1 Initial commit
```

Caratteristiche FF merge:

- Nessun commit di merge creato
- Storia lineare e pulita
- Possibile solo se nessun commit divergente

3.3.3 Three-Way Merge

Quando i branch hanno commit divergenti, Git crea **merge commit** con due parent.

Listing 3.7: Esempio Three-Way Merge

```
# Situazione: main e feature divergenti
$ git log --oneline --graph --all
* c3 (feature) Add feature C
| * c2 (HEAD -> main) Fix bug in main
|/
* c1 Initial commit
```

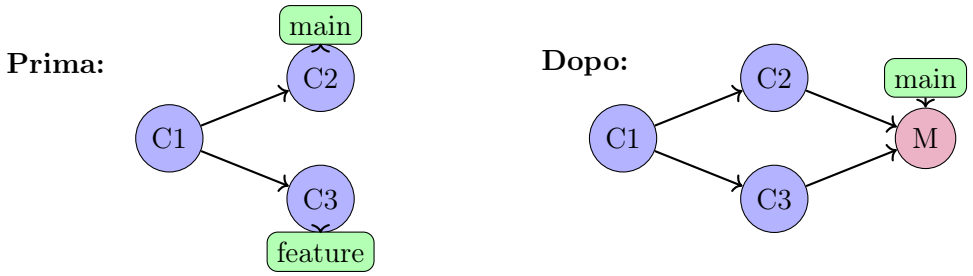


Figura 3.3: Three-Way Merge (M = merge commit)

```
7
8  # Merge feature in main
9  $ git switch main
10 $ git merge feature
11 Merge made by the 'recursive' strategy.
12  feature.py | 10 ++++++++++
13  1 file changed, 10 insertions(+)
14
15 # Storia dopo merge
16 $ git log --oneline --graph
17 *   m1 (HEAD -> main) Merge branch 'feature'
18 |\
19 | * c3 (feature) Add feature C
20 * | c2 Fix bug in main
21 |/
22 * c1 Initial commit
```

Caratteristiche three-way merge:

- Crea merge commit con due parent
- Messaggio auto-generato (modificabile)
- Preserva storia non lineare
- Mostra chiaramente quando feature fu integrata

3.3.4 Forzare merge commit (no fast-forward)

Listing 3.8: Merge con –no-ff

```
1  # Forza creazione merge commit anche se FF possibile
2  $ git merge --no-ff feature/login
3
4  # Utile per preservare informazione "questa era una feature"
5  $ git log --oneline --graph
6  *   m1 (HEAD -> main) Merge branch 'feature/login'
7  |\
8  | * c3 (feature/login) Complete login form
9  | * c2 Add login validation
10 |/
11 * c1 Initial commit
```

Best Practice: –no-ff per Feature

In workflow professionali (Git Flow), è comune usare `-no-ff` per feature branch. Mantiene traccia chiara di quando feature fu aggiunta. Storia mostra:

- Quali commit fanno parte della feature
- Quando fu integrata
- Facile revert intera feature (revert del merge commit)

3.3.5 Merge squash

Listing 3.9: Merge con squash

```
# Merge tutti commit feature in singolo commit
$ git merge --squash feature/experiment
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested

$ git commit -m "Add experimental feature"

# Storia risultante: un solo commit, non merge commit
$ git log --oneline
* m1 (HEAD -> main) Add experimental feature
* c1 Initial commit
```

Quando usare squash:

- Feature con molti commit WIP disordinati
- Vuoi storia main pulita e lineare
- Commit intermedi non aggiungono valore

Svantaggio: Perdi storia dettagliata feature development.

3.4 Conflitti di Merge

3.4.1 Quando avvengono conflitti

Conflitto avviene quando:

- Due branch modificano stessa riga di stesso file
- Un branch modifica file, altro lo elimina
- Rename complessi

3.4.2 Esempio conflitto

Listing 3.10: Merge con conflitto

```
$ git merge feature/conflict
Auto-merging main.py
CONFLICT (content): Merge conflict in main.py
Automatic merge failed; fix conflicts and then commit the result.

```

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   main.py
```

3.4.3 Formato marker conflitto

Listing 3.11: File con conflitto

```
def greet():
<<<<<<< HEAD
    print("Hello from main!")
=======
    print("Hello from feature!")
>>>>>>> feature/conflict
    return True
```

Marker conflitto:

- `<<<<< HEAD`: Inizio versione branch corrente
- `=======`: Separatore
- `>>>>> feature`: Fine versione branch da mergere

3.4.4 Risolvere conflitti

Processo risoluzione:

1. **Identifica file**: `git status` mostra "both modified"
2. **Apri file**: Cerca marker `<<<<<`
3. **Decidi versione**: Mantieni una, entrambe, o scrivi nuova
4. **Rimuovi marker**: Elimina `<<<<<`, `=======`, `>>>>>`
5. **Stage file**: `git add main.py`
6. **Completa merge**: `git commit`

Listing 3.12: Risoluzione conflitto manuale

```
# 1. Vedi conflitti
$ git status
Unmerged paths:
        both modified:   main.py

# 2. Modifica file (scegli versione)
# Versione risolta:
def greet():
    print("Hello from both branches!")
    return True

```

```
# 3. Stage file risolto
$ git add main.py

# 4. Verifica risoluzione completa
$ git status
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

# 5. Completa merge
$ git commit
# Editor apre con messaggio auto-generato
[main a4b6c8d] Merge branch 'feature/conflict'
```

3.4.5 Tool per risoluzione conflitti

Listing 3.13: Usare merge tool

```
# Configura merge tool (es: vimdiff, meld, kdiff3)
$ git config --global merge.tool vimdiff

# Lancia merge tool su conflitti
$ git mergetool

# Tool mostra 3 panel:
# - LOCAL (tua versione)
# - REMOTE (versione da mergere)
# - BASE (ancestor comune)
# - MERGED (risultato)
```

3.4.6 Abortire merge

Listing 3.14: Annullare merge in corso

```
# Se merge non riesce e vuoi annullare tutto
$ git merge --abort

# Torna allo stato pre-merge
$ git status
On branch main
nothing to commit, working tree clean
```

3.5 Rebase: Storia Lineare

3.5.1 Cos'è rebase

Rebase riscrive la storia spostando commit su nuova base. Alternativa a merge per integrare modifiche.

3.5.2 Rebase base

Listing 3.15: Rebase feature su main

```
# Situazione iniziale
$ git log --oneline --graph --all
* c4 (feature) Add feature part 2
```

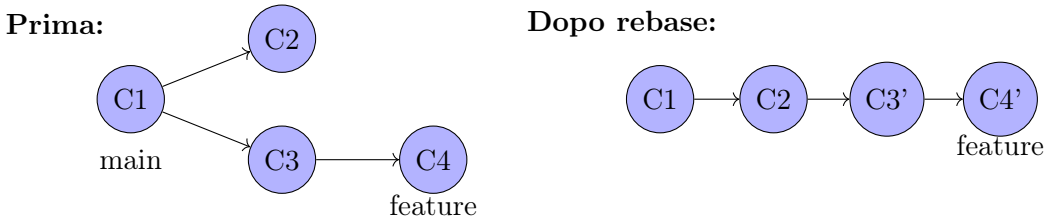


Figura 3.4: Rebase: C3' e C4' sono nuovi commit (hash diversi)

```
* c3 Add feature part 1
| * c2 (main) Fix bug
|/
* c1 Initial commit

# Switch a feature e rebase su main
$ git switch feature
$ git rebase main
First, rewinding head to replay your work on top of it...
Applying: Add feature part 1
Applying: Add feature part 2

# Storia dopo rebase (lineare!)
$ git log --oneline --graph
* c4' (HEAD -> feature) Add feature part 2
* c3' Add feature part 1
* c2 (main) Fix bug
* c1 Initial commit
```

3.5.3 Rebase vs Merge

Merge vs Rebase

Merge:

+ Preserva storia vera del progetto
+ Safe: non riscrive commit esistenti
+ Mostra quando branch fu integrato
- Storia non lineare (più complessa)
- Molti merge commit possono inquinare storia

Rebase:

+ Storia lineare e pulita
+ Facile da leggere e navigare
+ Nessun merge commit extra
- Riscrive storia (cambia hash commit)
- Può causare problemi su branch condivisi

3.5.4 Golden Rule of Rebase

REGOLA D'ORO: Mai Rebase Branch Pubblici

Mai fare rebase di commit già pushati su branch condivisi!
Rebase riscrive storia (cambia hash). Se altri hanno già pullato quei commit, creerai caos:

- Altri avranno commit duplicati
- Merge conflicts continui
- Storia corrotta e confusa

Safe: Rebase solo branch locali privati prima di pushare.
Unsafe: Rebase di `main` o branch condivisi.

3.5.5 Rebase interattivo

Listing 3.16: Rebase interattivo per pulire storia

```
# Rebase interattivo ultimi 3 commit
$ git rebase -i HEAD~3

# Editor apre con lista commit:
pick c3 Add feature part 1
pick c4 Fix typo
pick c5 Add feature part 2

# Comandi disponibili:
# pick = usa commit
# reword = usa commit ma modifica messaggio
# edit = usa commit ma fermati per amend
# squash = unisci con commit precedente
# fixup = come squash ma scarta messaggio
# drop = elimina commit

# Esempio: squash fix typo in commit precedente
pick c3 Add feature part 1
fixup c4 Fix typo
pick c5 Add feature part 2

# Salva e chiudi: Git applica modifiche
```

Use case rebase interattivo:

- Pulizia storia prima di PR (squash WIP commits)
- Riordinare commit logicamente
- Correggere messaggi commit
- Eliminare commit di debug

3.5.6 Rebase con conflitti

Listing 3.17: Risolvere conflitti durante rebase

```
$ git rebase main
Applying: Add feature
```

```
CONFLICT (content): Merge conflict in main.py

# Risolvi conflitti manualmente
$ vim main.py  # Risolvi e rimuovi marker

# Stage file risolto
$ git add main.py

# Continua rebase
$ git rebase --continue

# Oppure abbandona rebase
$ git rebase --abort
```

Riepilogo concetti chiave

Concetti fondamentali

- **Branch** è un puntatore mobile a commit
- `git branch` crea/lista branch, `git switch` cambia branch
- **Merge** integra modifiche da branch: FF o three-way
- **Conflitti** avvengono quando stessa riga modificata in entrambi
- Risolvi conflitti manualmente, poi `git add` e `git commit`
- **Rebase** riscrive storia per renderla lineare
- `-no-ff` forza merge commit (utile per feature)
- `-squash` comprime branch in singolo commit
- **Golden Rule**: Mai rebase branch pubblici condivisi
- Rebase interattivo (`-i`) per pulizia storia

Esercizi

1. Crea branch `feature/test`, aggiungi 2 commit, mergia in `main`.
2. Crea due branch divergenti, fai three-way merge, visualizza con `git log -graph`.
3. Crea conflitto intenzionale (modifica stessa riga in due branch), risolvi manualmente.
4. Usa `git merge -no-ff` e confronta storia con merge FF.
5. Crea branch con 5 commit, usa rebase interattivo per squashare in 2 commit.
6. Sperimenta `git rebase` per linearizzare storia.
7. Simula scenario: branch `main` avanza, feature branch diventa outdated, rebase feature su main.
8. Usa `git mergetool` (configura prima tool preferito).

9. Crea alias per `git log -graph -oneline -all`.
10. Pratica workflow: crea feature branch, sviluppa, rebase su main aggiornato, merge.

Riferimenti

- Git Branching: https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
- Merge vs Rebase: https://www.atlassian.com/git/tutorials/merging-vs-rebasing
- Resolving Conflicts: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Mergin

Capitolo 4

Repository Remoti e Collaborazione

Introduzione

Git è un sistema distribuito: ogni clone è un repository completo. I repository remoti permettono collaborazione e backup centralizzato. Questo capitolo copre configurazione remote, sincronizzazione con clone/fetch/pull/push, e gestione di branch remoti.

Obiettivi di apprendimento

- Comprendere il concetto di repository remoto
- Configurare remote (`git remote`)
- Clonare repository esistenti (`git clone`)
- Sincronizzare modifiche (`fetch`, `pull`, `push`)
- Gestire tracking branches
- Risolvere conflitti durante pull/push
- Best practices per collaborazione
- Gestire fork e upstream

4.1 Concetto di Repository Remoto

4.1.1 Architettura distribuita

In Git, ogni sviluppatore ha:

- **Repository locale**: Copia completa con tutta la storia
- **Repository remoto**: Server centrale (GitHub, GitLab, BitBucket) per condivisione

Workflow tipico:

1. **Clone**: Scarica repository da remote
2. **Commit**: Lavora localmente, crea commit
3. **Pull**: Scarica aggiornamenti da remote
4. **Push**: Carica commit locali su remote

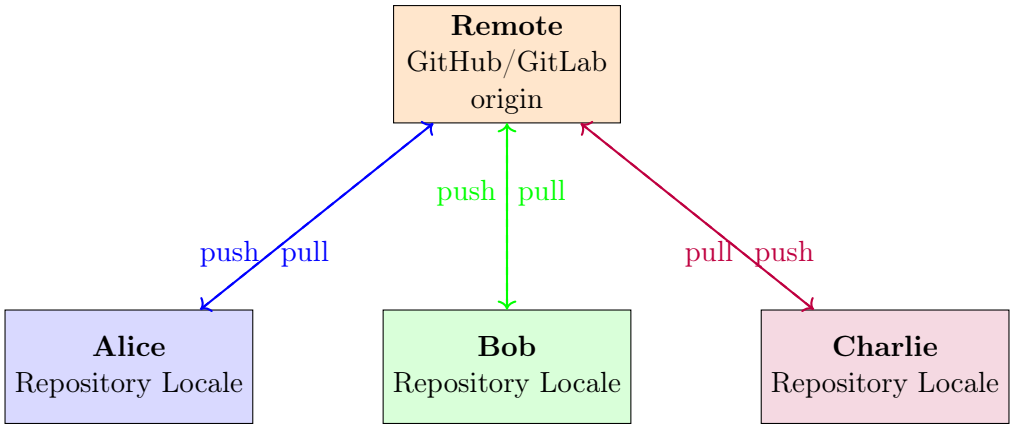


Figura 4.1: Architettura Git distribuita con remote centrale

4.1.2 Vantaggi repository remoti

Vantaggi Remote Repository

- **Backup**: Codice salvato su server (disaster recovery)
- **Collaborazione**: Team condivide codice facilmente
- **Code review**: Pull request per review pre-merge
- **CI/CD**: Automazione test/deploy su push
- **Visibilità**: Storia progetto accessibile a tutti
- **Issue tracking**: GitHub Issues, GitLab Boards
- **Documentation**: Wiki, README, GitHub Pages

4.2 Gestire Remote: git remote

4.2.1 Visualizzare remote configurati

Listing 4.1: Listare remote

```
# Lista remote (solo nomi)
$ git remote
origin

# Lista remote con URL
$ git remote -v
origin  https://github.com/user/repo.git (fetch)
origin  https://github.com/user/repo.git (push)

# Dettagli remote specifico
$ git remote show origin
* remote origin
  Fetch URL: https://github.com/user/repo.git
  Push  URL: https://github.com/user/repo.git
  HEAD branch: main
  Remote branches:
    main    tracked
```

```
    develop tracked
  Local branch configured for 'git pull':
    main merges with remote main
  Local ref configured for 'git push':
    main pushes to main (up to date)
```

4.2.2 Aggiungere remote

Listing 4.2: Aggiungere remote repository

```
# Aggiungi remote chiamato "origin"
$ git remote add origin https://github.com/user/repo.git

# Verifica aggiunta
$ git remote -v
origin  https://github.com/user/repo.git (fetch)
origin  https://github.com/user/repo.git (push)

# Aggiungi secondo remote (es: fork upstream)
$ git remote add upstream https://github.com/original/repo.git

$ git remote -v
origin    https://github.com/user/repo.git (fetch)
origin    https://github.com/user/repo.git (push)
upstream  https://github.com/original/repo.git (fetch)
upstream  https://github.com/original/repo.git (push)
```

Convenzioni naming:

- **origin**: Remote principale (tuo repository)
- **upstream**: Repository originale (in caso di fork)
- Nomi custom: staging, production, backup, etc.

4.2.3 Modificare e rimuovere remote

Listing 4.3: Gestione remote

```
# Rinomina remote
$ git remote rename origin upstream
$ git remote
upstream

# Cambia URL remote
$ git remote set-url origin https://github.com/newuser/repo.git

# Cambia URL da HTTPS a SSH
$ git remote set-url origin git@github.com:user/repo.git

# Rimuovi remote
$ git remote remove upstream
$ git remote
origin
```

4.2.4 HTTPS vs SSH

HTTPS:

```
https://github.com/user/repo.git
```

Pros: Setup semplice, funziona ovunque
Cons: Richiede username/password (o token) ogni push

SSH:

```
git@github.com:user/repo.git
```

Pros: Autenticazione automatica con SSH keys, più sicuro
Cons: Richiede setup chiavi SSH

Best Practice: Usa SSH

Per repository frequenti, configura SSH keys. Eviti di inserire credenziali ogni volta:

```
# Genera SSH key
$ ssh-keygen -t ed25519 -C "your_email@example.com"

# Aggiungi key a ssh-agent
$ ssh-add ~/.ssh/id_ed25519

# Copia public key e aggiungila su GitHub Settings > SSH Keys
$ cat ~/.ssh/id_ed25519.pub
```

4.3 Clonare Repository: git clone

4.3.1 Clone base

Listing 4.4: Clonare repository

```
# Clone da GitHub
$ git clone https://github.com/user/repo.git
Cloning into 'repo'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (3/3), done.

$ cd repo
$ git remote -v
origin  https://github.com/user/repo.git (fetch)
origin  https://github.com/user/repo.git (push)
```

Cosa fa `git clone`:

1. Crea directory con nome del repository
2. Inizializza `.git/`
3. Configura `origin` remote
4. Scarica tutta la storia
5. Checkout branch default (main/master)
6. Configura tracking per branch default

4.3.2 Opzioni clone

Listing 4.5: Clone con opzioni

```
# Clone con nome directory custom
$ git clone https://github.com/user/repo.git mio-progetto

# Shallow clone (solo ultimo commit, risparmia spazio)
$ git clone --depth 1 https://github.com/user/repo.git

# Clone singolo branch
$ git clone --branch develop --single-branch https://github.com/user/
   repo.git

# Clone bare (senza working directory, per server)
$ git clone --bare https://github.com/user/repo.git repo.git
```

Shallow clone use cases:

- CI/CD: Build solo ultimo stato
- Download veloce di repo enormi
- Demo/testing rapido

Limitazioni shallow clone:

- Non puoi push
- Git log limitato
- Alcuni comandi non funzionano

4.4 Fetch: Scaricare Aggiornamenti

4.4.1 Concetto di fetch

`git fetch` scarica commit da remote **senza** modificare working directory. Aggiorna solo remote-tracking branches.

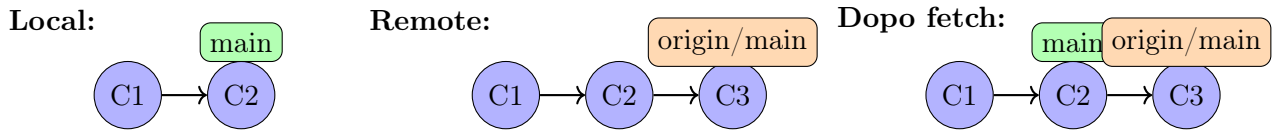


Figura 4.2: Fetch aggiorna origin/main, main rimane invariato

Listing 4.6: git fetch esempi

```
# Fetch da origin (default remote)
$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
Receiving objects: 100% (3/3), done.

# Fetch da remote specifico
$ git fetch upstream

```

```
# Fetch tutti i remote
$ git fetch --all

# Fetch e prune branch remoti eliminati
$ git fetch --prune
```

4.4.2 Visualizzare branch remoti

Listing 4.7: Branch remoti dopo fetch

```
# Lista branch remote
$ git branch -r
  origin/main
  origin/develop
  origin/feature/login

# Lista tutti (locali + remoti)
$ git branch -a
* main
  develop
  remotes/origin/main
  remotes/origin/develop
  remotes/origin/feature/login

# Log branch remoto
$ git log origin/main

# Confronta local vs remote
$ git log main..origin/main  # Commit in remote non in local
$ git log origin/main..main  # Commit in local non in remote
```

4.5 Pull: Scaricare e Merge

4.5.1 git pull = fetch + merge

`git pull` combina `fetch` e `merge`:

Listing 4.8: git pull

```
# Pull = fetch + merge
$ git pull
# Equivalente a:
$ git fetch origin
$ git merge origin/main

# Pull da remote/branch specifici
$ git pull origin develop

# Pull con rebase invece di merge
$ git pull --rebase
# Equivalente a:
$ git fetch origin
$ git rebase origin/main
```

Nota: Pull Default Behavior

`git pull` senza argomenti:

- Usa remote configurato per branch corrente (di solito `origin`)
- Mergia tracking branch corrispondente
- Se non configurato, richiede specificare remote/branch

4.5.2 Pull con conflitti

Listing 4.9: Risolvere conflitti durante pull

```
$ git pull
Auto-merging main.py
CONFLICT (content): Merge conflict in main.py
Automatic merge failed; fix conflicts and then commit the result.

# Risolvi conflitti manualmente
$ vim main.py

# Stage file risolto
$ git add main.py

# Completa merge
$ git commit
[main a4b6c8d] Merge branch 'main' of https://github.com/user/repo

# Oppure abbandona pull
$ git merge --abort
```

4.5.3 Pull rebase

Listing 4.10: Pull con rebase per storia lineare

```
# Pull con rebase
$ git pull --rebase

# Configura rebase come default per pull
$ git config --global pull.rebase true

# Storia lineare senza merge commit
$ git log --oneline --graph
* c3 (HEAD -> main, origin/main) Remote commit
* c2 Your local commit
* c1 Initial commit
```

Best Practice: Pull Rebase

`git pull -rebase` è preferibile per storia pulita:

- Evita merge commit inutili
- Storia lineare e leggibile
- Commit locali "sopra" commit remoti

Usa solo se:

- Commit locali non ancora pushati
- Storia locale non condivisa

4.6 Push: Caricare Commit

4.6.1 Push base

Listing 4.11: Pushare commit su remote

```
# Push branch corrente su origin
$ git push
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 280 bytes | 280.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/user/repo.git
   a3f5b21..b8c9d34  main -> main

# Push specificando remote e branch
$ git push origin main

# Push e imposta tracking
$ git push -u origin feature/nuova
# Equivalente a:
$ git push --set-upstream origin feature/nuova
```

Cosa fa `-u`/`-set-upstream`:

- Crea branch remoto se non esiste
- Configura tracking tra branch locale e remoto
- Successivi `git push` non richiedono argomenti

4.6.2 Push tutti i branch

Listing 4.12: Push multipli branch

```
# Push tutti i branch
$ git push --all

# Push anche tag
$ git push --tags

# Push tutto (branch + tag)
$ git push --all && git push --tags
```

4.6.3 Push rejected

Listing 4.13: Push rifiutato - remote ha commit nuovi

```
$ git push
To https://github.com/user/repo.git
```

```
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/user/repo.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository
   pushing
hint: to the same ref. You may want to first integrate the remote
   changes
hint: (e.g., 'git pull ...') before pushing again.
```

Soluzione:

Listing 4.14: Integrare modifiche remote prima di push

```
# Pull modifiche remote (fetch + merge)
$ git pull

# Risolvi eventuali conflitti

# Riprova push
$ git push
```

4.6.4 Force push

Listing 4.15: Force push - PERICOLOSO

```
# Force push (sovrascrive remote)
$ git push --force
# Equivalente (più safe):
$ git push --force-with-lease
```

ATTENZIONE: Force Push

`git push -force` sovrascrive storia remota. **Estremamente pericoloso** su branch condivisi:

Quando è OK:

- Branch privato personale
- Dopo rebase interattivo su feature branch
- Fix commit message su branch non condiviso

Quando NON usare:

- Mai su `main`/`master`
- Mai su branch condivisi con altri
- Mai se non sai cosa stai facendo

Alternativa più safe: `-force-with-lease` (fallisce se remote ha commit non previsti)

4.6.5 Eliminare branch remoto

Listing 4.16: Eliminare branch su remote

```
# Elimina branch remoto
$ git push origin --delete feature/vecchia
```

```
To https://github.com/user/repo.git
 - [deleted]         feature/vecchia

# Metodo alternativo (push "nothing" al branch)
$ git push origin :feature/vecchia
```

4.7 Tracking Branches

4.7.1 Cosa sono tracking branches

Tracking branch: Branch locale configurato per seguire branch remoto.

Listing 4.17: Configurare tracking

```
# Crea branch locale trackando remoto
$ git checkout -b feature origin/feature
Branch 'feature' set up to track remote branch 'feature' from 'origin'.
Switched to a new branch 'feature'

# Metodo moderno (Git 2.23+)
$ git switch -c feature origin/feature

# Imposta tracking su branch esistente
$ git branch --set-upstream-to=origin/main main
Branch 'main' set up to track remote branch 'main' from 'origin'.

# Abbreviazione
$ git branch -u origin/main main
```

4.7.2 Visualizzare tracking

Listing 4.18: Vedere configurazione tracking

```
# Branch con tracking info
$ git branch -vv
* main    a3f5b21 [origin/main: ahead 2, behind 1] Latest commit
  develop b8c9d34 [origin/develop] Develop work
  feature c7e2a19 Not tracking anything

# Spiegazione:
# ahead 2  = 2 commit locali non pushati
# behind 1 = 1 commit remoto non pullato
```

4.7.3 Vantaggi tracking branches

- `git pull` senza argomenti (sa da dove pullare)
- `git push` senza argomenti (sa dove pushare)
- `git status` mostra ahead/behind
- Comandi più concisi

4.8 Fork e Upstream

4.8.1 Workflow con fork

Contribuire a progetti open source:

1. **Fork**: Crea copia del repo su tuo account GitHub
2. **Clone**: Clona tuo fork localmente
3. **Add upstream**: Aggiungi repo originale come remote
4. **Branch**: Crea feature branch
5. **Commit**: Sviluppa feature
6. **Push**: Push su tuo fork
7. **Pull Request**: Proponi merge a repo originale

Listing 4.19: Setup fork workflow

```
# 1. Fork su GitHub (click button "Fork")

# 2. Clone tuo fork
$ git clone https://github.com/tuousername/progetto.git
$ cd progetto

# 3. Aggiungi upstream (repo originale)
$ git remote add upstream https://github.com/originale/progetto.git

# 4. Verifica remote
$ git remote -v
origin    https://github.com/tuousername/progetto.git (fetch)
origin    https://github.com/tuousername/progetto.git (push)
upstream  https://github.com/originale/progetto.git (fetch)
upstream  https://github.com/originale/progetto.git (push)
```

4.8.2 Sync fork con upstream

Listing 4.20: Aggiornare fork da upstream

```
# Fetch modifiche da upstream
$ git fetch upstream

# Merge upstream/main in tuo main
$ git checkout main
$ git merge upstream/main

# Push modifiche su tuo fork
$ git push origin main

# Workflow completo in un comando (con rebase)
$ git pull upstream main --rebase
$ git push origin main
```

Riepilogo concetti chiave

Concetti fondamentali

- **Remote**: Repository su server (GitHub, GitLab) per condivisione
- `git remote add` configura remote, convenzione: `origin`
- `git clone` scarica repository completo e configura origin
- `git fetch` scarica commit senza modificare working directory
- `git pull` = fetch + merge (o rebase con `-rebase`)
- `git push` carica commit locali su remote
- `-u`/`-set-upstream` configura tracking branch
- **Tracking branch**: Branch locale associato a remote
- **Fork workflow**: fork → clone → upstream → feature → PR
- `-force` è pericoloso, usa solo su branch privati

Esercizi

1. Crea repository su GitHub, clonalo localmente, verifica remote con `git remote -v`.
2. Aggiungi file, committi, pusha su remote con `git push -u origin main`.
3. Modifica file su GitHub web, usa `git fetch` poi `git merge origin/main` separatamente.
4. Pratica `git pull -rebase` per storia lineare.
5. Crea branch locale, pusha con `-u`, poi usa `git push` senza argomenti.
6. Simula conflitto: modifica stessa riga localmente e su remote, risolvi durante pull.
7. Configura SSH key per GitHub, cambia remote da HTTPS a SSH.
8. Fork repository open source, aggiungi upstream, sync fork.
9. Usa `git branch -vv` per vedere ahead/behind status.
10. Crea shallow clone con `-depth 1`, confronta dimensione con clone normale.

Riferimenti

- Git Remote: https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes
- GitHub SSH Setup: https://docs.github.com/en/authentication/connecting-to-github-with-ssh
- Fork workflow: https://docs.github.com/en/get-started/quickstart/fork-a-repo

Capitolo 5

Workflow di Collaborazione

Introduzione

Un workflow Git definisce come team collabora usando branch, merge e release. Workflow strutturati prevengono caos e garantiscono qualità del codice. Questo capitolo presenta i workflow più popolari: Git Flow, GitHub Flow, Trunk-Based Development, e le pratiche di fork e pull request.

Obiettivi di apprendimento

- Comprendere perché servono workflow strutturati
- Implementare Git Flow per release cicliche
- Usare GitHub Flow per deploy continuo
- Conoscere Trunk-Based Development
- Gestire fork e contribuire a progetti open source
- Creare e review Pull Request
- Best practices per code review
- Strategie di release e versioning

5.1 Perché Servono Workflow

5.1.1 Il problema senza workflow

Senza workflow definito:

- Sviluppatori committano direttamente su `main`
- Produzione rompe frequentemente
- Feature incomplete vanno in produzione
- Impossibile rollback selettivo
- Hotfix mescolati con feature
- Code review inconsistente
- Chaos totale in team >2 persone

5.1.2 Workflow risolve

Un workflow ben definito fornisce:

- **Separazione**: Feature/hotfix/release isolati
- **Stabilità**: `main` sempre deployabile
- **Qualità**: Code review obbligatoria
- **Tracciabilità**: Ogni feature ha branch dedicato
- **Release control**: Gestione versioni chiara
- **Rollback**: Facile tornare indietro

Scegliere Workflow Giusto

Non esiste workflow perfetto. Dipende da:

- **Team size**: 2 persone vs 50 persone
- **Release cycle**: Continuous deploy vs release trimestrali
- **Tipo progetto**: Web app vs libreria vs mobile app
- **Cultura team**: Startup agile vs enterprise strutturata

Principio guida: **Semplicità**. Workflow complicato non viene seguito.

5.2 Git Flow

5.2.1 Panoramica Git Flow

Git Flow è workflow strutturato creato da Vincent Driessen nel 2010. Ideale per progetti con release cicliche pianificate.

Branch principali:

`main` Codice in produzione (sempre stabile)

`develop` Branch di integrazione (prossima release)

Branch di supporto:

`feature/*` Nuove feature (da develop)

`release/*` Preparazione release (da develop)

`hotfix/*` Fix urgenti produzione (da main)

5.2.2 Feature branches

Scopo: Sviluppare nuove feature isolate.

Listing 5.1: Workflow feature branch

```
# Crea feature branch da develop
$ git checkout develop
$ git checkout -b feature/user-profile

```

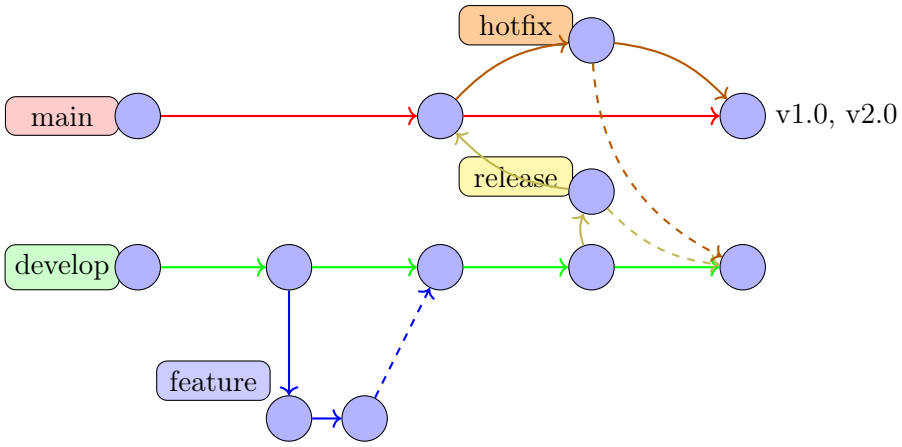


Figura 5.1: Git Flow: branch main, develop, feature, release, hotfix

```
# Sviluppa feature (commit multipli)
$ git add profile.py
$ git commit -m "Add user profile model"
$ git add profile_view.py
$ git commit -m "Add profile view"

# Merge feature in develop (quando completa)
$ git checkout develop
$ git merge --no-ff feature/user-profile
# --no-ff crea merge commit (preserva storia feature)

# Elimina feature branch
$ git branch -d feature/user-profile

# Push develop
$ git push origin develop
```

Naming convention:

- `feature/user-authentication`
- `feature/payment-integration`
- `feature/analytics-dashboard`

5.2.3 Release branches

Scopo: Preparare nuova release (fix minori, versioning, changelog).

Listing 5.2: Workflow release branch

```
# Crea release branch da develop
$ git checkout develop
$ git checkout -b release/1.2.0

# Fix minori, update versioning
$ echo "1.2.0" > VERSION
$ git commit -am "Bump version to 1.2.0"

# Update changelog
$ vim CHANGELOG.md
$ git commit -am "Update changelog for 1.2.0"
```

```
12
13 # Merge in main (produzione)
14 $ git checkout main
15 $ git merge --no-ff release/1.2.0
16 $ git tag -a v1.2.0 -m "Release version 1.2.0"
17
18 # Merge back in develop (per avere fix)
19 $ git checkout develop
20 $ git merge --no-ff release/1.2.0
21
22 # Elimina release branch
23 $ git branch -d release/1.2.0
24
25 # Push tutto
26 $ git push origin main develop --tags
```

Release branch permette:

- Develop continua con nuove feature
- Team QA testa release branch
- Fix minori su release senza bloccare develop
- Deploy quando pronto (non legato a develop)

5.2.4 Hotfix branches

Scopo: Fix urgenti in produzione.

Listing 5.3: Workflow hotfix branch

```
1  # Bug critico in produzione (main)
2  $ git checkout main
3  $ git checkout -b hotfix/security-patch
4
5  # Fix bug
6  $ vim security.py
7  $ git commit -am "Fix SQL injection vulnerability"
8
9  # Bump version (patch)
10 $ echo "1.2.1" > VERSION
11 $ git commit -am "Bump version to 1.2.1"
12
13 # Merge in main
14 $ git checkout main
15 $ git merge --no-ff hotfix/security-patch
16 $ git tag -a v1.2.1 -m "Hotfix: security patch"
17
18 # Merge in develop (per avere fix)
19 $ git checkout develop
20 $ git merge --no-ff hotfix/security-patch
21
22 # Elimina hotfix branch
23 $ git branch -d hotfix/security-patch
24
25 # Deploy urgente
26 $ git push origin main develop --tags
```

5.2.5 Vantaggi e svantaggi Git Flow

Vantaggi Git Flow

- Struttura chiara e ben documentata
- Adatto a release pianificate (ogni mese, trimestre)
- Separazione netta develop/produzione
- Hotfix non bloccano sviluppo feature
- Supporta release multiple in parallelo

Svantaggi Git Flow

- Complesso per team piccoli o progetti semplici
- Overhead di gestione branch
- Non adatto a continuous deployment
- Curva apprendimento per junior
- Merge conflicts frequenti con branch longevi

5.3 GitHub Flow

5.3.1 Panoramica GitHub Flow

GitHub Flow è workflow semplificato per continuous deployment. Usato da GitHub stesso e molte startup.

Principi:

1. `main` è sempre deployabile
2. Feature branch per ogni modifica
3. Pull request per code review
4. Deploy dopo merge (continuous)

Branch:

- `main`: Codice in produzione (protetto)
- `feature/*`: Branch temporanei per feature/fix

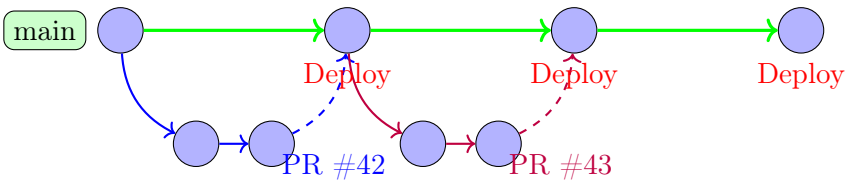


Figura 5.2: GitHub Flow: main + feature branches + PR + deploy

5.3.2 Workflow GitHub Flow

Listing 5.4: GitHub Flow step-by-step

```
# 1. Crea feature branch da main
$ git checkout main
$ git pull origin main
$ git checkout -b feature/add-search

# 2. Commit modifiche
$ git add search.py
$ git commit -m "Add basic search functionality"
$ git add search_tests.py
$ git commit -m "Add search tests"

# 3. Push branch
$ git push -u origin feature/add-search

# 4. Apri Pull Request su GitHub
#    - Descrivi modifiche
#    - Request review da colleghi
#    - CI/CD testa automaticamente

# 5. Code review
#    - Colleghi reviewano codice
#    - Suggerimenti e richieste modifiche
#    - Fix e push aggiuntivi

$ git add search.py
$ git commit -m "Address review feedback"
$ git push

# 6. Merge PR (quando approvata)
#    - Merge su GitHub (squash/merge commit/rebase)
#    - CI/CD deploya automaticamente
#    - Branch eliminato automaticamente

# 7. Pull main aggiornato
$ git checkout main
$ git pull origin main
$ git branch -d feature/add-search
```

5.3.3 Branch protection rules

Listing 5.5: Proteggere main branch (su GitHub)

```
# Settings > Branches > Branch protection rules
# Per "main":
- Require pull request before merging
- Require approvals: 2
- Require status checks to pass (CI)
- Require branches to be up to date
- Require conversation resolution before merging
- Do not allow bypassing the above settings
```

Effetto:

- Impossibile push diretto a `main`

- Tutte le modifiche via Pull Request
- Review obbligatoria (2 approvazioni)
- Test devono passare
- Garantisce qualità codice

5.3.4 Vantaggi e svantaggi GitHub Flow

Vantaggi GitHub Flow

- Semplicissimo: un branch principale + feature
- Ideale per continuous deployment
- Code review integrata (Pull Request)
- Deploy frequenti (anche più volte al giorno)
- Branch short-lived (meno conflitti)
- Curva apprendimento bassa

Svantaggi GitHub Flow

- Non adatto a release pianificate
- Richiede CI/CD robusto
- Richiede testing automatico esteso
- Difficile gestire multiple versioni in produzione
- Hotfix = feature branch (nessun trattamento speciale)

5.4 Trunk-Based Development

5.4.1 Panoramica

Trunk-Based Development è workflow estremo: sviluppatori committano direttamente su `main` (trunk) o usano branch brevissimi (1-2 giorni max).

Principi:

- Un branch principale (`main`/`trunk`)
- Commit piccoli e frequenti
- Feature flags per feature incomplete
- CI/CD estremamente robusto
- Test automatici estesi

Listing 5.6: Trunk-Based Development

```
# Commit diretto su main (se feature completa)
$ git checkout main
$ git pull
```

```
$ git add feature.py
$ git commit -m "Add small feature (behind feature flag)"
$ git push

# Oppure branch brevissimo (max 1-2 giorni)
$ git checkout -b quick-fix
$ git commit -am "Fix navbar alignment"
$ git push -u origin quick-fix
# Immediate PR + merge (stesso giorno)
```

5.4.2 Feature flags

Listing 5.7: Feature flags per trunk-based

```
# Codice in produzione ma feature disabilitata
def new_dashboard():
    if feature_flag('new_dashboard_enabled'):
        return render_new_dashboard()
    else:
        return render_old_dashboard()

# Deploy codice incomplete ma safe
# Abilita feature quando pronta (runtime toggle)
```

Vantaggi:

- Deploy continuo anche con feature incomplete
- Test in produzione con subset utenti
- Rollback istantaneo (toggle flag)
- No branch longevi

5.4.3 Quando usare Trunk-Based

Adatto per:

- Team senior esperti
- CI/CD molto maturo
- Test coverage >90%
- Deploy multipli al giorno
- Culture DevOps forte

Non adatto per:

- Team junior
- Progetti senza CI/CD
- Code review richiesta
- Release pianificate

5.5 Pull Request e Code Review

5.5.1 Cos'è una Pull Request

Pull Request (PR) è richiesta di merge branch feature in branch principale. Include:

- Descrizione modifiche
- Diff completo del codice
- Conversazione/commenti
- Review/approvazioni
- Status check CI/CD

5.5.2 Creare Pull Request efficace

Template Pull Request

Titolo: Add user authentication with JWT
Description:

```
## Cosa cambia
- Implementa JWT authentication
- Aggiunge login/logout endpoints
- Protegge API endpoints

## Perché
Richiesto per proteggere API da accessi non autorizzati.

## Come testare
1. POST /api/login con username/password
2. Ricevi JWT token
3. Usa token in header Authorization: Bearer <token>
4. Accedi a /api/protected (prima restituiva 401)

## Checklist
- [x] Test scritti e passano
- [x] Documentazione aggiornata
- [x] Nessun breaking change
- [x] Code review requested

## Screenshots
[Allegati screenshot se UI changes]

Fixes #123
```

5.5.3 Code Review Best Practices

Per autore PR:

- PR piccola (max 400 righe, ideale <200)
- Descrizione chiara e completa

- Self-review prima di request
- Test inclusi e passano
- Un PR = una feature/fix
- Rispondi a feedback costruttivamente

Per reviewer:

- Review entro 24h (non bloccare altri)
- Feedback costruttivo, non critica personale
- Distingui: *must fix* vs *nice to have*
- Approva quando "abbastanza buono", non "perfetto"
- Se molti commenti, call invece di 50 commenti

Listing 5.8: Commenti review esempi

```
# Bad review comment
"This is wrong."

# Good review comment
"Considera usare dict comprehension qui per migliore readability:
users = {u.id: u for u in user_list}
Cosa ne pensi?"

# Nitpick (non blocking)
"Nit: Typo in commento 'recieve' -> 'receive'"

# Blocking (must fix)
"Questo causa SQL injection. Usa parametrized query:
cursor.execute('SELECT * FROM users WHERE id = ?', (user_id,))"
```

5.5.4 Merge strategies su PR

Merge commit (default):

- Preserva storia completa
- Crea merge commit
- Storia non lineare

Squash and merge:

- Tutti commit feature → 1 commit
- Storia main pulita
- Perde storia intermedia feature

Rebase and merge:

- Replay commit su main
- Storia lineare

- Nessun merge commit

Listing 5.9: Configurare merge strategy (GitHub)

```
# Settings > General > Pull Requests
- Allow merge commits
- Allow squash merging (raccomandato per feature)
- Allow rebase merging

# Default: squash per main, merge per develop
```

5.6 Confronto Workflow

Caratteristica	Git Flow	GitHub Flow	Trunk-Based
Complessità	Alta	Bassa	Molto bassa
Branch principali	2 (main+develop)	1 (main)	1 (main)
Branch supporto	3 tipi	1 tipo	Nessuno/brevi
Release cycle	Pianificato	Continuo	Continuo
Team size	Grande	Medio	Piccolo/senior
CI/CD	Consigliato	Richiesto	Essenziale
Code review	Opzionale	Obbligatorio	Opzionale
Curva apprendimento	Ripida	Moderata	Bassa

Tabella 5.1: Confronto workflow Git

Riepilogo concetti chiave

Concetti fondamentali

- **Workflow** definisce come team collabora con Git
- **Git Flow**: main+develop, feature/release/hotfix, release pianificate
- **GitHub Flow**: main+feature, PR, continuous deployment
- **Trunk-Based**: commit diretti/branch brevissimi, feature flags
- **Pull Request**: Richiesta merge + code review + CI
- **Branch protection**: Impedisce push diretto, forza PR
- **Code review**: Qualità codice, knowledge sharing
- **Merge strategies**: merge commit vs squash vs rebase
- Workflow dipende da: team size, release cycle, CI/CD maturity
- **Semplicità** è chiave: workflow complesso non viene seguito

Esercizi

1. Simula Git Flow: crea `develop`, feature branch, release branch, merge in `main`.

2. Pratica GitHub Flow: crea feature, push, apri PR (su GitHub), merge.
3. Configura branch protection su repository GitHub per `main`.
4. Scrivi PR description completa per feature recente (usa template).
5. Fai code review di PR (proprio o altrui), pratica commenti costruttivi.
6. Sperimenta merge strategies: merge commit vs squash vs rebase, confronta storia.
7. Simula hotfix workflow in Git Flow.
8. Configura GitHub Actions per CI su PR (test automatici).
9. Crea fork, contribuisci a progetto open source, apri PR.
10. Discuti con team quale workflow è adatto al vostro progetto.

Riferimenti

- Git Flow originale: https://nvie.com/posts/a-successful-git-branching-model/
- GitHub Flow: https://guides.github.com/introduction/flow/
- Trunk-Based Development: https://trunkbaseddevelopment.com/
- Code Review Best Practices: https://google.github.io/eng-practices/review/
- Pull Request template: https://docs.github.com/en/communities/using-templates-to-encourage-u

Capitolo 6

Git Avanzato

Introduzione

Questo capitolo esplora funzionalità avanzate di Git che permettono di gestire situazioni complesse nel workflow di sviluppo. Imparerai a salvare temporaneamente modifiche con stash, selezionare commit specifici con cherry-pick, manipolare la cronologia con reset, recuperare commit persi con reflog, trovare bug con bisect e gestire versioni con i tag.

Obiettivi di apprendimento

- Usare `git stash` per salvare temporaneamente modifiche non committate
- Applicare commit specifici da altri branch con `git cherry-pick`
- Comprendere le differenze tra reset soft, mixed e hard
- Recuperare commit persi con `git reflog`
- Trovare bug usando la ricerca binaria con `git bisect`
- Creare e gestire tag per versioni del software
- Navigare e manipolare la cronologia di Git in modo efficace

6.1 Git Stash: Salvare Modifiche Temporanee

6.1.1 Cos'è lo Stash

Lo stash è un'area temporanea dove salvare modifiche non committate. È utile quando devi cambiare branch velocemente ma non vuoi fare commit di lavori incompleti.

Scenario Tipico

Stai lavorando su una feature quando ti chiedono urgentemente di fixare un bug su master. Non vuoi committare codice incompleto, ma neanche perdere il lavoro fatto. Soluzione: `git stash`.

6.1.2 Comandi Base dello Stash

```
# Salvare modifiche nello stash
git stash
# oppure con messaggio descrittivo
git stash save "WIP: implementazione login"

# Vedere lista degli stash
git stash list

# Applicare l'ultimo stash (mantiene lo stash)
git stash apply

# Applicare e rimuovere l'ultimo stash
git stash pop

# Applicare uno stash specifico
git stash apply stash@{2}

# Rimuovere uno stash
git stash drop stash@{0}

# Cancellare tutti gli stash
git stash clear
```

6.1.3 Esempio Pratico: Cambio Branch con Stash

```
# Situazione: stai modificando file su feature-branch
git status
# modified: src/login.js
# modified: src/utils.js

# Salva le modifiche
git stash save "Login form validation"

# Ora puoi cambiare branch
git checkout main

# Fix il bug urgente
echo "bugfix" >> hotfix.txt
git add hotfix.txt
git commit -m "Fix critical bug"

# Torna al tuo branch
git checkout feature-branch

# Recupera le modifiche
git stash pop
# I file modificati tornano nella working directory
```

6.1.4 Stash Avanzato

```
# Includere file untracked nello stash
git stash -u
# oppure
git stash --include-untracked

```

```
# Includere anche file ignored
git stash -a
# oppure
git stash --all

# Creare un branch da uno stash
git stash branch nuovo-branch stash@{1}

# Visualizzare dettagli di uno stash
git stash show stash@{0}

# Vedere diff completo
git stash show -p stash@{0}
```

Attenzione: Stash non è Permanente

Gli stash sono locali e non vengono pushati. Non usare stash per salvare lavoro a lungo termine: usa branch e commit invece.

6.2 Git Cherry-Pick: Selezionare Commit Specifici

6.2.1 Cos'è Cherry-Pick

Cherry-pick permette di applicare modifiche di un commit specifico al branch corrente, senza fare merge dell'intero branch.

```
# Applicare un singolo commit
git cherry-pick <commit-hash>

# Applicare multipli commit
git cherry-pick <commit1> <commit2> <commit3>

# Applicare un range di commit
git cherry-pick <commit-start>..<commit-end>
```

6.2.2 Esempio Pratico: Portare Fix da un Branch

```
# Situazione: hai fatto un fix importante su develop
git checkout develop
git log --oneline
# abc123 Fix security vulnerability
# def456 Add feature X
# ...

# Vuoi portare solo il fix su main
git checkout main
git cherry-pick abc123

# Il commit abc123 viene applicato su main
# con un nuovo hash (perché parent diverso)
```

6.2.3 Cherry-Pick con Conflitti

```
# Tentativo di cherry-pick
git cherry-pick abc123

# Se ci sono conflitti
# CONFLICT (content): Merge conflict in file.js
# error: could not apply abc123... Fix security

# Risolvi i conflitti manualmente
vim file.js  # risolvi conflitti

# Continua il cherry-pick
git add file.js
git cherry-pick --continue

# Oppure abbandona
git cherry-pick --abort
```

6.2.4 Opzioni Avanzate di Cherry-Pick

```
# Cherry-pick senza committare (staging only)
git cherry-pick -n <commit>
git cherry-pick --no-commit <commit>

# Modificare il messaggio del commit
git cherry-pick -e <commit>
git cherry-pick --edit <commit>

# Mantenere l'autore originale
git cherry-pick -x <commit>
# Aggiunge "(cherry picked from commit ...)"
```

Best Practice

Usa cherry-pick con cautela. Crea duplicazione nella cronologia e può causare confusione. Preferisci merge quando possibile. Cherry-pick è ottimo per:

- Hotfix urgenti da applicare a multiple branch
- Recuperare commit da branch che verranno cancellati
- Applicare commit specifici senza tutto il branch

6.3 Git Reset: Manipolare la Cronologia

6.3.1 Tre Modalità di Reset

Git reset sposta il puntatore HEAD (e opzionalmente modifica staging area e working directory). Esistono tre modalità:

`-soft` Sposta solo HEAD. Staging area e working directory restano invariati.

`-mixed` (default) Sposta HEAD e resetta staging area. Working directory invariata.

`-hard` Sposta HEAD, resetta staging area E working directory. **ATTENZIONE: Perdita dati!**

6.3.2 Reset Soft

```
# Situazione: hai fatto 3 commit ma vuoi combinali in uno
git log --oneline
# c789 Add tests
# b456 Add documentation
# a123 Add feature

# Torna indietro di 2 commit ma mantieni le modifiche
git reset --soft HEAD~2

# Ora le modifiche dei 3 commit sono in staging
git status
# Changes to be committed:
#   modified: feature.js
#   new file: feature.test.js
#   modified: README.md

# Crea un singolo commit
git commit -m "Add feature with tests and docs"
```

Quando Usare Reset Soft

Perfetto per:

- Combinare multipli commit in uno
- Riscrivere messaggi di commit multipli
- Riorganizzare modifiche già committate

6.3.3 Reset Mixed (Default)

```
# Hai aggiunto file allo staging per errore
git add file1.txt file2.txt file3.txt
git status
# Changes to be committed:
#   modified: file1.txt
#   modified: file2.txt
#   modified: file3.txt

# Rimuovi tutto dallo staging
git reset
# oppure esplicitamente
git reset --mixed HEAD

git status
# Changes not staged for commit:
#   modified: file1.txt
#   modified: file2.txt
#   modified: file3.txt
```

6.3.4 Reset Hard: PERICOLO!

```
# ATTENZIONE: Questo cancella modifiche!
git reset --hard HEAD~1
```

```
3
4 # Tutte le modifiche del commit sono PERSE
5 # Working directory torna allo stato del commit precedente
```

PERICOLO: Reset Hard

`git reset -hard` cancella permanentemente modifiche non committate! Usa con estrema cautela. Verifica sempre:

```
1 git status  # cosa perderai
2 git stash   # salva modifiche prima se necessario
```

6.3.5 Reset vs Revert

```
1 # Reset: riscrive cronologia (pericoloso su branch condivisi)
2 git reset --hard HEAD~1
3
4 # Revert: crea nuovo commit che annulla modifiche (sicuro)
5 git revert HEAD
```

Regola d'Oro

Mai usare reset su commit già pushati e condivisi! Usa `git revert` invece.

- **Reset**: OK per commit locali
- **Revert**: OK per commit pushati

6.4 Git Reflog: Recuperare Commit Persi

6.4.1 Cos'è Reflog

Reflog (reference log) è un registro locale di tutte le modifiche a HEAD. Anche se un commit sembra "perso" dopo un reset, reflog permette di recuperarlo.

```
1 # Visualizzare reflog
2 git reflog
3
4 # Output esempio:
5 # a1b2c3d HEAD@{0}: commit: Add feature
6 # e4f5g6h HEAD@{1}: reset: moving to HEAD~1
7 # i7j8k9l HEAD@{2}: commit: Bugfix
8 # ...
```

6.4.2 Recupero da Reset Hard

```
1 # Situazione disastrosa: hai fatto reset hard per errore
2 git reset --hard HEAD~3
3 # OH NO! Hai perso 3 commit importanti!
4
5 # Calma, usa reflog
6 git reflog
7 # Output:
8 # a1b2c3d HEAD@{0}: reset: moving to HEAD~3
```

```
# m4n5o6p HEAD@{1}: commit: Important work
# q7r8s9t HEAD@{2}: commit: More important work
# u0v1w2x HEAD@{3}: commit: Even more work

# Recupera tornando allo stato prima del reset
git reset --hard HEAD@{1}
# oppure usando l'hash direttamente
git reset --hard m4n5o6p

# I tuoi commit sono tornati!
```

6.4.3 Altri Usi di Reflog

```
# Vedere reflog di un branch specifico
git reflog show feature-branch

# Vedere solo ultimi N movimenti
git reflog -5

# Creare branch da posizione reflog
git branch recovered-branch HEAD@{3}

# Vedere diff tra due posizioni reflog
git diff HEAD@{0} HEAD@{2}
```

Reflog è il Tuo Salvavita

Reflog conserva la cronologia per 90 giorni (default). Se hai fatto un errore, reflog può salvare il tuo lavoro. Ma ricorda:

- Reflog è locale (non viene pushato)
- Dopo 90 giorni le entry vengono rimosse
- `git gc` può pulire oggetti non referenziati

6.5 Git Bisect: Trovare Bug con Ricerca Binaria

6.5.1 Cos'è Bisect

Bisect usa ricerca binaria per trovare il commit che ha introdotto un bug. Invece di controllare centinaia di commit manualmente, Git dimezza automaticamente lo spazio di ricerca.

6.5.2 Workflow Bisect Manuale

```
# Inizia bisect
git bisect start

# Marca commit corrente come cattivo (ha il bug)
git bisect bad

# Marca un commit vecchio come buono (senza bug)
git bisect good v1.0
# oppure
git bisect good a1b2c3d
```

```
11
12 # Git checkoutterà commit a metà tra good e bad
13 # Testa se il bug è presente
14
15 # Se il bug c'è
16 git bisect bad
17
18 # Se il bug non c'è
19 git bisect good
20
21 # Ripeti finché Git trova il commit colpevole
22 # Git mostrerà:
23 # abc123 is the first bad commit
24 # commit abc123...
25
26 # Termina bisect
27 git bisect reset
```

6.5.3 Esempio Pratico di Bisect

```
1  # Situazione: il test fallisce ora, ma passava 2 settimane fa
2  git log --oneline
3  # Ci sono 50 commit tra oggi e 2 settimane fa
4
5  git bisect start
6  git bisect bad                    # HEAD ha il bug
7  git bisect good HEAD~50          # 50 commit fa ok
8
9  # Git checkout commit #25
10 # Bisecting: 25 revisions left to test
11
12 # Esegui test
13 npm test
14
15 # Test fallisce
16 git bisect bad
17
18 # Git checkout commit #12
19 # Bisecting: 12 revisions left to test
20
21 npm test
22
23 # Test passa
24 git bisect good
25
26 # Continua...
27 # Dopo ~6 iterazioni (log2(50))
28
29 # abc123 is the first bad commit
30 # commit abc123
31 # Author: Developer
32 # Date: 3 days ago
33 #     Refactor database connection
34
35 git bisect reset
36 git show abc123  # analizza il commit colpevole
```

6.5.4 Bisect Automatico

```
# Automatizzare bisect con uno script di test
git bisect start HEAD v1.0
git bisect run ./test.sh

# test.sh deve:
# - exit 0 se test passa (good)
# - exit 1-127 (tranne 125) se test fallisce (bad)
# - exit 125 per skip (commit non testabile)
```

Esempio `test.sh`:

```
#!/bin/bash
# test.sh

make clean
make || exit 125  # skip se non compila

# Esegui test
./run_tests

# Exit code del test diventa exit code dello script
exit $?
```

Efficienza di Bisect

Con 1000 commit, bisect trova il colpevole in circa 10 test ($\log_2(1000) \approx 10$). Manualmente impiegheresti ore!

6.6 Git Tag: Versioning e Release

6.6.1 Tipi di Tag

Git supporta due tipi di tag:

Lightweight tag Semplice puntatore a un commit (come un branch che non si muove)

Annotated tag Oggetto Git completo con autore, data, messaggio, firma GPG

6.6.2 Creare Tag

```
# Tag lightweight (sconsigliato per release)
git tag v1.0.0

# Tag annotato (raccomandato)
git tag -a v1.0.0 -m "Release 1.0.0 - First stable version"

# Tag su commit specifico
git tag -a v0.9.0 a1b2c3d -m "Beta release"

# Tag con firma GPG
git tag -s v1.0.0 -m "Signed release"
```

6.6.3 Visualizzare Tag

```
# Elenco tutti i tag
git tag

# Cercare tag con pattern
git tag -l "v1.*"
# v1.0.0
# v1.0.1
# v1.1.0

# Mostrare informazioni tag annotato
git show v1.0.0

# Output:
# tag v1.0.0
# Tagger: Developer
# Date: ...
# Release 1.0.0 - First stable version
# commit abc123...
```

6.6.4 Condividere Tag

```
# Push singolo tag
git push origin v1.0.0

# Push tutti i tag
git push origin --tags

# Push solo tag annotati
git push origin --follow-tags
```

6.6.5 Cancellare Tag

```
# Cancellare tag locale
git tag -d v1.0.0

# Cancellare tag remoto
git push origin --delete v1.0.0
# oppure
git push origin :refs/tags/v1.0.0
```

6.6.6 Checkout di Tag

```
# Vedere codice di una release specifica
git checkout v1.0.0
# HEAD is now at abc123... (detached HEAD state)

# Per lavorare su quella versione, crea branch
git checkout -b bugfix-1.0 v1.0.0
```

6.6.7 Semantic Versioning

Semantic Versioning (SemVer)

Schema di versioning: `MAJOR.MINOR.PATCH`

- **MAJOR**: Cambiamenti incompatibili (breaking changes)
- **MINOR**: Nuove funzionalità compatibili
- **PATCH**: Bugfix compatibili

Esempi:

- `v1.0.0`: Prima versione stabile
- `v1.1.0`: Aggiunte feature (compatibile con 1.0.0)
- `v1.1.1`: Bugfix
- `v2.0.0`: Breaking changes (non compatibile con 1.x.x)

6.7 Esercizi

6.7.1 Esercizio 1: Stash Practice

1. Crea un nuovo repository e file `app.js`
2. Modifica `app.js` ma non committare
3. Usa `git stash` per salvare le modifiche
4. Crea e committa un altro file
5. Recupera le modifiche con `git stash pop`

6.7.2 Esercizio 2: Cherry-Pick Simulation

1. Crea branch `develop` e `main`
2. Su `develop` fai 3 commit
3. Cherry-pick solo il secondo commit su `main`
4. Verifica che `main` abbia solo quel commit

6.7.3 Esercizio 3: Reset Modes

1. Crea 3 commit consecutivi
2. Usa `git reset -soft HEAD 2` e osserva staging
3. Ripristina con reflog
4. Prova `git reset -mixed HEAD 1`
5. Confronta le differenze

6.7.4 Esercizio 4: Reflog Recovery

1. Crea alcuni commit
2. Esegui `git reset -hard HEAD 3`
3. Usa `git reflog` per trovare i commit persi
4. Recupera usando `git reset -hard HEAD@n`

6.7.5 Esercizio 5: Tag Release

1. Crea tag annotato `v1.0.0` su HEAD
2. Fai altri commit
3. Crea tag `v1.1.0`
4. Elenca tutti i tag
5. Fai checkout del tag `v1.0.0` e osserva il codice

6.7.6 Esercizio 6: Bisect Challenge (Avanzato)

1. Crea repository con file `calc.sh` che contiene funzione matematica
2. Fai 20 commit, introducendo un bug al commit 13
3. Usa `git bisect` per trovare il commit che ha introdotto il bug
4. Bonus: Crea script di test e usa `git bisect run`

Soluzioni

Le soluzioni dettagliate degli esercizi sono nell'Appendice Esercizi alla fine del libro.

Capitolo 7

GitHub e GitLab

Introduzione

GitHub e GitLab sono piattaforme di hosting per repository Git che offrono strumenti di collaborazione, CI/CD, issue tracking e molto altro. Questo capitolo esplora le funzionalità principali di entrambe le piattaforme, con focus su GitHub che è la più diffusa.

Obiettivi di apprendimento

- Comprendere differenze tra GitHub e GitLab
- Gestire repository remoti su piattaforme cloud
- Usare issues per tracciare bug e feature
- Organizzare progetti con GitHub Projects
- Creare e gestire Pull Request
- Automatizzare workflow con GitHub Actions
- Configurare webhooks per integrazioni
- Pubblicare siti statici con GitHub Pages

7.1 GitHub vs GitLab: Confronto

7.1.1 Caratteristiche Principali

Caratteristica	**GitHub**	**GitLab**
Hosting	Cloud / Enterprise	Cloud / Self-hosted
Repository privati	Illimitati (gratis)	Illimitati (gratis)
CI/CD integrato	GitHub Actions	GitLab CI (più maturo)
Issue tracking	Sì	Sì (più avanzato)
Wiki	Sì	Sì
Container registry	Sì	Sì (integrato)
Code review	Pull Request	Merge Request
Community	Più grande	Crescente

Quale Scegliere?

- **GitHub**: Preferito per open source, community enorme, integrazioni abbondanti
- **GitLab**: Migliore per CI/CD complesso, self-hosting, DevOps completo
- Entrambi offrono funzionalità simili per la maggior parte dei casi d'uso

7.2 Repository su GitHub

7.2.1 Creare un Repository

```
# Via web: github.com -> New repository
# Nome: my-project
# Visibilità: Public / Private
# Initialize: README, .gitignore, License

# Collegare repository locale a GitHub
git remote add origin https://github.com/username/my-project.git

# Push iniziale
git branch -M main
git push -u origin main
```

7.2.2 Clonare Repository

```
# Clone via HTTPS
git clone https://github.com/username/repo.git

# Clone via SSH (raccomandato)
git clone git@github.com:username/repo.git

# Clone di branch specifico
git clone -b develop https://github.com/username/repo.git

# Clone shallow (solo ultimo commit, più veloce)
git clone --depth 1 https://github.com/username/repo.git
```

7.2.3 SSH Keys per GitHub

```
# Generare chiave SSH
ssh-keygen -t ed25519 -C "your_email@example.com"

# Aggiungere a ssh-agent
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519

# Copiare chiave pubblica
cat ~/.ssh/id_ed25519.pub

# Incollare in GitHub:
# Settings -> SSH and GPG keys -> New SSH key

# Testare connessione
```

```
ssh -T git@github.com
# Hi username! You've successfully authenticated...
```

7.3 Issues: Tracciamento Attività

7.3.1 Cosa Sono le Issues

Le issues sono il sistema di ticket per tracciare:

- Bug reports
- Feature requests
- Domande e discussioni
- To-do list per il progetto

7.3.2 Anatomia di una Issue

Esempio Issue

Titolo: Login button not responsive on mobile
Descrizione:

```
## Bug Description
The login button on mobile devices doesn't respond to touch events.

## Steps to Reproduce
1. Open app on mobile (iOS/Android)
2. Navigate to login page
3. Tap login button
4. Nothing happens

## Expected Behavior
Should submit login form

## Environment
- Device: iPhone 12
- OS: iOS 15.2
- Browser: Safari

## Screenshots
[attach screenshot]
```

Labels: bug, mobile, high-priority
Assignee: @developer-name
Milestone: v1.2.0

7.3.3 Comandi Issue da Git

```
# Referenziare issue in commit
git commit -m "Fix login button (#42)"

# Chiudere issue automaticamente
git commit -m "Fix mobile login

Fixes #42
```

```
Closes #43"

# Quando fai push, issues #42 e #43 vengono chiuse
```

7.3.4 Labels e Milestones

```
# Labels comuni:
bug              # Errori da fixare
enhancement      # Nuove feature
documentation    # Miglioramenti doc
good first issue # Per nuovi contributor
help wanted      # Richiesta aiuto
wontfix          # Non verrà implementato
duplicate        # Duplicato di altra issue

# Milestones:
v1.0.0 - Initial Release (15 issues)
v1.1.0 - Mobile Support (8 issues)
v2.0.0 - Major Refactor (23 issues)
```

7.3.5 Issue Templates

File `.github/ISSUE_TEMPLATE/bug_report.md`:

```
---
name: Bug Report
about: Report a bug
title: '[BUG] '
labels: bug
assignees: ''
---

## Describe the bug
A clear description of what the bug is.

## To Reproduce
Steps to reproduce:
1. Go to '...'
2. Click on '...'
3. See error

## Expected behavior
What you expected to happen.

## Screenshots
If applicable, add screenshots.

## Environment
- OS: [e.g. Windows 10]
- Browser: [e.g. Chrome 95]
- Version: [e.g. 1.2.0]
```

7.4 GitHub Projects: Project Management

7.4.1 Creazione Project Board

1. Repository → Projects → New Project
2. Scegli template: Kanban, Table, Roadmap
3. Configura colonne: To Do, In Progress, Done

7.4.2 Automazione Project Board

```
# Workflow automatico:
- Issue aperta -> colonna "To Do"
- PR aperta -> colonna "In Progress"
- PR merged -> colonna "Done"

# Configurazione automazione:
Project Settings -> Workflows -> Enable automation
```

7.4.3 Esempio Kanban Board

Sprint Board Example

Backlog: 12 issues
To Do (Sprint corrente):

- #45 Implement user authentication
- #46 Add password reset
- #47 Create dashboard

In Progress:

- #42 Fix mobile login (assigned to @dev1)
- #43 Optimize database queries (assigned to @dev2)

Code Review:

- #40 Add email notifications (PR #105)

Done (questo sprint):

- #38 Setup CI/CD pipeline
- #39 Configure production server

7.5 Pull Requests

7.5.1 Workflow Pull Request

```
# 1. Crea branch per feature
git checkout -b feature/user-auth

# 2. Implementa e committa
```

```
git add .
git commit -m "Add user authentication system"

# 3. Push branch
git push -u origin feature/user-auth

# 4. Apri PR su GitHub
# Repository -> Pull Requests -> New Pull Request
# Base: main <- Compare: feature/user-auth

# 5. Code review e discussione

# 6. Approva e merge
# Merge pull request (create merge commit)
# Squash and merge (combina commit)
# Rebase and merge (linear history)
```

7.5.2 Template Pull Request

File `.github/PULL_REQUEST_TEMPLATE.md`:

```
## Description
Brief description of changes.

## Type of Change
- [ ] Bug fix
- [ ] New feature
- [ ] Breaking change
- [ ] Documentation update

## Related Issues
Fixes #(issue)

## Testing
Describe tests performed:
- [ ] Unit tests pass
- [ ] Integration tests pass
- [ ] Manual testing completed

## Checklist
- [ ] Code follows style guidelines
- [ ] Self-review completed
- [ ] Comments added for complex code
- [ ] Documentation updated
- [ ] No new warnings generated

## Screenshots
If applicable.
```

7.5.3 Code Review Best Practices

Come Fare Code Review

Reviewer:

- Sii costruttivo e specifico nei commenti
- Suggerisci soluzioni, non solo problemi

- Approva quando il codice è sufficientemente buono
- Usa "Request changes" solo per problemi bloccanti

Author:

- Mantieni PR piccole (< 400 righe)
- Descrivi chiaramente le modifiche
- Rispondi a tutti i commenti
- Non prendere critiche personalmente

7.5.4 Protected Branches

```
# Settings -> Branches -> Add rule

Branch name pattern: main

Protezioni:
[X] Require pull request before merging
    [X] Require approvals (minimo 1-2)
    [X] Dismiss stale reviews
[X] Require status checks to pass
    [X] Require branches to be up to date
    - CI/CD tests
    - Code quality checks
[X] Require linear history
[X] Include administrators
```

7.6 GitHub Actions: CI/CD

7.6.1 Cos'è GitHub Actions

GitHub Actions automatizza workflow di sviluppo: testing, building, deployment, e altro. I workflow sono definiti in file YAML in `.github/workflows/`.

7.6.2 Workflow Base: Test su Push

File `.github/workflows/test.yml`:

```yaml
name: Run Tests

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
```

```
      uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '18'

    - name: Install dependencies
      run: npm ci

    - name: Run tests
      run: npm test

    - name: Upload coverage
      uses: codecov/codecov-action@v3
      with:
        files: ./coverage/coverage.xml
```

7.6.3 Workflow Avanzato: Matrix Testing

```
name: Matrix Test

on: [push, pull_request]

jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node: [14, 16, 18]

    steps:
    - uses: actions/checkout@v3
    - name: Setup Node ${{ matrix.node }}
      uses: actions/setup-node@v3
      with:
        node-version: ${{ matrix.node }}
    - run: npm ci
    - run: npm test
```

7.6.4 Deployment Automatico

```
name: Deploy to Production

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

```

```
    - name: Build application
      run: |
        npm ci
        npm run build

    - name: Deploy to server
      uses: appleboy/ssh-action@master
      with:
        host: ${{ secrets.SERVER_HOST }}
        username: ${{ secrets.SERVER_USER }}
        key: ${{ secrets.SSH_PRIVATE_KEY }}
        script: |
          cd /var/www/app
          git pull origin main
          npm install
          pm2 restart app
```

7.6.5 Secrets e Environment Variables

```
# Settings -> Secrets and variables -> Actions

Secrets (encrypted):
- DATABASE_URL
- API_KEY
- SSH_PRIVATE_KEY

# Uso nel workflow:
env:
  DATABASE_URL: ${{ secrets.DATABASE_URL }}

# O direttamente:
run: echo "${{ secrets.API_KEY }}" > .env
```

7.7 Webhooks

7.7.1 Cos'è un Webhook

Un webhook è un HTTP callback che GitHub invia al tuo server quando avvengono eventi specifici nel repository.

7.7.2 Configurazione Webhook

```
# Settings -> Webhooks -> Add webhook

Payload URL: https://your-server.com/webhook
Content type: application/json
Secret: your-secret-token

Events:
[X] Push
[X] Pull request
[X] Issues
[ ] Just the push event
[ ] Send me everything
```

7.7.3 Esempio Server Webhook (Node.js)

```
const express = require('express');
const crypto = require('crypto');
const app = express();

app.use(express.json());

// Verifica firma GitHub
function verifySignature(req) {
    const signature = req.headers['x-hub-signature-256'];
    const hmac = crypto.createHmac('sha256', process.env.WEBHOOK_SECRET);
    const digest = 'sha256=' + hmac.update(
        JSON.stringify(req.body)
    ).digest('hex');
    return signature === digest;
}

app.post('/webhook', (req, res) => {
    if (!verifySignature(req)) {
        return res.status(403).send('Invalid signature');
    }

    const event = req.headers['x-github-event'];
    const payload = req.body;

    switch(event) {
        case 'push':
            console.log(`Push to ${payload.ref}`);
            console.log(`Commits: ${payload.commits.length}`);
            // Trigger deployment, tests, etc.
            break;

        case 'pull_request':
            console.log(`PR #${payload.number}: ${payload.action}`);
            // Notify team, run checks, etc.
            break;

        case 'issues':
            console.log(`Issue #${payload.issue.number}: ${payload.action}`);
            // Add to project, notify, etc.
            break;
    }

    res.status(200).send('Webhook received');
});

app.listen(3000, () => console.log('Webhook server running'));
```

7.8 GitHub Pages

7.8.1 Pubblicare Sito Statico

```
# Opzione 1: Branch gh-pages
```

```
git checkout -b gh-pages
# Aggiungi file HTML/CSS/JS
git add .
git commit -m "Initial site"
git push origin gh-pages

# Settings -> Pages -> Source: gh-pages branch

# Sito disponibile a:
# https://username.github.io/repository-name/
```

7.8.2 Opzione 2: GitHub Actions per Deploy

File `.github/workflows/deploy-pages.yml`:

```
name: Deploy to GitHub Pages

on:
  push:
    branches: [ main ]

permissions:
  contents: read
  pages: write
  id-token: write

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup Node
        uses: actions/setup-node@v3
        with:
          node-version: '18'

      - name: Install and build
        run: |
          npm ci
          npm run build

      - name: Upload artifact
        uses: actions/upload-pages-artifact@v1
        with:
          path: ./dist

  deploy:
    environment:
      name: github-pages
      url: ${{ steps.deployment.outputs.page_url }}
    runs-on: ubuntu-latest
    needs: build
    steps:
      - name: Deploy to GitHub Pages
        id: deployment
        uses: actions/deploy-pages@v1
```

7.8.3 Custom Domain

```
# Aggiungi file CNAME nel repository
echo "www.yoursite.com" > CNAME
git add CNAME
git commit -m "Add custom domain"
git push

# Configura DNS:
# CNAME record: www -> username.github.io

# Settings -> Pages -> Custom domain
# Abilita HTTPS
```

7.9 GitLab: Differenze Chiave

7.9.1 GitLab CI/CD

File `.gitlab-ci.yml`:

```
stages:
  - test
  - build
  - deploy

test:
  stage: test
  image: node:18
  script:
    - npm ci
    - npm test
  coverage: '/Coverage: \d+\.\d+%/'

build:
  stage: build
  script:
    - npm run build
  artifacts:
    paths:
      - dist/

deploy_production:
  stage: deploy
  script:
    - ./deploy.sh
  only:
    - main
  when: manual
```

7.9.2 Merge Requests vs Pull Requests

- **Nome diverso**: GitLab usa "Merge Request", GitHub usa "Pull Request"
- **Funzionalità simili**: Code review, discussione, approval
- **GitLab extra**: Merge trains, merge when pipeline succeeds

7.10 Esercizi

7.10.1 Esercizio 1: Setup Repository

1. Crea account GitHub (se non hai già)
2. Crea nuovo repository pubblico "git-practice"
3. Aggiungi README, .gitignore (Node), License (MIT)
4. Clona localmente e aggiungi file
5. Push modifiche

7.10.2 Esercizio 2: Issue Tracking

1. Crea 3 issues: 1 bug, 1 feature, 1 documentation
2. Aggiungi labels appropriate
3. Crea milestone "v1.0"
4. Assegna issues al milestone
5. Chiudi una issue tramite commit message

7.10.3 Esercizio 3: Pull Request Workflow

1. Crea branch `feature/add-calculator`
2. Implementa semplice calcolatrice
3. Push branch e apri PR
4. Aggiungi descrizione dettagliata
5. Simula review (commenta il tuo codice)
6. Merge PR

7.10.4 Esercizio 4: GitHub Actions

1. Crea workflow che esegue test su push
2. Aggiungi badge status al README
3. Configura matrix testing per Node 16 e 18
4. Aggiungi step per code coverage
5. Verifica che workflow funzioni

7.10.5 Esercizio 5: GitHub Pages

1. Crea semplice sito HTML nel repository
2. Configura GitHub Pages da branch main
3. Verifica che sito sia online
4. Modifica sito e verifica deploy automatico
5. Bonus: Usa Jekyll o framework static site

Risorse

- GitHub Docs: https://docs.github.com
- GitHub Actions Marketplace: https://github.com/marketplace?type=actions
- GitLab Docs: https://docs.gitlab.com
- GitHub Learning Lab: https://lab.github.com

Capitolo 8

Best Practices e Convenzioni

Introduzione

Scrivere buon codice non basta: è fondamentale usare Git in modo efficace. Questo capitolo presenta best practices consolidate per commit messages, struttura dei commit, branching strategies e workflow collaborativi. Seguire queste convenzioni rende il progetto più manutenibile e la collaborazione più fluida.

Obiettivi di apprendimento

- Scrivere commit messages chiari e informativi
- Creare commit atomici e logicamente coerenti
- Scegliere la branching strategy appropriata per il progetto
- Applicare best practices per code review
- Decidere quando usare merge vs rebase
- Mantenere una cronologia Git pulita e comprensibile
- Gestire .gitignore efficacemente
- Implementare security best practices

8.1 Commit Messages: L'Arte della Comunicazione

8.1.1 Anatomia di un Buon Commit Message

```
<type>(<scope>): <subject>

<body>

<footer>
```

Esempio:

```
feat(auth): add JWT token authentication

Implement JSON Web Token based authentication system to replace
session-based auth. This provides better scalability and enables
stateless API design.
```

```
6
7  - Add JWT generation and validation
8  - Implement middleware for protected routes
9  - Add token refresh mechanism
10 - Update user model with token fields
11
12 Closes #142
13 Breaking change: Session-based auth is deprecated
```

8.1.2 Conventional Commits

`feat:` Nuova funzionalità

`fix:` Bug fix

`docs:` Solo modifiche documentazione

`style:` Formattazione, missing semicolons, etc. (non cambia codice)

`refactor:` Refactoring (né fix né feature)

`perf:` Miglioramento performance

`test:` Aggiunta o correzione test

`build:` Build system, dipendenze (webpack, npm, etc.)

`ci:` CI configuration (GitHub Actions, Travis, etc.)

`chore:` Manutenzione varia (update dependencies, etc.)

8.1.3 Esempi di Commit Messages

Esempi BUONI

```
1  feat(api): add pagination to user list endpoint
2
3  fix(login): resolve password validation bug
4  Fixes #234
5
6  docs(README): update installation instructions
7
8  refactor(database): optimize query performance
9  Reduce query time from 2s to 200ms by adding index
10
11 test(auth): add integration tests for login flow
12
13 perf(image): implement lazy loading for gallery
14 Improves page load time by 40%
```

Esempi CATTIVI

```
1 update stuff
2 fixed bug
3 WIP
4 asdfsdf
5 Final version
```

```
Final version 2
Really final version
```

Perché sono cattivi?

- Non descrittivi
- Non spiegano il "perché"
- Rendono impossibile capire la cronologia
- Complicano bisect, revert, review

8.1.4 Subject Line: Le 7 Regole

1. **Separa subject da body con riga vuota**
2. **Limita subject a 50 caratteri**
3. **Inizia con lettera maiuscola**
4. **Non terminare con punto**
5. **Usa imperativo** ("Add feature" non "Added feature")
6. **Spiega COSA e PERCHÉ, non COME**
7. **Body: max 72 caratteri per riga**

```
# BUONO
Add user authentication system

# CATTIVO
added user authentication system.
```

8.1.5 Body: Quando e Come

Il body è opzionale per commit semplici, ma fondamentale per modifiche complesse.

```
refactor(database): migrate from MongoDB to PostgreSQL

The application has grown beyond MongoDB's capabilities for our
use case. PostgreSQL provides:
- Better ACID compliance for financial transactions
- Superior JOIN performance for complex queries
- More mature ecosystem for data analytics

Migration steps:
1. Set up PostgreSQL database
2. Create schema matching current data model
3. Migrate data using custom script
4. Update ORM configuration
5. Deploy new version with backward compatibility

This is a breaking change requiring infrastructure updates.
See MIGRATION.md for deployment guide.

Refs #456, #457, #458
```

8.2 Atomic Commits

8.2.1 Cos'è un Commit Atomico

Un commit è atomico quando contiene **un singolo cambiamento logico** che:

- È autocontenuto e completo
- Compila senza errori
- Non rompe test esistenti
- Può essere revertito indipendentemente

8.2.2 Esempio: NON Atomico

Commit Troppo Grande

```
commit abc123
Author: Developer
Date: ...

Add user authentication, fix login bug, update dependencies,
refactor database queries, add email validation

Modified files:
src/auth/login.js
src/auth/register.js
src/auth/middleware.js
src/database/queries.js
src/database/connection.js
src/validation/email.js
package.json
package-lock.json
README.md
```

Problemi:

- Troppi cambiamenti non correlati
- Impossibile revert solo una parte
- Code review difficile
- Bisect non efficace

8.2.3 Esempio: Atomico Corretto

Commit Atomici Separati

```
# Commit 1
feat(auth): add JWT middleware
Modified: src/auth/middleware.js

# Commit 2
feat(auth): implement login endpoint
Modified: src/auth/login.js

```

```
# Commit 3
feat(auth): implement registration endpoint
Modified: src/auth/register.js

# Commit 4
feat(validation): add email validation helper
Modified: src/validation/email.js

# Commit 5
refactor(database): optimize user queries
Modified: src/database/queries.js

# Commit 6
fix(auth): resolve password hashing bug
Modified: src/auth/login.js

# Commit 7
chore(deps): update security dependencies
Modified: package.json, package-lock.json

# Commit 8
docs(README): add authentication setup guide
Modified: README.md
```

8.2.4 Staging Parziale per Commit Atomici

```
# Hai modificato 3 file per 2 feature diverse
git status
# modified: feature-a.js
# modified: feature-b.js
# modified: utils.js (usato da entrambi)

# Commit solo feature A
git add feature-a.js
git add -p utils.js  # seleziona solo modifiche per A
git commit -m "feat: implement feature A"

# Commit feature B
git add feature-b.js utils.js
git commit -m "feat: implement feature B"
```

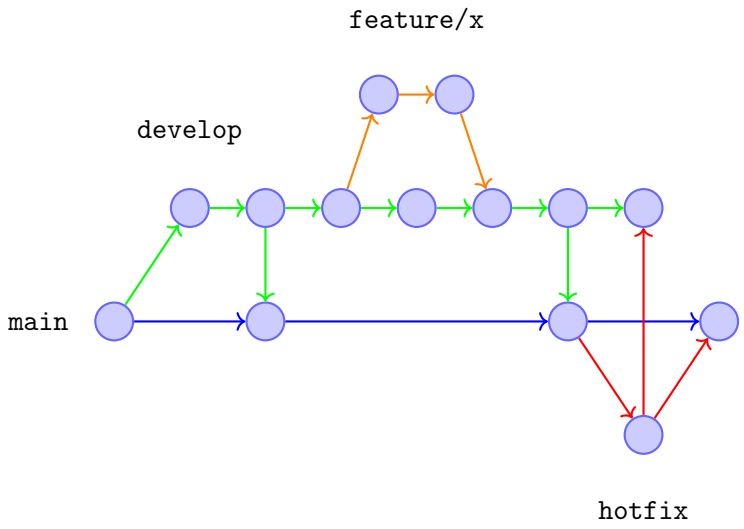
8.2.5 Interactive Staging

```
# Staging interattivo per file grande
git add -p myfile.js

# Git mostrerà ogni "hunk" (blocco modifiche)
# y = stage this hunk
# n = don't stage
# s = split into smaller hunks
# e = edit hunk manually
# q = quit
```

8.3 Branching Strategies

8.3.1 Git Flow

Git Flow è una branching strategy strutturata per progetti con release cicliche.



Branch principali:

`main` Produzione, solo release stabili

`develop` Integrazione, development attivo

Branch temporanei:

`feature/*` Nuove funzionalità (da develop)

`release/*` Preparazione release (da develop)

`hotfix/*` Fix urgenti produzione (da main)

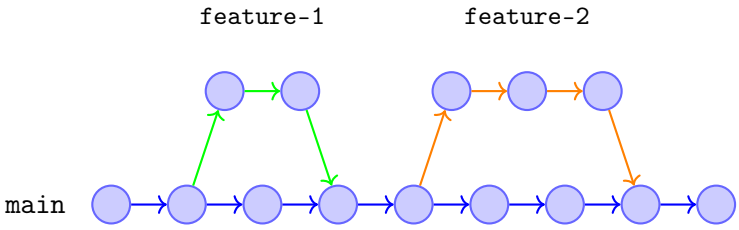
8.3.2 Git Flow: Comandi

```
# Setup repository
git checkout -b develop main

# Feature workflow
git checkout -b feature/user-profile develop
# ... lavoro ...
git checkout develop
git merge --no-ff feature/user-profile
git branch -d feature/user-profile

# Release workflow
git checkout -b release/1.2.0 develop
# ... fix bug release ...
git checkout main
git merge --no-ff release/1.2.0
git tag -a v1.2.0
git checkout develop
git merge --no-ff release/1.2.0
git branch -d release/1.2.0
```

```
20
21 # Hotfix workflow
22 git checkout -b hotfix/1.2.1 main
23 # ... fix bug ...
24 git checkout main
25 git merge --no-ff hotfix/1.2.1
26 git tag -a v1.2.1
27 git checkout develop
28 git merge --no-ff hotfix/1.2.1
29 git branch -d hotfix/1.2.1
```

8.3.3 GitHub Flow

GitHub Flow è più semplice: ottimo per continuous deployment.



Regole:

1. `main` è sempre deployable
2. Crea branch descrittivo da `main`
3. Commit regolari e push al remote
4. Apri Pull Request quando pronto
5. Code review e discussione
6. Deploy da branch per testing
7. Merge in `main` dopo approval
8. Deploy immediatamente da `main`

```
1  # GitHub Flow workflow
2  git checkout main
3  git pull origin main
4  git checkout -b feature/add-search
5
6  # ... sviluppo ...
7  git push -u origin feature/add-search
8
9  # Apri Pull Request su GitHub
10
11 # Dopo approval e CI/CD green
12 git checkout main
13 git merge feature/add-search
14 git push origin main
15
16 # Auto-deploy to production
17 git branch -d feature/add-search
18 git push origin --delete feature/add-search
```

8.3.4 Trunk-Based Development

Trunk-based è estremamente semplice: sviluppo diretto su main (o trunk).

Trunk-Based Development

Caratteristiche:

- Branch di breve durata (< 1 giorno)
- Commit frequenti su main
- Feature flags per funzionalità incomplete
- CI/CD obbligatorio
- Test automatici robusti

Pro:

- Integrazione continua reale
- Meno merge conflicts
- Feedback veloce

Contro:

- Richiede disciplina del team
- Necessita ottimi test automatici
- Feature flags possono complicare codice

8.3.5 Quale Strategia Scegliere?

Scenario	Strategia Consigliata
Release pianificate, versioning	Git Flow
Continuous deployment, startup	GitHub Flow
Team piccolo, alta frequenza deploy	Trunk-Based
Open source, molti contributor	GitHub Flow / Forking
Enterprise, processi rigidi	Git Flow

8.4 Merge vs Rebase

8.4.1 Quando Usare Merge

```
1 git checkout main
2 git merge feature-branch
```

Pro:

- Preserva cronologia completa
- Non riscrive storia
- Sicuro per branch condivisi
- Mostra merge conflicts esplicitamente

Contro:

- Crea merge commits extra
- Cronologia può essere confusa

8.4.2 Quando Usare Rebase

```
git checkout feature-branch
git rebase main
```

Pro:

- Cronologia lineare e pulita
- Nessun merge commit
- Più facile da leggere

Contro:

- Riscrive cronologia
- Pericoloso su branch condivisi
- Può perdere contesto merge

8.4.3 Golden Rule of Rebase

REGOLA D'ORO

Mai fare rebase di commit già pushati e condivisi con altri!
Rebase riscrive cronologia. Se altri hanno basato lavoro sui tuoi commit, rebase crea caos.
Rebase è OK:

- Su branch locali non pushati
- Su feature branch usati solo da te
- Prima di aprire Pull Request (pulizia)

Rebase è PERICOLOSO:

- Su main/develop
- Su branch condivisi con team
- Su commit già pushati in PR

8.4.4 Workflow Raccomandato

```
# Lavorando su feature branch
git checkout feature-branch

# Aggiorna con main (rebase locale)
git fetch origin
git rebase origin/main

# Se ci sono conflitti, risolvili
```

```
git rebase --continue

# Quando pronto per PR, pulisci cronologia
git rebase -i HEAD~5  # squash commit WIP

# Push (prima volta o force dopo rebase)
git push -f origin feature-branch

# Apri PR

# Quando merge PR, usa merge (non rebase su main!)
# Via GitHub: "Merge pull request" oppure "Squash and merge"
```

8.5 .gitignore Best Practices

8.5.1 File .gitignore Strutturato

```
# .gitignore

# ========================================
# Dependencies
# ========================================
node_modules/
vendor/
bower_components/

# ========================================
# Build Output
# ========================================
dist/
build/
*.min.js
*.min.css

# ========================================
# Environment Variables
# ========================================
.env
.env.local
.env.*.local
config/secrets.yml

# ========================================
# IDE / Editor
# ========================================
.vscode/
.idea/
*.swp
*.swo
*~
.DS_Store

# ========================================
# Logs
# ========================================
*.log
npm-debug.log*
```

```
logs/

# ========================================
# Testing
# ========================================
coverage/
.nyc_output/
*.test.db

# ========================================
# OS Generated
# ========================================
Thumbs.db
.DS_Store
desktop.ini

# ========================================
# Temporary Files
# ========================================
tmp/
temp/
*.tmp
```

8.5.2 Pattern Avanzati

```
# Ignora tutto in cartella tranne file specifico
build/*
!build/version.txt

# Ignora solo in root (non in subdirectories)
/TODO.txt

# Ignora pattern ricorsivamente
**/*.log

# Negazione (non ignorare)
*.log
!important.log

# Ignora cartella ma traccia contenuto
cache/.gitkeep
# (crea file vuoto .gitkeep per tracciare cartella vuota)
```

8.5.3 .gitignore Globale

```
# Configura gitignore globale per tutti i repository
git config --global core.excludesfile ~/.gitignore_global

# ~/.gitignore_global
.DS_Store
.vscode/
.idea/
*.swp
```

8.6 Security Best Practices

8.6.1 Mai Committare Secrets

PERICOLO: Secrets in Repository

MAI committare:

- Password, API keys, token
- Chiavi private SSH/SSL
- Database credentials
- OAuth secrets
- File .env con secrets

Anche se cancelli il commit dopo, rimane nella cronologia Git!

8.6.2 Uso di .env e Secrets

```
# .env (MAI committare)
DATABASE_URL=postgresql://user:password@localhost/db
API_KEY=super-secret-key-123
JWT_SECRET=my-jwt-secret

# .env.example (COMMITTARE questo)
DATABASE_URL=postgresql://user:password@localhost/db_name
API_KEY=your-api-key-here
JWT_SECRET=your-jwt-secret-here
```

8.6.3 Rimuovere Secrets dalla Cronologia

Se hai committato secrets, rimuovili SUBITO:

```
# Rimuovi file da tutta la cronologia
git filter-branch --force --index-filter \
  "git rm --cached --ignore-unmatch path/to/secrets.env" \
  --prune-empty --tag-name-filter cat -- --all

# Oppure usa tool più moderno
git filter-repo --path secrets.env --invert-paths

# Force push (ATTENZIONE!)
git push origin --force --all

# IMPORTANTE: Revoca le credenziali esposte!
# Cambia password, rigenera API keys, etc.
```

8.7 Esercizi

8.7.1 Esercizio 1: Commit Message Quality

Riscrivi questi commit messages seguendo Conventional Commits:

1. "fixed stuff"

2. "update"
3. "added login and fixed bugs and updated readme"

8.7.2 Esercizio 2: Atomic Commits

Hai un file con 3 funzionalità mescolate. Usa `git add -p` per creare 3 commit atomici separati.

8.7.3 Esercizio 3: Git Flow

Implementa completo Git Flow workflow:

1. Setup main e develop
2. Crea feature branch e merge in develop
3. Crea release branch
4. Merge release in main e develop
5. Simula hotfix

8.7.4 Esercizio 4: Rebase Practice

1. Crea branch con 5 commit "WIP"
2. Usa `git rebase -i` per squash in 2 commit significativi
3. Modifica commit messages

8.7.5 Esercizio 5: .gitignore

Crea .gitignore per progetto Node.js che:

- Ignora node_modules
- Ignora .env ma traccia .env.example
- Ignora build/ tranne build/version.txt

Risorse

- Conventional Commits: https://www.conventionalcommits.org
- gitignore.io: https://www.toptal.com/developers/gitignore
- Git Flow Cheatsheet: https://danielkummer.github.io/git-flow-cheatsheet

Capitolo 9

Troubleshooting e Recovery

Introduzione

Errori capitano. Questo capitolo è una guida pratica per risolvere i problemi più comuni in Git: annullare modifiche, recuperare commit persi, risolvere merge conflicts, gestire repository corrotti, e molto altro. Ogni sezione presenta problema, diagnosi e soluzione.

Obiettivi di apprendimento

- Annullare modifiche a vari livelli (working directory, staging, commit)
- Risolvere merge conflicts efficacemente
- Recuperare da detached HEAD state
- Recuperare commit e branch cancellati
- Gestire errori di force push
- Risolvere problemi di autenticazione
- Diagnosticare e risolvere repository corrotti
- Rimuovere file grandi dalla cronologia

9.1 Undo Changes: Livelli di Annullamento

9.1.1 Livello 1: Working Directory

Problema: Hai modificato file ma non hai ancora fatto `git add`.

```
# Situazione
git status
# modified: file.js (not staged)

# Soluzione: Scarta modifiche singolo file
git checkout -- file.js
# oppure (Git 2.23+)
git restore file.js

# Soluzione: Scarta tutte le modifiche
git checkout -- .
# oppure
git restore .
```

Attenzione

`git checkout -` e `git restore` distruggono modifiche non salvate. Non c'è modo di recuperarle!

9.1.2 Livello 2: Staging Area

Problema: Hai fatto `git add` ma non ancora `git commit`.

```
# Situazione
git status
# Changes to be committed:
#   modified: file.js

# Soluzione: Rimuovi da staging (mantieni modifiche)
git reset HEAD file.js
# oppure (Git 2.23+)
git restore --staged file.js

# Soluzione: Rimuovi tutto da staging
git reset HEAD
```

9.1.3 Livello 3: Ultimo Commit (Non Pushato)

Problema: Hai fatto commit ma vuoi modificarlo.

```
# Soluzione 1: Modifica ultimo commit
# (aggiunge modifiche al commit precedente)
git add file.js
git commit --amend

# Soluzione 2: Modifica solo messaggio commit
git commit --amend -m "New commit message"

# Soluzione 3: Annulla commit, mantieni modifiche in staging
git reset --soft HEAD~1

# Soluzione 4: Annulla commit, mantieni modifiche unstaged
git reset HEAD~1
# oppure
git reset --mixed HEAD~1

# Soluzione 5: Annulla commit, DISTRUGGI modifiche
git reset --hard HEAD~1
```

9.1.4 Livello 4: Commit Vecchi o Pushati

Problema: Hai pushato commit che vuoi annullare.

```
# Soluzione: Crea commit che annulla modifiche (sicuro)
git revert <commit-hash>

# Esempio
git log --oneline
# abc123 Add buggy feature
# def456 Previous commit

```

```
# Annulla abc123
git revert abc123
# Crea nuovo commit che annulla modifiche di abc123

git push origin main
# Sicuro: non riscrive cronologia
```

Reset vs Revert

- **Reset**: Cancella commit dalla cronologia (pericoloso se pushato)
- **Revert**: Crea nuovo commit che annulla modifiche (sicuro sempre)

Regola: Se il commit è pushato, usa SEMPRE `git revert`.

9.1.5 Tabella Riepilogativa

Scenario	Comando	Modif. Perse?
File modificato, not staged	`git restore file.js`	SÌ
File in staging	`git restore -staged file.js`	NO
Modifica ultimo commit	`git commit -amend`	NO
Annulla ultimo commit	`git reset -soft HEAD~1`	NO
Distruggi ultimo commit	`git reset -hard HEAD~1`	SÌ
Annulla commit pushato	`git revert <hash>`	NO

9.2 Merge Conflicts

9.2.1 Anatomia di un Merge Conflict

```
# Tentativo merge
git merge feature-branch

# Output
Auto-merging file.js
CONFLICT (content): Merge conflict in file.js
Automatic merge failed; fix conflicts and then commit the result.
```

File con conflitto:

```
function calculateTotal(items) {
<<<<<<< HEAD
    // Implementazione main branch
    return items.reduce((sum, item) => sum + item.price, 0);
=======
    // Implementazione feature branch
    let total = 0;
    for (let item of items) {
        total += item.price;
    }
    return total;
>>>>>>> feature-branch
}
```

9.2.2 Risolvere Conflicts Manualmente

```
# 1. Identifica file con conflitti
git status
# Unmerged paths:
#   both modified: file.js

# 2. Apri file e risolvi
# Rimuovi marker (<<<<, ====, >>>>)
# Scegli versione da mantenere o combina

# Dopo risoluzione:
function calculateTotal(items) {
    return items.reduce((sum, item) => sum + item.price, 0);
}

# 3. Marca come risolto
git add file.js

# 4. Completa merge
git commit -m "Merge feature-branch, resolve conflicts"
```

9.2.3 Strumenti Merge

```
# Usa merge tool grafico
git mergetool

# Configura merge tool preferito
git config --global merge.tool vimdiff
# oppure meld, kdiff3, p4merge, etc.

# Accetta versione specifica per tutti i conflitti
# Accetta versione "ours" (branch corrente)
git checkout --ours file.js

# Accetta versione "theirs" (branch in merge)
git checkout --theirs file.js
```

9.2.4 Abbandonare Merge

```
# Se merge è troppo complicato, abbandona
git merge --abort

# Torna allo stato prima del merge
git status
# On branch main
# nothing to commit, working tree clean
```

9.2.5 Strategie per Evitare Conflicts

Prevenire è Meglio che Curare

1. **Pull frequente**: Aggiorna spesso da main
2. **Commit piccoli**: Più facili da mergeare
3. **Comunicazione**: Coordina modifiche a stessi file
4. **Feature branch brevi**: Merge veloce = meno conflitti
5. **Code review**: Identifica potenziali conflitti in PR

9.3 Detached HEAD State

9.3.1 Cos'è Detached HEAD

```
# Checkout di commit specifico
git checkout abc123

# Output
You are in 'detached HEAD' state...
```

In detached HEAD, non sei su un branch. I commit fatti non appartengono a nessun branch e possono essere persi.

9.3.2 Come Uscire da Detached HEAD

```
# Situazione: sei in detached HEAD e hai fatto commit
git log --oneline
# xyz789 My commit in detached HEAD
# abc123 Previous commit

# Soluzione 1: Crea branch dai commit
git branch recovery-branch
git checkout recovery-branch
# Ora i commit sono al sicuro

# Soluzione 2: Torna a branch senza salvare
git checkout main
# Warning: commit xyz789 perso (recuperabile con reflog)

# Soluzione 3: Merge commit in branch esistente
git checkout main
git merge xyz789
```

9.3.3 Quando Accade Detached HEAD

```
# Checkout commit specifico
git checkout abc123

# Checkout tag
git checkout v1.0.0

```

```
# Checkout file da altro commit
git checkout main~3 -- file.js
```

Evitare Detached HEAD

Se vuoi lavorare su commit vecchio, crea sempre branch:

```
git checkout -b fix-old-version abc123
```

9.4 Recuperare Commit e Branch Cancellati

9.4.1 Recupero con Reflog

```
# Scenario: hai cancellato branch per errore
git branch -D feature-important
# Deleted branch feature-important (was abc123)

# OH NO! Era importante!

# Soluzione: usa reflog
git reflog
# abc123 HEAD@{0}: branch: deleted feature-important
# def456 HEAD@{1}: commit: Last commit on feature
# ...

# Recupera creando branch al commit
git branch feature-important abc123
# oppure
git checkout -b feature-important abc123

# Branch recuperato!
```

9.4.2 Recupero Commit Dopo Reset Hard

```
# Disastro: reset hard per errore
git reset --hard HEAD~5
# Persi 5 commit!

# Reflog to the rescue
git reflog
# xyz789 HEAD@{1}: commit: Important work 5
# ...

# Recupera
git reset --hard xyz789
# Oppure crea branch
git branch recovered xyz789
```

9.4.3 Quando Reflog Non Funziona

```
# Reflog è locale e temporaneo (default 90 giorni)

# Se hai fatto git gc (garbage collection)
```

```
git reflog expire --expire=now --all
git gc --prune=now
# I commit unreachable sono persi

# Se repository è clonato recentemente
# Reflog del repo originale non viene clonato
```

9.5 Force Push Gone Wrong

9.5.1 Scenario

```
# Developer A fa force push
git push -f origin main

# Developer B (che aveva commit locali) tenta push
git push origin main
# Error: Updates were rejected
# hint: 'git pull' before pushing
```

9.5.2 Soluzione per Developer B

```
# Opzione 1: Salva lavoro locale
git stash
git pull origin main
git stash pop
# Risolvi eventuali conflitti

# Opzione 2: Rebase su nuovo main
git fetch origin
git rebase origin/main
# Risolvi conflitti
git push origin main

# Opzione 3: Se lavoro è su branch
git fetch origin
git checkout feature-branch
git rebase origin/main
```

9.5.3 Recupero dopo Force Push

```
# Se qualcuno ha fatto force push su main
# e hai perso commit

# 1. Verifica reflog
git reflog origin/main
# Trova commit prima di force push

# 2. Recupera
git reset --hard origin/main@{1}

# 3. Crea backup branch
git branch backup-before-force-push

# 4. Comunica con team per recovery
```

Prevenire Force Push Disasters

Best Practices:

1. Proteggi branch main/develop (GitHub: branch protection rules)
2. Force push SOLO su feature branch personali
3. Usa `-force-with-lease` invece di `-f`
4. Comunica prima di force push su branch condivisi

```
# --force-with-lease fallisce se altri hanno pushato
git push --force-with-lease origin feature-branch
# Più sicuro di -f
```

9.6 Repository Corrotto

9.6.1 Sintomi

```
# Errori tipici
error: object file is empty
error: bad object
fatal: loose object is corrupt
error: unable to read sha1 file
```

9.6.2 Diagnosi

```
# Verifica integrità repository
git fsck --full

# Output se ci sono problemi:
# error: object file is empty
# error: sha1 mismatch
# missing blob abc123
```

9.6.3 Recovery

```
# Soluzione 1: Recupero da remote
git fetch origin
git reset --hard origin/main

# Soluzione 2: Clone fresco
cd ..
mv myproject myproject-backup
git clone <remote-url> myproject
# Copia eventuali modifiche locali da backup

# Soluzione 3: Ricostruzione oggetti
# Se hai backup del .git
rm -rf .git/objects/*
git fetch origin
git reset --hard origin/main

```

```
# Soluzione 4: Pulizia repository
git gc --aggressive --prune=now
git fsck --full
```

9.6.4 Prevenzione

Evitare Corruzione

- Push regolarmente a remote (backup automatico)
- Non killare git durante operazioni
- Usa file system affidabili
- Backup periodici di .git/
- Evita modifiche manuali in .git/

9.7 File Grandi Committati per Errore

9.7.1 Problema

```
# Hai committato file grande
git add large-file.zip  # 500 MB
git commit -m "Add resources"
git push origin main

# Error: file exceeds GitHub's limit of 100 MB
```

9.7.2 Soluzione: Rimuovi da Ultimo Commit

```
# Se NON hai pushato
git reset --soft HEAD~1
git restore --staged large-file.zip
git commit -m "Add resources"
```

9.7.3 Soluzione: Rimuovi da Cronologia

```
# Se hai pushato o commit è vecchio
# Usa git filter-repo (tool moderno)

# Installa git filter-repo
# pip3 install git-filter-repo

# Rimuovi file da tutta la cronologia
git filter-repo --path large-file.zip --invert-paths

# Force push (attenzione!)
git push origin --force --all

# Cleanup locale
git gc --aggressive --prune=now
```

9.7.4 Alternativa: BFG Repo-Cleaner

```
# Scarica BFG Repo-Cleaner

# Rimuovi file oltre 100MB
java -jar bfg.jar --strip-blobs-bigger-than 100M myrepo.git

# Cleanup
cd myrepo
git reflog expire --expire=now --all
git gc --prune=now --aggressive

# Force push
git push origin --force --all
```

9.7.5 Prevenzione: Git LFS

```
# Installa Git Large File Storage
git lfs install

# Traccia file grandi
git lfs track "*.zip"
git lfs track "*.mp4"
git lfs track "*.psd"

# Aggiungi .gitattributes
git add .gitattributes

# Ora i file grandi sono gestiti da LFS
git add large-file.zip
git commit -m "Add large file via LFS"
git push origin main
```

9.8 Commit su Branch Sbagliato

9.8.1 Problema

```
# Hai committato su main invece di feature branch
git status
# On branch main

git log --oneline
# abc123 Feature work (OOPS, doveva essere su feature branch)
```

9.8.2 Soluzione 1: Sposta Commit su Nuovo Branch

```
# Crea branch dal commit corrente
git branch feature-branch

# Torna indietro main di 1 commit
git reset --hard HEAD~1

# Vai al branch con il commit
git checkout feature-branch
```

```
9
10 # Ora il commit è su feature-branch, non su main
```

9.8.3 Soluzione 2: Sposta su Branch Esistente

```
1  # Il commit è su main, vuoi su existing-branch
2  git log --oneline
3  # abc123 My commit (on main)
4
5  # Vai al branch corretto
6  git checkout existing-branch
7
8  # Cherry-pick il commit
9  git cherry-pick abc123
10
11 # Torna a main e rimuovi commit
12 git checkout main
13 git reset --hard HEAD~1
```

9.9 Problemi di Autenticazione

9.9.1 HTTPS: Username/Password

```
1  # Errore tipico
2  git push origin main
3  # remote: Support for password authentication was removed
4  # fatal: Authentication failed
5
6  # Soluzione: Usa Personal Access Token (GitHub)
7  # 1. GitHub -> Settings -> Developer settings -> Personal access tokens
8  # 2. Generate new token (classic)
9  # 3. Usa token invece di password
10
11 # Cache credentials
12 git config --global credential.helper cache
13 # oppure permanent
14 git config --global credential.helper store
15
16 # Push con token
17 git push https://TOKEN@github.com/user/repo.git
```

9.9.2 SSH: Permission Denied

```
1  # Errore
2  git push origin main
3  # Permission denied (publickey)
4
5  # Diagnosi
6  ssh -T git@github.com
7  # Permission denied (publickey)
8
9  # Soluzione
10 # 1. Verifica chiave SSH esiste
11 ls -la ~/.ssh/
```

```
# Cerca id_ed25519, id_rsa

# 2. Se non esiste, genera
ssh-keygen -t ed25519 -C "email@example.com"

# 3. Aggiungi a ssh-agent
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519

# 4. Copia chiave pubblica
cat ~/.ssh/id_ed25519.pub

# 5. Aggiungi a GitHub
# Settings -> SSH and GPG keys -> New SSH key

# 6. Testa
ssh -T git@github.com
# Hi username! You've successfully authenticated
```

9.10 Performance Issues

9.10.1 Repository Lento

```
# Diagnosi: dimensione repository
git count-objects -vH

# Output:
# size: 1.5 GB
# size-pack: 800 MB

# Soluzione 1: Garbage collection
git gc --aggressive --prune=now

# Soluzione 2: Ottimizza pack files
git repack -a -d --depth=250 --window=250

# Soluzione 3: Shallow clone (per clonare più veloce)
git clone --depth 1 <url>

# Soluzione 4: Sparse checkout (solo parte del repo)
git clone --filter=blob:none --sparse <url>
git sparse-checkout set folder1 folder2
```

9.11 Cheat Sheet Troubleshooting

Quick Reference

```
# Undo modifiche non staged
git restore <file>

# Undo staging
git restore --staged <file>

# Modifica ultimo commit
```

```
git commit --amend

# Annulla ultimo commit (keep changes)
git reset --soft HEAD~1

# Distruggi ultimo commit
git reset --hard HEAD~1

# Annulla commit pushato
git revert <hash>

# Abbandona merge
git merge --abort

# Recupera commit perso
git reflog

# Verifica repository
git fsck --full

# Cleanup repository
git gc --aggressive --prune=now
```

9.12 Esercizi

9.12.1 Esercizio 1: Undo Practice

Simula e risolvi:

1. Modifica file, poi annulla modifiche
2. Aggiungi a staging, poi rimuovi
3. Fai commit, poi annulla mantenendo modifiche
4. Fai commit pushato, poi annulla con revert

9.12.2 Esercizio 2: Merge Conflict

1. Crea due branch che modificano stesso file
2. Merge e crea conflitto intenzionale
3. Risolvi manualmente
4. Tenta merge –abort e riprova

9.12.3 Esercizio 3: Recovery

1. Crea commit
2. Cancella branch con `git branch -D`
3. Recupera con reflog
4. Verifica che commit sia recuperato

9.12.4 Esercizio 4: Wrong Branch

1. Fai commit su main per errore
2. Sposta commit su nuovo branch
3. Verifica che main sia pulito

Nota Finale

La maggior parte dei problemi Git sono risolvibili. Ricorda:

- Reflog salva quasi tutto (per 90 giorni)
- Non panico: raramente perdi dati definitivamente
- Fai backup (push regolare è backup)
- Impara dai errori: capire cosa è andato storto previene ripetizioni

Capitolo 10

Continuous Integration e Continuous Deployment

Introduzione

Continuous Integration (CI) e Continuous Deployment (CD) sono pratiche fondamentali dello sviluppo moderno. CI automatizza testing e verifica del codice ad ogni commit. CD automatizza il deployment in produzione. Questo capitolo introduce i concetti e implementa pipeline CI/CD con GitHub Actions.

Obiettivi di apprendimento

- Comprendere concetti di CI/CD e benefici
- Configurare pipeline GitHub Actions
- Implementare testing automatico
- Automatizzare build e deployment
- Gestire secrets e environment variables
- Usare matrix builds per multi-platform testing
- Implementare deployment strategies
- Monitorare e debuggare pipeline

10.1 Cos'è CI/CD

10.1.1 Continuous Integration (CI)

Definizione: Pratica di integrare codice nel repository principale frequentemente (più volte al giorno), con verifica automatica tramite build e test.

Workflow tradizionale (senza CI):

1. Developer scrive codice per giorni/settimane
2. Commit e push
3. Integrazione manuale con codice di altri
4. Scoperta di bug e conflitti

5. Fix lunghi e dolorosi

Workflow con CI:

1. Developer scrive codice
2. Commit e push frequenti (più volte al giorno)
3. CI server automaticamente: compila, testa, verifica
4. Feedback immediato su errori
5. Fix veloce (codice fresco in memoria)

10.1.2 Continuous Deployment (CD)

Definizione: Estensione di CI che automatizza deployment in produzione dopo successful build/test.

Varianti:

Continuous Delivery Codice sempre pronto per deploy, ma deploy manuale

Continuous Deployment Deploy automatico in produzione ad ogni commit

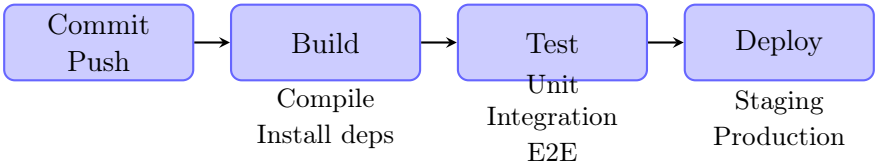
10.1.3 Benefici CI/CD

Vantaggi

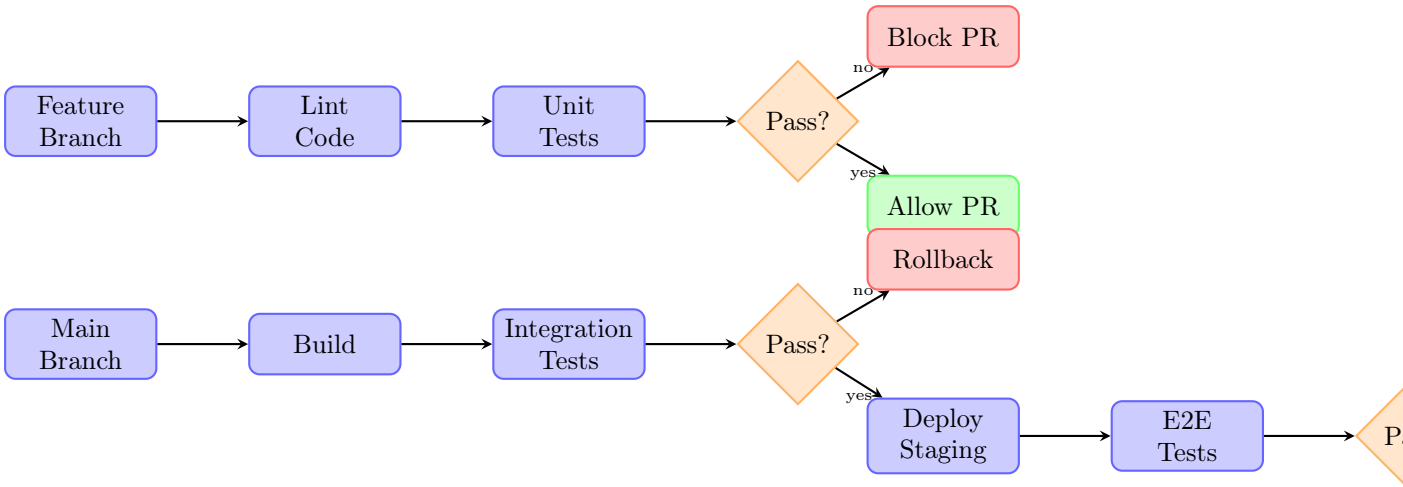
- **Feedback rapido**: Scopri errori subito, non dopo giorni
- **Qualità codice**: Test automatici prevengono regressioni
- **Deployment veloce**: Da commit a produzione in minuti
- **Riduzione rischio**: Deploy piccoli e frequenti vs grandi release
- **Automazione**: Meno errori umani
- **Confidence**: Deploy sicuri grazie a test completi

10.2 Pipeline CI/CD: Concetti

10.2.1 Anatomia di una Pipeline



10.2.2 Pipeline Completa con Branch



10.2.3 Terminologia

Pipeline Insieme completo di stage/job per CI/CD

Job Unità di lavoro (es: "run tests", "deploy")

Step Singola azione in un job (es: "npm install")

Runner Server che esegue i job

Artifact Output di build (binari, report, etc.)

Environment Target deployment (dev, staging, prod)

10.3 GitHub Actions: Primi Passi

10.3.1 Struttura File Workflow

```
# Directory structure
.github/
  workflows/
    ci.yml           # Workflow CI
    deploy.yml       # Workflow deployment
    release.yml      # Workflow release
```

10.3.2 Workflow Minimo

File `.github/workflows/hello.yml`:

```
name: Hello World

on: [push]

jobs:
  greet:
    runs-on: ubuntu-latest
```

```
    steps:
      - name: Say hello
        run: echo "Hello, World!"
```

Componenti:

- `name`: Nome workflow (opzionale)
- `on`: Eventi che triggherano workflow
- `jobs`: Lista job da eseguire
- `runs-on`: Sistema operativo runner
- `steps`: Lista step del job

10.3.3 Workflow CI Completo: Node.js

File `.github/workflows/ci.yml`:

```
name: CI Pipeline

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  lint:
    name: Lint Code
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run ESLint
        run: npm run lint

  test:
    name: Run Tests
    runs-on: ubuntu-latest
    needs: lint
    steps:
      - uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
```

```
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Run unit tests
        run: npm test

      - name: Run integration tests
        run: npm run test:integration

      - name: Generate coverage report
        run: npm run coverage

      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v3
        with:
          files: ./coverage/coverage.xml
          flags: unittests
          fail_ci_if_error: true

  build:
    name: Build Application
    runs-on: ubuntu-latest
    needs: test
    steps:
      - uses: actions/checkout@v3

      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
          cache: 'npm'

      - name: Install dependencies
        run: npm ci

      - name: Build
        run: npm run build

      - name: Upload build artifacts
        uses: actions/upload-artifact@v3
        with:
          name: build-output
          path: dist/
          retention-days: 7
```

10.3.4 Triggers: Eventi che Avviano Workflow

```
# Push su branch specifici
on:
  push:
    branches:
      - main
      - develop
      - 'release/**'

```

```yaml
# Pull request
on:
  pull_request:
    branches: [ main ]
    types: [opened, synchronize, reopened]

# Schedule (cron)
on:
  schedule:
    - cron: '0 2 * * *'  # Ogni giorno alle 2 AM

# Manuale
on:
  workflow_dispatch:
    inputs:
      environment:
        description: 'Environment to deploy'
        required: true
        default: 'staging'

# Multiple eventi
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '0 0 * * 0'  # Domenica mezzanotte
```

10.4 Matrix Builds: Testing Multi-Platform

10.4.1 Matrix Strategy

```yaml
name: Matrix Testing

on: [push, pull_request]

jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node: [14, 16, 18, 20]
        include:
          - os: ubuntu-latest
            node: 20
            experimental: true
        exclude:
          - os: macos-latest
            node: 14

    steps:
      - uses: actions/checkout@v3

      - name: Setup Node.js ${{ matrix.node }}
        uses: actions/setup-node@v3
```

```
        with:
          node-version: ${{ matrix.node }}

      - run: npm ci
      - run: npm test
```

Questo workflow crea **11 job**:

- 3 OS × 4 Node versions = 12 combinazioni
- -1 (exclude: macos + node 14)
- Total: 11 job paralleli

10.4.2 Matrix con Database

```
jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        db: [postgres, mysql, mongodb]
        node: [16, 18]

    services:
      postgres:
        image: postgres:15
        env:
          POSTGRES_PASSWORD: postgres
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5

    steps:
      - uses: actions/checkout@v3
      - name: Setup Node ${{ matrix.node }}
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node }}
      - run: npm ci
      - run: npm test
        env:
          DB_TYPE: ${{ matrix.db }}
```

10.5 Secrets e Environment Variables

10.5.1 Configurare Secrets

```
# GitHub: Repository -> Settings -> Secrets and variables -> Actions

Secrets:
- DATABASE_URL
- API_KEY
- AWS_ACCESS_KEY_ID
```

```
- AWS_SECRET_ACCESS_KEY
- DEPLOY_TOKEN
```

10.5.2 Uso nei Workflow

```yaml
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Deploy to server
        env:
          DATABASE_URL: ${{ secrets.DATABASE_URL }}
          API_KEY: ${{ secrets.API_KEY }}
        run: |
          echo "DATABASE_URL=$DATABASE_URL" > .env
          echo "API_KEY=$API_KEY" >> .env
          ./deploy.sh

      - name: Deploy to AWS
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1
```

10.5.3 Environment Variables

```yaml
# Globale per tutto il workflow
env:
  NODE_ENV: production
  APP_VERSION: 1.2.0

jobs:
  build:
    # Per singolo job
    env:
      BUILD_TYPE: release
    steps:
      - name: Build
        # Per singolo step
        env:
          CUSTOM_VAR: value
        run: npm run build
```

10.6 Deployment Automation

10.6.1 Deploy su Server via SSH

```yaml
name: Deploy to Production

on:
  push:
```

```
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment:
      name: production
      url: https://myapp.com

    steps:
      - uses: actions/checkout@v3

      - name: Build application
        run: |
          npm ci
          npm run build

      - name: Deploy via SSH
        uses: appleboy/ssh-action@master
        with:
          host: ${{ secrets.SERVER_HOST }}
          username: ${{ secrets.SERVER_USER }}
          key: ${{ secrets.SSH_PRIVATE_KEY }}
          script: |
            cd /var/www/myapp
            git pull origin main
            npm ci --production
            npm run build
            pm2 restart myapp
```

10.6.2 Deploy su Heroku

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Deploy to Heroku
        uses: akhileshns/heroku-deploy@v3.12.14
        with:
          heroku_api_key: ${{ secrets.HEROKU_API_KEY }}
          heroku_app_name: "my-app-name"
          heroku_email: "email@example.com"
```

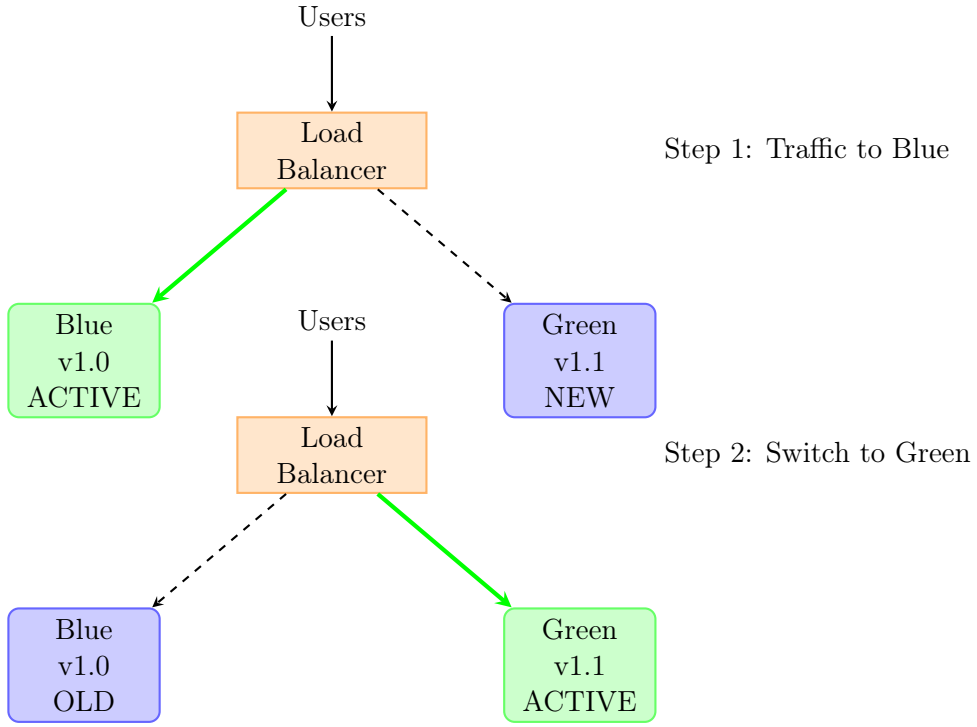
10.6.3 Deploy su AWS S3 (Sito Statico)

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Build static site
        run: |
          npm ci
```

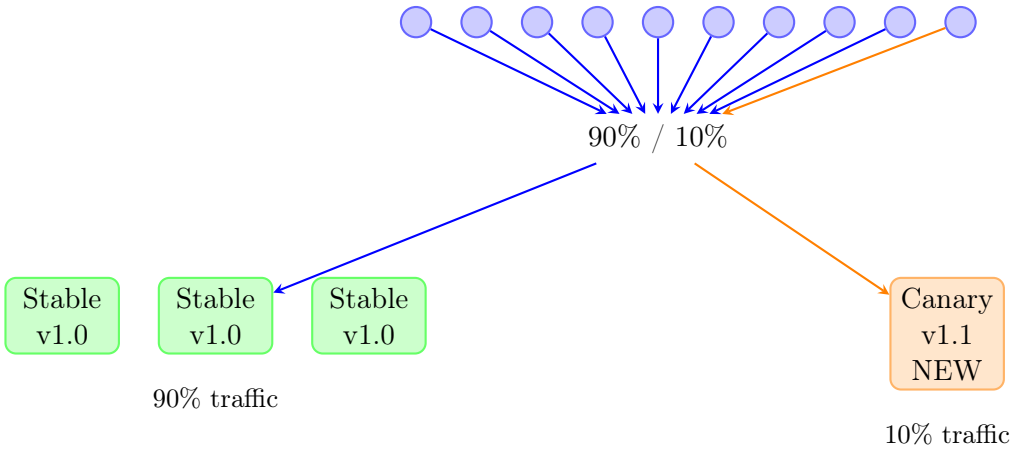
```
10          npm run build
11
12      - name: Configure AWS credentials
13        uses: aws-actions/configure-aws-credentials@v2
14        with:
15          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
16          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
17          aws-region: us-east-1
18
19      - name: Deploy to S3
20        run: |
21          aws s3 sync ./dist s3://my-bucket-name --delete
22
23      - name: Invalidate CloudFront cache
24        run: |
25          aws cloudfront create-invalidation \
26            --distribution-id ${{ secrets.CLOUDFRONT_ID }} \
27            --paths "/*"
```

10.7 Deployment Strategies

10.7.1 Blue-Green Deployment



10.7.2 Canary Deployment



10.8 Best Practices CI/CD

Best Practices

1. **Keep builds fast**: Target $<$ 10 minuti, max 30 minuti
2. **Fail fast**: Test veloci prima, lenti dopo
3. **Parallelize**: Usa matrix e job paralleli
4. **Cache dependencies**: npm ci cache, docker layer cache
5. **Meaningful names**: Job e step descrittivi
6. **Monitor failures**: Alert team su fallimenti
7. **Idempotent scripts**: Script deployment ripetibili
8. **Version everything**: Pin versions (actions, node, etc.)
9. **Test before merge**: Branch protection rules
10. **Automated rollback**: Meccanismi automatici di rollback

10.8.1 Ottimizzazione Pipeline

```yaml
# Cache npm dependencies
- name: Setup Node.js
  uses: actions/setup-node@v3
  with:
    node-version: '18'
    cache: 'npm'

# Cache multipli
- name: Cache dependencies
  uses: actions/cache@v3
  with:
    path: |
      ~/.npm
      ~/.cache
      node_modules
```

```
    key: ${{ runner.os }}-deps-${{ hashFiles('**/package-lock.json') }}

# Parallel jobs
jobs:
  lint:
    runs-on: ubuntu-latest
    # ...

  test:
    runs-on: ubuntu-latest
    # ...

  # lint e test girano in parallelo
```

10.9 Esercizi

10.9.1 Esercizio 1: First Pipeline

1. Crea repository con applicazione Node.js semplice
2. Aggiungi workflow che esegue `npm test` su ogni push
3. Verifica che workflow funzioni
4. Aggiungi badge status al README

10.9.2 Esercizio 2: Multi-Stage Pipeline

1. Crea pipeline con 3 stage: lint, test, build
2. Configura dependencies tra stage (lint → test → build)
3. Aggiungi upload artifacts per build output
4. Testa con commit che rompe lint/test

10.9.3 Esercizio 3: Matrix Testing

1. Configura matrix per Node 16, 18, 20
2. Aggiungi matrix per OS (Ubuntu, Windows, macOS)
3. Osserva job paralleli in esecuzione
4. Analizza tempo totale vs sequenziale

10.9.4 Esercizio 4: Deployment Pipeline

1. Crea workflow deployment su branch main
2. Aggiungi manual approval (environment protection)
3. Implementa deployment su server test
4. Verifica rollback in caso di errore

Risorse

- GitHub Actions Docs: https://docs.github.com/en/actions
- Actions Marketplace: https://github.com/marketplace?type=actions
- Awesome Actions: https://github.com/sdras/awesome-actions
- CI/CD Best Practices: https://www.martinfowler.com/articles/continuousIntegration.html

Appendice A

Appendice: Reference Rapida Comandi

Introduzione

Questa appendice è una reference rapida di tutti i comandi Git più importanti, organizzati per categoria. Ogni comando include sintassi, descrizione breve e opzioni comuni.

A.1 Setup e Configurazione

A.1.1 Configurazione Iniziale

```
# Imposta nome utente globale
git config --global user.name "Your Name"

# Imposta email globale
git config --global user.email "your.email@example.com"

# Imposta editor di default
git config --global core.editor "vim"
git config --global core.editor "code --wait"  # VS Code

# Visualizza configurazione
git config --list
git config --list --show-origin  # mostra file sorgente

# Visualizza configurazione specifica
git config user.name
git config user.email

# Configurazione locale (solo repository corrente)
git config --local user.name "Work Name"

# Rimuovi configurazione
git config --global --unset user.name
```

A.1.2 Alias Utili

```
# Alias comuni
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
```

```
git config --global alias.unstage 'reset HEAD --'

# Alias avanzati
git config --global alias.last 'log -1 HEAD'
git config --global alias.visual 'log --oneline --graph --all --decorate'
git config --global alias.aliases 'config --get-regexp alias'

# Uso
git st            # invece di git status
git visual        # log grafico
```

A.1.3 Configurazioni Utili

```
# Colori in output
git config --global color.ui auto

# Cache credentials (HTTPS)
git config --global credential.helper cache
git config --global credential.helper 'cache --timeout=3600'

# Default branch name
git config --global init.defaultBranch main

# Line endings
git config --global core.autocrlf true    # Windows
git config --global core.autocrlf input   # Mac/Linux

# gitignore globale
git config --global core.excludesfile ~/.gitignore_global
```

A.2 Repository: Creazione e Clonazione

```
# Inizializza repository locale
git init
git init my-project  # crea directory e inizializza

# Clona repository remoto
git clone <url>
git clone <url> <directory-name>
git clone -b <branch> <url>  # clona branch specifico

# Clone shallow (solo ultimo commit)
git clone --depth 1 <url>

# Clone con submodules
git clone --recursive <url>
```

A.3 Status e Informazioni

```
# Status working directory
git status
git status -s          # formato breve
```

```
git status -sb          # breve con branch info

# Visualizza modifiche
git diff                # working directory vs staging
git diff --staged       # staging vs ultimo commit
git diff HEAD           # working directory vs ultimo commit
git diff <branch1> <branch2>  # confronta branch

# Log commits
git log
git log --oneline       # una riga per commit
git log --graph         # grafico ASCII
git log --all           # tutti i branch
git log -n 5            # ultimi 5 commit
git log --since="2 weeks ago"
git log --author="Name"
git log --grep="keyword"  # cerca in commit messages
git log -- <file>       # cronologia file specifico

# Log avanzato
git log --oneline --graph --all --decorate
git log --stat          # mostra file modificati
git log -p              # mostra patch (diff)
git log --follow <file>  # segue rename

# Mostra commit specifico
git show <commit-hash>
git show HEAD
git show HEAD~3         # 3 commit indietro
```

A.4 Staging e Commit

```
# Aggiungi file a staging
git add <file>
git add .               # tutti i file nella directory
git add -A              # tutti i file nel repository
git add *.js            # pattern
git add -p              # interattivo (per hunk)

# Rimuovi da staging
git reset HEAD <file>
git restore --staged <file>  # Git 2.23+

# Commit
git commit -m "message"
git commit -am "message"  # add + commit (solo tracked files)
git commit --amend        # modifica ultimo commit
git commit --amend -m "new message"  # cambia messaggio

# Rimuovi file
git rm <file>           # rimuove e stage
git rm --cached <file> # rimuove da Git, mantieni locale
git rm -r <directory>  # ricorsivo

# Rinomina/Sposta file
git mv <old> <new>
```

A.5 Branch

```
# Lista branch
git branch              # locali
git branch -r           # remoti
git branch -a           # tutti
git branch -v           # con ultimo commit
git branch -vv          # con tracking info

# Crea branch
git branch <name>
git branch <name> <commit>  # da commit specifico

# Cambia branch
git checkout <branch>
git switch <branch>    # Git 2.23+

# Crea e cambia branch
git checkout -b <name>
git switch -c <name>   # Git 2.23+

# Rinomina branch
git branch -m <old-name> <new-name>
git branch -m <new-name>  # rinomina branch corrente

# Cancella branch
git branch -d <name>   # safe delete (merged)
git branch -D <name>   # force delete

# Branch tracking
git branch -u origin/<branch>  # imposta upstream
git branch --unset-upstream    # rimuovi upstream
```

A.6 Merge

```
# Merge branch nel corrente
git merge <branch>

# Merge con opzioni
git merge <branch> --no-ff     # crea sempre merge commit
git merge <branch> --squash    # squash tutti i commit
git merge <branch> -m "message"

# Abbandona merge
git merge --abort

# Merge strategies
git merge -X ours <branch>     # preferisci versione corrente
git merge -X theirs <branch>   # preferisci versione in merge

# Merge tool
git mergetool
git mergetool --tool=vimdiff
```

A.7 Rebase

```
# Rebase branch corrente su altro branch
git rebase <branch>
git rebase origin/main

# Rebase interattivo
git rebase -i HEAD~3   # ultimi 3 commit
git rebase -i <commit>

# Durante rebase
git rebase --continue  # dopo risoluzione conflitti
git rebase --abort     # abbandona
git rebase --skip      # salta commit corrente

# Opzioni rebase
git rebase -i --autosquash  # auto squash commit fixup/squash
```

A.8 Remote Repository

```
# Lista remote
git remote
git remote -v          # con URL

# Aggiungi remote
git remote add <name> <url>
git remote add origin https://github.com/user/repo.git

# Rimuovi/rinomina remote
git remote remove <name>
git remote rename <old> <new>

# Mostra info remote
git remote show origin

# Modifica URL remote
git remote set-url origin <new-url>

# Fetch (scarica senza merge)
git fetch
git fetch origin
git fetch --all        # tutti i remote
git fetch --prune      # rimuovi reference obsoleti

# Pull (fetch + merge)
git pull
git pull origin main
git pull --rebase      # rebase invece di merge
git pull --ff-only     # solo fast-forward

# Push
git push
git push origin main
git push -u origin main  # imposta upstream
git push --all         # tutti i branch
git push --tags        # tutti i tag
```

```
git push --force        # PERICOLO: forza push
git push --force-with-lease  # forza ma verifica remote
git push origin --delete <branch>  # cancella branch remoto
```

A.9 Stash

```
# Salva modifiche
git stash
git stash save "message"
git stash -u            # include untracked files
git stash --all         # include anche ignored files

# Lista stash
git stash list

# Applica stash
git stash apply         # applica ultimo, mantieni stash
git stash apply stash@{2}  # applica specifico
git stash pop           # applica e rimuovi stash

# Visualizza stash
git stash show
git stash show -p       # con diff

# Rimuovi stash
git stash drop stash@{0}
git stash clear         # rimuovi tutti

# Crea branch da stash
git stash branch <branch-name>
```

A.10 Reset e Revert

```
# Reset (modifica HEAD)
git reset <file>        # unstage file (mixed)
git reset               # unstage tutto

git reset --soft HEAD~1     # sposta HEAD, mantieni staging + working
git reset --mixed HEAD~1    # (default) resetta staging
git reset --hard HEAD~1     # PERICOLO: resetta tutto

# Reset a commit specifico
git reset --hard <commit>
git reset --hard origin/main

# Revert (crea commit che annulla)
git revert <commit>
git revert HEAD
git revert HEAD~3
git revert <commit1> <commit2>  # multipli

# Opzioni revert
git revert --no-commit <commit>  # revert senza committare
git revert --abort      # abbandona revert in corso
```

A.11 Cherry-Pick

```
# Applica commit da altro branch
git cherry-pick <commit>
git cherry-pick <commit1> <commit2>
git cherry-pick <commit-start>..<commit-end>

# Opzioni
git cherry-pick --no-commit <commit>  # applica senza commit
git cherry-pick -e <commit>  # modifica messaggio
git cherry-pick -x <commit>  # aggiungi reference originale

# Durante cherry-pick con conflitti
git cherry-pick --continue
git cherry-pick --abort
git cherry-pick --skip
```

A.12 Tag

```
# Lista tag
git tag
git tag -l "v1.*"      # pattern

# Crea tag
git tag <tag-name>     # lightweight
git tag -a <tag-name> -m "message"  # annotated
git tag -a <tag-name> <commit>  # su commit specifico

# Mostra tag
git show <tag-name>

# Push tag
git push origin <tag-name>
git push origin --tags  # tutti i tag
git push --follow-tags  # solo annotated

# Cancella tag
git tag -d <tag-name>  # locale
git push origin --delete <tag-name>  # remoto
git push origin :refs/tags/<tag-name>  # alternativa

# Checkout tag
git checkout <tag-name>  # detached HEAD
git checkout -b <branch-name> <tag-name>  # crea branch
```

A.13 Reflog

```
# Visualizza reflog
git reflog
git reflog show HEAD
git reflog show <branch>

# Limita output
git reflog -5          # ultimi 5 movimenti
```

```
8
9  # Reset usando reflog
10 git reset --hard HEAD@{2}
11 git reset --hard main@{yesterday}
12
13 # Crea branch da reflog
14 git branch <branch-name> HEAD@{3}
15
16 # Cleanup reflog
17 git reflog expire --expire=now --all
18 git reflog expire --expire=30.days.ago --all
```

A.14 Bisect

```
1  # Inizia bisect
2  git bisect start
3
4  # Marca commit good/bad
5  git bisect bad          # commit corrente ha bug
6  git bisect good <commit>  # commit senza bug
7
8  # Durante bisect
9  git bisect good         # commit corrente ok
10 git bisect bad          # commit corrente ha bug
11 git bisect skip         # non testabile
12
13 # Bisect automatico
14 git bisect run <script>
15
16 # Termina bisect
17 git bisect reset
18
19 # Visualizza log bisect
20 git bisect log
```

A.15 Informazioni e Diagnostica

```
1  # Verifica integrità repository
2  git fsck
3  git fsck --full
4
5  # Conta oggetti
6  git count-objects -v
7  git count-objects -vH  # human readable
8
9  # Garbage collection
10 git gc
11 git gc --aggressive --prune=now
12
13 # Blame (chi ha modificato ogni riga)
14 git blame <file>
15 git blame -L 10,20 <file>  # solo righe 10-20
16 git blame -w <file>  # ignora whitespace
17
```

```
# Grep (cerca nel codice)
git grep "pattern"
git grep -n "pattern"  # con numeri riga
git grep --count "pattern"  # conta occorrenze

# Trova commit che ha aggiunto/rimosso stringa
git log -S "function_name"
git log -G "regex_pattern"

# Mostra chi ha introdotto un file
git log --diff-filter=A -- <file>

# File tree
git ls-tree HEAD
git ls-tree -r HEAD    # ricorsivo
git ls-tree -r HEAD --name-only  # solo nomi
```

A.16 Pulizia e Manutenzione

```
# Rimuovi file untracked
git clean -n           # dry run (mostra cosa rimuove)
git clean -f           # rimuovi file
git clean -fd          # rimuovi file e directory
git clean -fx          # include ignored files

# Ottimizzazione
git gc --auto
git repack -a -d
git prune

# Verifica repository
git fsck --full --no-dangling
```

A.17 Submodules

```
# Aggiungi submodule
git submodule add <url> <path>

# Inizializza submodules dopo clone
git submodule init
git submodule update

# Clone con submodules
git clone --recursive <url>

# Update submodules
git submodule update --remote

# Rimuovi submodule
git submodule deinit <path>
git rm <path>
```

A.18 Worktree

```
# Crea worktree (multiple working directory)
git worktree add <path> <branch>
git worktree add ../hotfix hotfix-branch

# Lista worktree
git worktree list

# Rimuovi worktree
git worktree remove <path>
git worktree prune
```

A.19 Advanced

```
# Filter branch (riscrive cronologia)
git filter-branch --tree-filter 'rm -f passwords.txt' HEAD

# Filter repo (tool moderno)
git filter-repo --path file-to-keep --invert-paths

# Patch
git format-patch -1 HEAD  # crea patch da ultimo commit
git apply <patch-file>    # applica patch
git am <patch-file>       # applica come commit

# Archive
git archive --format=zip HEAD > archive.zip
git archive --format=tar.gz --prefix=project/ HEAD > project.tar.gz

# Bundle (repository portabile)
git bundle create repo.bundle --all
git clone repo.bundle -b main new-repo

# Sparse checkout
git sparse-checkout init --cone
git sparse-checkout set folder1 folder2

# Rerere (riuso resolution)
git config --global rerere.enabled true
```

A.20 Shortlog e Contributors

```
# Lista contributors
git shortlog -sn       # ordina per numero commit
git shortlog -sne      # include email

# Statistiche
git log --author="Name" --oneline | wc -l  # commit per autore
git log --since="1 year ago" --oneline | wc -l  # commit ultimo anno
```

A.21 Tabella Riassuntiva: Annullare Modifiche

Scenario	Comando	Pericolo
File modificato (unstaged)	`git restore <file>`	SÌ (perdi modifiche)
File in staging	`git restore -staged <file>`	NO
Modifica ultimo commit	`git commit -amend`	Medio (se già pushato)
Annulla ultimo commit	`git reset -soft HEAD~1`	NO
Annulla + unstage	`git reset HEAD~1`	NO
Distruggi ultimo commit	`git reset -hard HEAD~1`	SÌ (perdi modifiche)
Annulla commit pushato	`git revert <commit>`	NO (sicuro)

A.22 Tabella Riassuntiva: Reset

Opzione	HEAD	Staging	Working Dir
`-soft`	Modifica	Invariato	Invariato
`-mixed` (default)	Modifica	Resetta	Invariato
`-hard`	Modifica	Resetta	Resetta

A.23 Simboli Speciali

```
HEAD          # Commit corrente
HEAD~1        # 1 commit indietro
HEAD~3        # 3 commit indietro
HEAD^         # Parent del commit (equivalente a HEAD~1)
HEAD^^        # 2 commit indietro

# In merge commits (multiple parents)
HEAD^1        # Primo parent
HEAD^2        # Secondo parent

# Combinazioni
main~3        # 3 commit indietro da main
origin/main # Branch main su remote origin

# Reference
@             # Shortcut per HEAD
@{-1}         # Branch precedente
main@{yesterday}  # main di ieri
main@{2.days.ago} # main 2 giorni fa
```

A.24 Pattern .gitignore

```
# Commenti
# Questo è un commento

# File specifico
debug.log

# Tutti i file con estensione
*.log
*.tmp

# Directory
```

```
node_modules/
dist/

# Pattern ricorsivo
**/*.pyc

# Negazione (non ignorare)
!important.log

# Solo in root
/TODO.txt

# Tutti tranne
build/*
!build/version.txt
```

A.25 Caratteri Speciali in Comandi

```
# Range di commit
git log <commit1>..<commit2>    # da commit1 a commit2 (escluso commit1)
git log <commit1>...<commit2>   # symmetric difference

# Tutti i commit di branch2 non in branch1
git log branch1..branch2

# Commit raggiungibili da HEAD ma non da origin/main
git log origin/main..HEAD

# Tutti i file modificati tra commit
git diff <commit1>..<commit2>
```

Suggerimento

Stampa questa appendice e tienila a portata di mano! I comandi Git sono tanti, ma con la pratica i più comuni diventeranno automatici.

Appendice B

Appendice: Esercizi e Soluzioni

Introduzione

Questa appendice contiene esercizi pratici completi per consolidare le competenze Git. Ogni esercizio include obiettivi, istruzioni dettagliate e soluzione completa. Gli esercizi sono organizzati per difficoltà crescente.

B.1 Livello Base

B.1.1 Esercizio 1.1: Primo Repository

Obiettivo: Creare repository, fare commit, visualizzare cronologia.

Task:

1. Crea directory `my-first-repo`
2. Inizializza repository Git
3. Crea file `README.md` con contenuto " My Project"
4. Aggiungi e committa il file
5. Crea file `app.js` con `console.log("Hello")`
6. Committa `app.js`
7. Visualizza cronologia commit

Soluzione:

```
# 1-2. Crea directory e inizializza
mkdir my-first-repo
cd my-first-repo
git init

# 3-4. Crea e committa README
echo "# My Project" > README.md
git add README.md
git commit -m "Add README"

# 5-6. Crea e committa app.js
echo 'console.log("Hello");' > app.js
git add app.js
git commit -m "Add app.js with hello message"

```

```
# 7. Visualizza cronologia
git log
git log --oneline
```

B.1.2 Esercizio 1.2: Modifiche e Staging

Obiettivo: Gestire modifiche, staging, e undo.

Task:

1. Modifica `app.js` aggiungendo altra console.log
2. Aggiungi a staging
3. Visualizza status
4. Rimuovi da staging
5. Verifica che modifiche siano ancora presenti
6. Scarta le modifiche

Soluzione:

```
# 1. Modifica file
echo 'console.log("World");' >> app.js

# 2. Aggiungi a staging
git add app.js

# 3. Verifica status
git status
# Changes to be committed:
#   modified: app.js

# 4. Rimuovi da staging
git restore --staged app.js
# oppure: git reset HEAD app.js

# 5. Verifica modifiche ancora presenti
git status
# Changes not staged for commit:
#   modified: app.js

cat app.js  # le modifiche ci sono ancora

# 6. Scarta modifiche
git restore app.js
# oppure: git checkout -- app.js

# Verifica
cat app.js  # torna a versione committata
```

B.1.3 Esercizio 1.3: Branch Base

Obiettivo: Creare branch, switchare, fare merge.

Task:

1. Crea branch `feature-greeting`

2. Cambia a quel branch
3. Modifica `app.js` cambiando "Hello" in "Hi"
4. Committa modifiche
5. Torna a `main`
6. Merge `feature-greeting`
7. Cancella branch feature

Soluzione:

```
# 1-2. Crea e cambia branch
git checkout -b feature-greeting
# oppure: git switch -c feature-greeting

# 3. Modifica file
echo 'console.log("Hi");' > app.js

# 4. Committa
git add app.js
git commit -m "Change greeting to Hi"

# 5. Torna a main
git checkout main

# Verifica: app.js ancora con "Hello"
cat app.js

# 6. Merge
git merge feature-greeting

# Verifica merge
cat app.js  # ora ha "Hi"
git log --oneline --graph --all

# 7. Cancella branch
git branch -d feature-greeting
```

B.2 Livello Intermedio

B.2.1 Esercizio 2.1: Merge Conflicts

Obiettivo: Creare e risolvere merge conflict.

Task:

1. Su `main`, modifica `app.js`: "console.log('Main branch')"
2. Committa
3. Crea branch `feature-A`
4. Su `feature-A`, modifica stessa riga: "console.log('Feature A')"
5. Committa
6. Torna a `main`

7. Tenta merge di `feature-A`
8. Risolvi conflitto mantenendo entrambi i messaggi
9. Completa merge

Soluzione:

```
# 1-2. Modifica su main
echo "console.log('Main branch');" > app.js
git add app.js
git commit -m "Update message on main"

# 3-5. Branch e modifica
git checkout -b feature-A
echo "console.log('Feature A');" > app.js
git add app.js
git commit -m "Update message on feature-A"

# 6. Torna a main
git checkout main

# 7. Tenta merge (CONFLICT!)
git merge feature-A
# Auto-merging app.js
# CONFLICT (content): Merge conflict in app.js

# 8. Visualizza conflitto
cat app.js
# <<<<<<< HEAD
# console.log('Main branch');
# =======
# console.log('Feature A');
# >>>>>>> feature-A

# Risolvi manualmente
cat > app.js << 'EOF'
console.log('Main branch');
console.log('Feature A');
EOF

# 9. Completa merge
git add app.js
git commit -m "Merge feature-A, keep both messages"

# Verifica
git log --oneline --graph --all
```

B.2.2 Esercizio 2.2: Remote Repository

Obiettivo: Simulare collaborazione con remote.

Task:

1. Crea repository "bare" come remote simulato
2. Configura come remote `origin`
3. Push branch `main`

4. Crea branch `feature-B`, committa modifiche
5. Push `feature-B`
6. Simula clone in altra directory
7. Nel clone, modifica e push
8. Nel repo originale, pull modifiche

Soluzione:

```
# 1. Crea bare repository (simula GitHub)
cd /tmp
git init --bare remote-repo.git

# 2. Configura remote nel tuo repo
cd /path/to/my-first-repo
git remote add origin /tmp/remote-repo.git

# 3. Push main
git push -u origin main

# 4-5. Crea feature branch e push
git checkout -b feature-B
echo "console.log('Feature B');" >> app.js
git add app.js
git commit -m "Add feature B"
git push -u origin feature-B

# 6. Simula clone (altro developer)
cd /tmp
git clone /tmp/remote-repo.git developer2-repo
cd developer2-repo

# 7. Modifica e push
echo "# Updated by dev2" >> README.md
git add README.md
git commit -m "Update README"
git push origin main

# 8. Nel repo originale, pull
cd /path/to/my-first-repo
git checkout main
git pull origin main

# Verifica modifiche
cat README.md
```

B.2.3 Esercizio 2.3: Rebase vs Merge

Obiettivo: Confrontare merge e rebase.

Task:

1. Setup: main con 2 commit, branch `feature-merge` con 2 commit
2. Merge `feature-merge` in `main`
3. Osserva cronologia (merge commit presente)

4. Reset `main` a prima del merge
5. Questa volta: checkout `feature-merge` e rebase su `main`
6. Merge (fast-forward)
7. Osserva cronologia (lineare, no merge commit)

Soluzione:

```
# Setup
git checkout main
echo "Line 1" > file.txt
git add file.txt
git commit -m "Commit 1 on main"

echo "Line 2" >> file.txt
git add file.txt
git commit -m "Commit 2 on main"

# Branch feature
git checkout -b feature-merge
echo "Feature line 1" > feature.txt
git add feature.txt
git commit -m "Feature commit 1"

echo "Feature line 2" >> feature.txt
git add feature.txt
git commit -m "Feature commit 2"

# Merge (crea merge commit)
git checkout main
git merge feature-merge

# Cronologia
git log --oneline --graph --all
# * abc123 (HEAD -> main) Merge branch 'feature-merge'
# |\
# | * def456 Feature commit 2
# | * ghi789 Feature commit 1
# * | jkl012 Commit 2 on main
# * | mno345 Commit 1 on main

# Reset per riprovare con rebase
git reset --hard HEAD~1  # torna prima del merge

# Rebase invece
git checkout feature-merge
git rebase main

# Ora merge (sarà fast-forward)
git checkout main
git merge feature-merge

# Cronologia lineare
git log --oneline --graph --all
# * def456 (HEAD -> main, feature-merge) Feature commit 2
# * ghi789 Feature commit 1
# * jkl012 Commit 2 on main
# * mno345 Commit 1 on main
```

B.3 Livello Avanzato

B.3.1 Esercizio 3.1: Interactive Rebase

Obiettivo: Riorganizzare commit con rebase interattivo.

Task:

1. Crea 5 commit con messaggi "WIP 1", "WIP 2", "Feature done", "WIP 3", "Fix typo"
2. Usa `git rebase -i` per:
   - Squash "WIP 1", "WIP 2", "WIP 3" in "Feature done"
   - Fixup "Fix typo" nel commit precedente
   - Modifica messaggio finale
3. Risultato: 1 commit pulito

Soluzione:

```
# Setup: 5 commit
echo "v1" > code.js
git add code.js
git commit -m "WIP 1"

echo "v2" >> code.js
git commit -am "WIP 2"

echo "v3" >> code.js
git commit -am "Feature done"

echo "v4" >> code.js
git commit -am "WIP 3"

echo "v4 fixed" > code.js
git commit -am "Fix typo"

# Verifica
git log --oneline
# abc123 Fix typo
# def456 WIP 3
# ghi789 Feature done
# jkl012 WIP 2
# mno345 WIP 1

# Rebase interattivo ultimi 5 commit
git rebase -i HEAD~5

# Editor si apre con:
# pick mno345 WIP 1
# pick jkl012 WIP 2
# pick ghi789 Feature done
# pick def456 WIP 3
# pick abc123 Fix typo

# Modifica a:
# pick mno345 WIP 1
# squash jkl012 WIP 2
# squash ghi789 Feature done
# squash def456 WIP 3
```

```
# fixup abc123 Fix typo

# Salva e chiudi. Si apre editor per messaggio:
# Usa: "Implement feature X with all improvements"

# Risultato finale
git log --oneline
# xyz999 Implement feature X with all improvements
```

B.3.2 Esercizio 3.2: Stash Workflow

Obiettivo: Usare stash in scenario reale.

Task:

1. Stai lavorando su `feature-X` con modifiche non committate
2. Ti chiedono urgentemente di fixare bug su `main`
3. Usa stash per salvare lavoro temporaneo
4. Cambia a `main`, fixa bug, committa
5. Torna a `feature-X`
6. Recupera modifiche da stash
7. Completa feature e committa

Soluzione:

```
# 1. Setup: lavoro su feature
git checkout -b feature-X
echo "Feature work in progress" > feature.js
echo "More feature work" >> feature.js

# Modifiche non committate
git status
# modified: feature.js (not staged)

# 2-3. Emergenza! Salva con stash
git stash save "WIP: feature X implementation"

# Verifica working directory pulito
git status
# nothing to commit, working tree clean

# 4. Cambia a main e fixa bug
git checkout main
echo "critical fix" > bugfix.js
git add bugfix.js
git commit -m "fix: critical security bug"

# 5. Torna a feature branch
git checkout feature-X

# 6. Recupera lavoro
git stash list
# stash@{0}: On feature-X: WIP: feature X implementation

```

```
git stash pop

# Verifica modifiche tornate
cat feature.js

# 7. Completa e committa
echo "Feature complete" >> feature.js
git add feature.js
git commit -m "feat: complete feature X"
```

B.3.3 Esercizio 3.3: Cherry-Pick Scenario

Obiettivo: Applicare commit specifico tra branch.

Task:

1. Branch `develop` ha 5 commit
2. Solo commit 3 contiene fix critico
3. Applica solo quel fix su `main`
4. Verifica che altri commit non siano su `main`

Soluzione:

```
# Setup develop con 5 commit
git checkout -b develop
echo "Feature 1" > f1.js
git add f1.js
git commit -m "feat: feature 1"

echo "Feature 2" > f2.js
git add f2.js
git commit -m "feat: feature 2"

echo "Critical fix" > fix.js
git add fix.js
CRITICAL_COMMIT=$(git rev-parse HEAD)
git commit -m "fix: critical security issue"

echo "Feature 3" > f3.js
git add f3.js
git commit -m "feat: feature 3"

echo "Feature 4" > f4.js
git add f4.js
git commit -m "feat: feature 4"

# Visualizza commit
git log --oneline
# Identifica hash del commit "critical security issue"

# Cherry-pick su main
git checkout main
git cherry-pick $CRITICAL_COMMIT

# Verifica
ls
# Deve esserci solo fix.js, non f1-f4.js
```

```
35
36 git log --oneline
37 # Deve mostrare solo il commit del fix
```

B.3.4 Esercizio 3.4: Reflog Recovery

Obiettivo: Recuperare commit dopo reset disastroso.

Task:

1. Crea 5 commit importanti
2. Per errore fai `git reset -hard HEAD~5`
3. Panic! Hai perso tutto
4. Usa reflog per recuperare
5. Verifica che commit siano tornati

Soluzione:

```
1  # 1. Crea commit importanti
2  for i in {1..5}; do
3      echo "Important work $i" > file$i.txt
4      git add file$i.txt
5      git commit -m "Important commit $i"
6  done
7
8  # Verifica
9  git log --oneline
10 # abc123 Important commit 5
11 # def456 Important commit 4
12 # ...
13
14 # 2. DISASTER: Reset hard
15 git reset --hard HEAD~5
16
17 # 3. Verifica: tutto perso!
18 git log --oneline
19 # (nessuno dei commit importanti)
20 ls
21 # file1-5.txt non esistono
22
23 # 4. RECOVERY: usa reflog
24 git reflog
25
26 # Output:
27 # xyz789 HEAD@{0}: reset: moving to HEAD~5
28 # abc123 HEAD@{1}: commit: Important commit 5
29 # def456 HEAD@{2}: commit: Important commit 4
30 # ...
31
32 # Recupera tornando a HEAD@{1}
33 git reset --hard HEAD@{1}
34 # oppure usando hash direttamente
35 # git reset --hard abc123
36
37 # 5. Verifica: tutto tornato!
38 git log --oneline
```

```
ls  # tutti i file tornati
```

B.3.5 Esercizio 3.5: Bisect per Bug Hunting

Obiettivo: Trovare commit che ha introdotto bug.

Task:

1. Crea 20 commit
2. Commit 13 introduce bug (divide by zero)
3. Usa bisect per trovarlo

Soluzione:

```
# Setup: crea 20 commit
for i in {1..20}; do
    if [ $i -eq 13 ]; then
        # Introduce bug al commit 13
        echo "result = 100 / 0;" > calc.js
    else
        # Codice corretto
        echo "result = 100 / $i;" > calc.js
    fi
    git add calc.js
    git commit -m "Update calculation (commit $i)"
done

# Ora sei a commit 20, test fallisce
# Sai che commit 1 era ok

# Inizia bisect
git bisect start

# Marca corrente (20) come bad
git bisect bad

# Marca commit 1 come good
git bisect good HEAD~19

# Git checkoutterà commit 10 (metà tra 1 e 20)
# Testa
cat calc.js

# Se ok (commit 1-12):
git bisect good

# Se bug (commit 13-20):
git bisect bad

# Continua finché Git trova il colpevole
# Output finale:
# abc123 is the first bad commit
# commit abc123
# Author: ...
# Date: ...
# Update calculation (commit 13)

```

```
# Termina bisect
git bisect reset

# Verifica commit colpevole
git show abc123
```

B.4 Progetti Completi

B.4.1 Progetto 1: Blog Engine con Git Flow

Obiettivo: Implementare completo Git Flow workflow.

Scenario: Stai sviluppando blog engine. Release v1.0.0 in produzione. Devi:

- Sviluppare feature "comments"
- Preparare release v1.1.0
- Fixare bug urgente in produzione (v1.0.1)

Soluzione:

```
# Setup iniziale
mkdir blog-engine
cd blog-engine
git init

# Crea main e develop
echo "# Blog Engine v1.0.0" > README.md
git add README.md
git commit -m "Initial commit"
git tag -a v1.0.0 -m "Release 1.0.0"

git checkout -b develop

# Feature: Comments
git checkout -b feature/comments develop

cat > comments.js << 'EOF'
class Comments {
    constructor() {
        this.comments = [];
    }

    add(comment) {
        this.comments.push(comment);
    }

    getAll() {
        return this.comments;
    }
}

module.exports = Comments;
EOF

git add comments.js
git commit -m "feat: add comments system"

```

```
# Merge feature in develop
git checkout develop
git merge --no-ff feature/comments -m "Merge feature/comments into
    develop"
git branch -d feature/comments

# Release branch
git checkout -b release/1.1.0 develop

# Update version
echo "# Blog Engine v1.1.0" > README.md
git commit -am "chore: bump version to 1.1.0"

# Merge release in main
git checkout main
git merge --no-ff release/1.1.0 -m "Release 1.1.0"
git tag -a v1.1.0 -m "Release 1.1.0 - Add comments"

# Merge release back to develop
git checkout develop
git merge --no-ff release/1.1.0
git branch -d release/1.1.0

# HOTFIX urgente su v1.0.0
git checkout -b hotfix/1.0.1 v1.0.0

cat > security-fix.js << 'EOF'
// Critical security patch
function sanitizeInput(input) {
    return input.replace(/[<>]/g, '');
}
module.exports = sanitizeInput;
EOF

git add security-fix.js
git commit -m "fix: critical XSS vulnerability"

# Merge hotfix in main
git checkout main
git merge --no-ff hotfix/1.0.1 -m "Hotfix 1.0.1"
git tag -a v1.0.1 -m "Hotfix 1.0.1 - Security patch"

# Merge hotfix in develop
git checkout develop
git merge --no-ff hotfix/1.0.1
git branch -d hotfix/1.0.1

# Visualizza cronologia completa
git log --oneline --graph --all --decorate
```

B.4.2 Progetto 2: Open Source Contribution

Obiettivo: Simulare contributo a progetto open source.

Scenario:

1. Fork di repository esistente
2. Crea feature branch

3. Implementa feature
4. Apri Pull Request
5. Code review: richieste modifiche
6. Aggiorna PR
7. Merge

Soluzione:

```
# Simula "upstream" repository (progetto originale)
mkdir /tmp/upstream-project
cd /tmp/upstream-project
git init --bare

# Crea contenuto iniziale
cd /tmp
git clone /tmp/upstream-project original
cd original
echo "# Awesome Project" > README.md
git add README.md
git commit -m "Initial commit"
git push origin main

# Fork (simula)
cd /tmp
git clone /tmp/upstream-project my-fork
cd my-fork

# Configura remote
git remote rename origin upstream
git remote add origin /tmp/my-fork-remote.git
git init --bare /tmp/my-fork-remote.git
git push -u origin main

# Feature branch
git checkout -b feature/add-documentation

# Implementa feature
cat > CONTRIBUTING.md << 'EOF'
# Contributing Guide

## How to Contribute
1. Fork repository
2. Create feature branch
3. Make changes
4. Submit Pull Request

## Code Style
- Use 2 spaces for indentation
- Add tests for new features
EOF

git add CONTRIBUTING.md
git commit -m "docs: add contributing guide"

# Push branch
git push -u origin feature/add-documentation
```

```
49
50 # Simula Pull Request aperta
51 # Maintainer chiede modifiche
52
53 # Code review: aggiungi section testing
54 cat >> CONTRIBUTING.md << 'EOF'
55
56 ## Testing
57 Run tests with: npm test
58 EOF
59
60 git commit -am "docs: add testing section"
61 git push origin feature/add-documentation
62
63 # Maintainer approva e merge
64 git checkout main
65 git pull upstream main
66 git merge --no-ff feature/add-documentation -m "Merge PR #1: Add
       contributing guide"
67 git push upstream main
68
69 # Cleanup
70 git branch -d feature/add-documentation
71 git push origin --delete feature/add-documentation
72
73 # Update fork
74 git push origin main
```

B.4.3 Progetto 3: CI/CD Pipeline Completa

Obiettivo: Setup completo CI/CD con GitHub Actions.

Soluzione:

```
1  # Crea progetto Node.js
2  mkdir my-app
3  cd my-app
4  git init
5
6  # Package.json
7  cat > package.json << 'EOF'
8  {
9    "name": "my-app",
10   "version": "1.0.0",
11   "scripts": {
12     "test": "echo 'Running tests...' && exit 0",
13     "lint": "echo 'Linting code...' && exit 0",
14     "build": "echo 'Building app...' && mkdir -p dist && echo 'built' >
         dist/app.js"
15   }
16 }
17 EOF
18
19 # App code
20 cat > index.js << 'EOF'
21 function greet(name) {
22     return `Hello, ${name}!`;
23 }
24
```

```
console.log(greet('World'));
module.exports = { greet };
EOF

# Tests
cat > test.js << 'EOF'
const { greet } = require('./index');

if (greet('Test') === 'Hello, Test!') {
    console.log('    Test passed');
    process.exit(0);
} else {
    console.log('    Test failed');
    process.exit(1);
}
EOF

# GitHub Actions workflow
mkdir -p .github/workflows
cat > .github/workflows/ci-cd.yml << 'EOF'
name: CI/CD Pipeline

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
      - run: npm run lint

  test:
    runs-on: ubuntu-latest
    needs: lint
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
      - run: npm run test

  build:
    runs-on: ubuntu-latest
    needs: test
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
```

```
          node-version: '18'
      - run: npm run build
      - name: Upload artifacts
        uses: actions/upload-artifact@v3
        with:
          name: build
          path: dist/

  deploy:
    runs-on: ubuntu-latest
    needs: build
    if: github.ref == 'refs/heads/main'
    steps:
      - name: Download artifacts
        uses: actions/download-artifact@v3
        with:
          name: build
      - name: Deploy
        run: echo "Deploying to production..."
EOF

# Commit tutto
git add .
git commit -m "feat: initial project setup with CI/CD"

# Crea feature branch per testare PR workflow
git checkout -b feature/improve-greeting

# Modifica
cat > index.js << 'EOF'
function greet(name) {
    return `Hello, ${name}! Welcome!`;
}

console.log(greet('World'));
module.exports = { greet };
EOF

# Update test
cat > test.js << 'EOF'
const { greet } = require('./index');

if (greet('Test') === 'Hello, Test! Welcome!') {
    console.log('    Test passed');
    process.exit(0);
} else {
    console.log('    Test failed');
    process.exit(1);
}
EOF

git commit -am "feat: improve greeting message"

# Simula PR e merge
git checkout main
git merge --no-ff feature/improve-greeting -m "Merge PR: Improve
    greeting"
git branch -d feature/improve-greeting
```

```
140
141 # Visualizza pipeline (in GitHub, workflow runs automaticamente)
142 echo "    CI/CD pipeline configurata"
143 echo "    Workflow triggherà su push e PR"
144 echo "    Deploy automatico su main"
```

B.5 Challenge Avanzati

B.5.1 Challenge 1: Repository Corrupted Recovery

Scenario: Repository corrotto, alcuni oggetti mancanti.
Task: Recupera quanto possibile usando fsck e reflog.

B.5.2 Challenge 2: Monorepo Management

Task: Gestisci monorepo con 3 progetti, ognuno con versioning indipendente.

B.5.3 Challenge 3: Complex Rebase

Task: Branch con 50 commit, rebase interattivo per riorganizzare in 10 commit logici.

Congratulazioni!

Completando questi esercizi hai acquisito competenze Git solide. La pratica costante è fondamentale: usa Git quotidianamente nei tuoi progetti per consolidare le conoscenze.
Next steps:

- Contribuisci a progetti open source
- Setup CI/CD per progetti personali
- Esplora Git hooks per automazioni
- Studia Git internals (objects, pack files)