Sistemi di Controllo Versione Distribuito

Analisi Tecnica e Applicativa di Git

Alessio Avarappattu

Indice

1 Git **5**

1.1 Introduzione . . . 5
1.1.1 Cenni Storici e Motivazione . . . 5
1.1.2 Funzionalità Principali . . . 5
1.1.3 Modello di Dati: Snapshot vs Delta . . . 5
1.2 Installazione e Configurazione . . . 6
1.2.1 Installazione . . . 6
1.2.2 Gerarchia di Configurazione . . . 6
1.2.3 Configurazione Iniziale . . . 7
1.3 Struttura Interna: La Directory .git . . . 8
1.3.1 Struttura del File System . . . 8
1.3.2 Componenti Chiave . . . 8
1.3.3 La Staging Area (Index) . . . 9
1.4 Il Ciclo di Vita dei File . . . 10
1.4.1 Le Tre Aree di Lavoro . . . 10
1.4.2 Fase 1: Lo Staging (git add) . . . 10
1.4.3 Stato dei File . . . 10
1.5 Il Commit . . . 12
1.5.1 Struttura di un Oggetto Commit . . . 12
1.5.2 1. Atomicità . . . 12
1.5.3 2. Il Messaggio (Imperative Mood) . . . 12
1.5.4 Comandi Operativi . . . 13
1.5.5 Correggere l'ultimo Commit (–amend) . . . 13
1.6 Git Status: Analisi dello Stato Corrente . . . 14
1.6.1 Logica di Comparazione . . . 14
1.6.2 Interpretazione dell'Output Standard . . . 14
1.6.3 Esclusione dei File (.gitignore) . . . 14
1.6.4 Rimozione dal Versionamento (Un-track) . . . 15
1.6.5 Short Format (Output) . . . 15
1.7 I Tre Alberi . . . 17
1.7.1 Definizione delle Entità . . . 17
1.7.2 Visualizzazione del Flusso di Modifica . . . 17
1.7.3 I Delta e il comando Git Diff . . . 17
1.8 Gestione delle Ramificazioni (Branching) . . . 19
1.8.1 Definizione Tecnica . . . 19
1.8.2 In Pratica... . . . 19
1.8.3 Il Ruolo di HEAD . . . 20
1.8.4 Creazione e Cambio (Switching) . . . 20
1.8.5 Divergenza della Storia . . . 21
1.8.6 Gestione e Pulizia . . . 21
1.8.7 Detached HEAD (Testa Distaccata) . . . 22

1.9 `git log` . . . 24
1.9.1 Reachability (Raggiungibilità) . . . 24
1.9.2 Visualizzazione Topologica . . . 24
1.9.3 Filtraggio dei Dati (Data Mining) . . . 24
1.9.4 Git diff - Analisi delle Differenze) . . . 26
1.10 `git merge` . . . 29
1.10.1 Strategia 1: Fast-Forward . . . 29
1.10.2 Strategia 2: Three-Way Merge (Merge Commit) . . . 29
1.10.3 Gestione dei Conflitti . . . 30
1.11 Salvataggio Temporaneo (Stash) . . . 32
1.11.1 Topologia dello Stash . . . 32
1.11.2 Operazioni Fondamentali . . . 33
1.11.3 Gestione e Pulizia . . . 33
1.12 Riscritura della Storia (Rebase) . . . 35
1.12.1 Topologia e Funzionamento . . . 35
1.12.2 Interactive Rebase (-i) . . . 35
1.12.3 Attenzione al rebase . . . 36
1.12.4 Merge o Rebase? . . . 36

2 GitHub e i Remote Repository **38**
2.1 Architettura Distribuita: Git vs GitHub . . . 38
2.1.1 Topologia del Workflow Remoto . . . 38
2.2 Configurazione dei Remotes . . . 39
2.2.1 Protocolli di Connessione . . . 39
2.2.2 Configurazione dell'Autenticazione Crittografica (SSH) . . . 39
2.3 `git remote` . . . 41
2.3.1 Ispezione dei Remote Esistenti . . . 41
2.3.2 Aggiunta di un nuovo Remote . . . 41
2.3.3 Gestione degli Errori Comuni . . . 42
2.3.4 Stato del Repository Locale e Vincoli di Sincronizzazione . . . 42
2.4 `git push` . . . 43
2.4.1 Sintassi e Flusso di Lavoro . . . 44
2.4.2 Fast-Forward . . . 44
2.4.3 Force Push . . . 45
2.4.4 Push di Tag . . . 46
2.4.5 Configurazione del Tracciamento (Upstream) . . . 46
2.5 `git clone` . . . 48
2.5.1 Sintassi Fondamentale . . . 48
2.5.2 Struttura del comando . . . 48
2.5.3 Ottimizzazione delle Risorse . . . 49
2.5.4 Potenziali limitazioni derivanti dai Repository Parziali . . . 50
2.6 `git fetch` . . . 51
2.6.1 Principio di Sicurezza (Non-Distruttivo) . . . 51
2.6.2 Manutenzione dei Riferimenti (`--prune`) . . . 52
2.7 `git pull` . . . 52
2.7.1 Dinamica dell'Operazione . . . 53
2.7.2 Comparazione dei Flussi di Dati . . . 53
2.7.3 Conseguenze del `git pull` . . . 53
2.7.4 Misconcetti sul `git pull` . . . 54

3 Collaborazione Distribuita e Modelli di Sviluppo **56**
3.1 GitHub e l'Equivoco dell'Open Source 56
3.1.1 Codice Proprietario vs. Codice Aperto 56
3.1.2 Modello Social Coding 57
3.2 Il Flusso di Contribuzione Standard 57
3.2.1 L'Unità di Lavoro: GitHub Issues 57
3.2.2 Protocollo di Avvio Lavori 58
3.3 Dualità del Fork: Contribuzione vs Divergenza 59
3.3.1 1. Il Fork Temporaneo (Workflow di Contribuzione) 60
3.3.2 2. Il Fork Permanente (Hard Fork / Sviluppo Indipendente) 60
3.4 Caso di Studio: Il Ciclo di Vita di una Contribuzione 61
3.4.1 Fase 1: Apertura della Issue (Segnalazione) 61
3.4.2 Fase 2: Preparazione dell'Ambiente (Fork & Clone) 61
3.4.3 Fase 3: Isolamento e Correzione (Branching) 61
3.4.4 Fase 4: Pubblicazione e Proposta (Push & Pull Request) 62
3.4.5 Visualizzazione del Flusso Architetturale 62
3.5 Manutenzione del Fork 63
3.5.1 Configurazione del Remote "Upstream" 63
3.5.2 La Procedura di Sincronizzazione 64
3.6 Gestione dei Conflitti di Merge 64
3.6.1 Struttura di un Conflitto 65
3.6.2 Protocollo di Risoluzione 65
3.7 L'importanza del File README.md 66
3.7.1 Standard Tecnologico: Markdown 66
3.7.2 Anatomia di un README Professionale 66
3.7.3 Esempio di Sintassi Markdown 67

Elenco degli snippet di codice

1.1 Comandi di installazione per Linux . . . 6
1.2 Setup iniziale obbligatorio . . . 7
1.3 Struttura della directory .git . . . 8
1.4 Varianti di git add . . . 10
1.5 Esempio di git status . . . 11
1.6 Comandi di Commit . . . 13
1.7 Modificare l'ultimo commit . . . 13
1.8 Esempio di output verboso . . . 14
1.9 Workflow per ignorare file già tracciati . . . 15
1.10 Esempio di file .gitignore . . . 15
1.11 Esempio reale di Short Status . . . 16
1.12 Gestione dei Branch . . . 21
1.13 Cancellazione Branch . . . 21
1.14 Salvataggio di un Detached HEAD . . . 23
1.15 Visualizzazione del Grafo . . . 24
1.16 Filtri temporali e numerici . . . 25
1.17 Ricerca nel codice sorgente . . . 25
1.18 Storia specifica di un modulo . . . 25
1.19 Esempio di output statistico . . . 25
1.20 Analisi differenziale tra commit . . . 26
1.21 Analisi differenziale tra branch . . . 26
1.22 Output di una modifica standard . . . 27
1.23 Output di cancellazione file . . . 27
1.24 Esempio di Unified Diff . . . 28
1.25 Salvataggio con messaggio . . . 33
1.26 Applicazione specifica . . . 33
2.1 Generazione chiave Ed25519 . . . 40
2.2 Attivazione dell'Agent . . . 40
2.3 Visualizzazione della chiave pubblica . . . 40
2.4 Ispezione dei remote con flag verbose . . . 41
2.5 Esempio di aggiunta remote . . . 41
2.6 Errore di duplicazione . . . 42
2.7 Primo invio con configurazione upstream . . . 44
2.8 Errore di Non-Fast-Forward . . . 44
2.9 Meccanica interna di un git clone . . . 48
2.10 Ripristino della visibilità completa . . . 51
3.1 Esempio di correzione del bug . . . 62
3.2 Esempio di file con marcatori di conflitto . . . 65
3.3 Esempio di file README.md raw . . . 67

Capitolo 1

Git

1.1 Introduzione

1.1.1 Cenni Storici e Motivazione

Prima dell'avvento dei moderni sistemi di controllo versione (VCS), la gestione delle modifiche avveniva manualmente tramite copie di file (es. `progetto_v1`, `progetto_finale`). Questo approccio presentava criticità quali ridondanza dei dati, alta probabilità di errore umano e difficoltà nella collaborazione simultanea.

Git è un **Distributed Version Control System (DVCS)**, sviluppato nel 2005 per la gestione del kernel Linux. A differenza dei sistemi centralizzati, ogni client possiede una copia completa della cronologia del progetto.

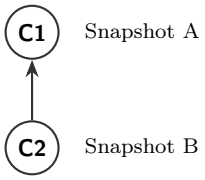
> Git è un database basato su un **Grafo Orientato Aciclico (DAG)** che memorizza istantanee (snapshot) del filesystem. Ogni commit è un nodo immutabile che contiene un riferimento al suo stato genitore.

1.1.2 Funzionalità Principali

- **Versionamento:** Accesso a qualsiasi stato precedente del progetto.
- **Branching:** Creazione di ramificazioni divergenti per lo sviluppo isolato di funzionalità.
- **Merging:** Algoritmi automatici per l'unione di modifiche concorrenti.
- **Audit:** Tracciabilità completa delle modifiche (autore, data, motivazione).

1.1.3 Modello di Dati: Snapshot vs Delta

I sistemi legacy (es. SVN) memorizzano le differenze (delta) tra i file. Git memorizza l'intero stato dei file per ogni commit (snapshot). Per efficienza, i file non modificati non vengono duplicati, ma referenziati tramite puntatori.



1.2 Installazione e Configurazione

1.2.1 Installazione

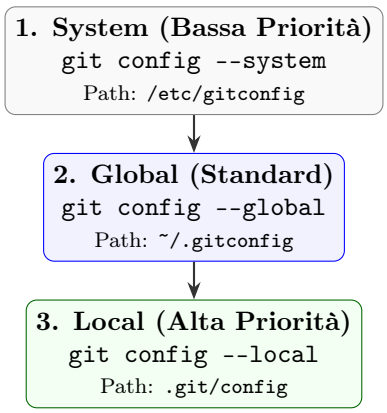
Si raccomanda l'installazione tramite il gestore di pacchetti del sistema operativo in uso.

Codice 1.1: Comandi di installazione per Linux

```
# Debian / Ubuntu
sudo apt update && sudo apt install git

# Fedora / RHEL
sudo dnf install git

# Verifica versione
git --version
```

1.2.2 Gerarchia di Configurazione

Git determina le impostazioni seguendo un rigoroso ordine di sovrascrittura (override). Quando viene lanciato un comando, Git legge i file in sequenza: l'ultimo letto ha priorità massima



Dettaglio dei Livelli

1. **System:** Contiene configurazioni valide per *ogni utente* del sistema operativo e ogni repository. Richiede permessi di amministratore ('sudo'). Raramente modificato manualmente.
2. **Global:** È il livello dove risiede l'"identità predefinita". Qui si impostano il nome, l'email personale e le preferenze dell'editor.
3. **Local:** Specifico per il singolo repository. Si trova nella cartella nascosta `.git` del progetto. Viene usato per le eccezioni.

Caso d'Uso: Doppia Identità (Lavoro vs Personale)

Si ipotizzi di lavorare su un progetto aziendale. La configurazione **Global** contiene l'email personale, ma per questo specifico progetto è necessario usare quella aziendale.
Soluzione: Accedere alla cartella del progetto aziendale ed eseguire:

```
# Questo comando crea una voce solo nel .git/config di QUESTO
    progetto
git config --local user.email "nome.cognome@azienda.com"
```

Risultato:

- In questo progetto: verrà usata `@azienda.com`
- In tutti gli altri progetti: si continuerà ad usare `@gmail.com`

1.2.3 Configurazione Iniziale

È necessario configurare l'identità dell'utente prima di effettuare il primo commit. Queste impostazioni sono solitamente applicate a livello globale.

Codice 1.2: Setup iniziale obbligatorio

```
# 1. Identità (Visibile nei log dei commit)
git config --global user.name "Nome Cognome"
git config --global user.email "email@esempio.com"

# 2. Editor di testo predefinito (es. Nano o VS Code)
git config --global core.editor nano

# 3. Definizione del ramo predefinito (Standard attuale: main)
git config --global init.defaultBranch main

# 4. Output colorato CLI
git config --global color.ui auto
```

Verifica Configurazioni

Per visualizzare tutte le configurazioni attive e la loro origine:

```
git config --list --show-origin
```

1.3 Struttura Interna: La Directory .git

Quando si inizializza un repository con `git init`, Git non fa altro che creare una directory nascosta denominata `.git`. Questa directory è il repository. La cartella di lavoro visibile (Working Directory) è solo un checkout temporaneo di una specifica versione contenuta nel database interno.

> Eliminare la cartella `.git` comporta la perdita irreversibile di tutta la cronologia del progetto, dei branch e delle configurazioni locali. Rimarranno solo i file nello stato corrente (snapshot attuale).

1.3.1 Struttura del File System

Di seguito è riportata la struttura standard della directory, con la descrizione dei componenti critici.

Codice 1.3: Struttura della directory .git

```
.git/
|-- HEAD            # Puntatore al branch corrente (riferimento
   simbolico)
|-- config          # Configurazioni specifiche del repository (Livello
   Local)
|-- index           # File binario che rappresenta la Staging Area
|-- objects/        # Il Database vero e proprio (Content-Addressable)
|   |-- info/
|   `-- pack/       # File compressi (per efficienza dello spazio)
|-- refs/           # Puntatori ai commit (Riferimenti)
|   |-- heads/      # Contiene i puntatori ai Branch locali
|   `-- tags/       # Contiene i puntatori ai Tag (versioni fisse)
`-- hooks/          # Script di automazione (pre-commit, post-merge, ecc
   .)
```

1.3.2 Componenti Chiave

1. Objects: Il Database Content-Addressable

Git non salva i file in base al nome, ma in base al contenuto. Ogni oggetto viene compresso (zlib) e identificato da un hash **SHA-1** di 40 caratteri. Esistono tre tipi principali di oggetti nella cartella `objects/`:

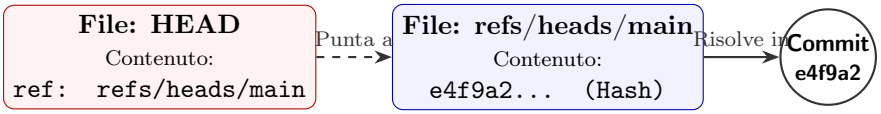
- **Blob (Binary Large Object):** Contiene solo il contenuto del file (senza nome o metadati). Se due file hanno lo stesso contenuto, condividono lo stesso Blob (deduplicazione).
- **Tree:** Rappresenta una directory. Associa i nomi dei file ai rispettivi Blob e le sottocartelle ad altri Tree.
- **Commit:** Contiene il puntatore al Tree principale (snapshot), il puntatore al commit genitore, l'autore, la data e il messaggio.

2. Refs: I Puntatori

I file contenuti in `refs/heads/` sono semplici file di testo. Se si apre il file `refs/heads/main`, si troverà al suo interno solo una stringa SHA-1 (es. `a1b2c3d...`). Questo hash indica l'ultimo commit di quel ramo. Spostare un branch in Git è un'operazione istantanea perché consiste solo nel sovrascrivere questo file di testo con un nuovo hash.

3. HEAD: Il Puntatore Simbolico

Il file `HEAD` determina la propria posizione attuale nel repository. Solitamente contiene un riferimento a un branch.



1.3.3 La Staging Area (Index)

Il file `index` è un file binario dinamico che agisce da zona cuscinetto tra la Working Directory e il Repository (Objects). Quando si esegue `git add`, Git:

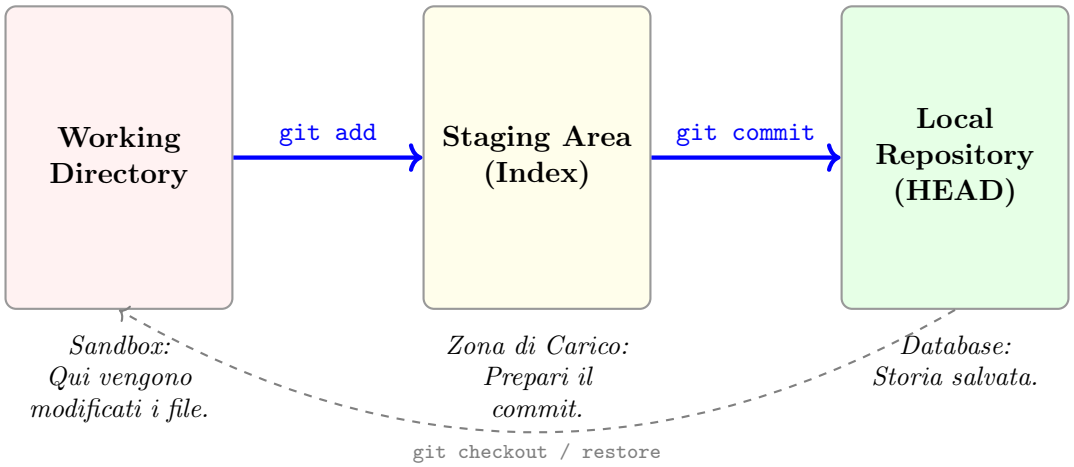
1. Comprime il file e crea un oggetto **Blob** nel database `objects/`.
2. Aggiorna il file `index` con il riferimento a quel Blob.

Il comando `git commit` trasforma successivamente lo stato attuale dell'`index` in un oggetto **Commit** permanente.

1.4 Il Ciclo di Vita dei File

Per lavorare con Git è indispensabile comprendere come i file si spostano tra le tre aree logiche principali. Ogni comando di Git serve a spostare dati da un'area all'altra.

1.4.1 Le Tre Aree di Lavoro



1.4.2 Fase 1: Lo Staging (git add)

Il comando `git add` non serve solo ad "aggiungere file nuovi". Serve a comunicare a Git: *"Prendi il contenuto di questo file in questo preciso istante e mettilo nell'Index"*.

Concetto Chiave

Git non salva file, salva **contenuti**. Se si modifica un file **dopo** aver eseguito `git add`, sarà necessario eseguire nuovamente `git add` per aggiornare la Staging Area con le ultime modifiche.

Sintassi e Comandi

Codice 1.4: Varianti di git add

```
# Aggiunge un file specifico
git add nomefile.py

# Aggiunge TUTTO (file nuovi, modificati e cancellati) nella cartella
    corrente
git add .

# Aggiunge interagendo (patch mode) - PER ESPERTI
# Permette di scegliere quali righe di un file aggiungere
git add -p
```

1.4.3 Stato dei File

Un file nella Working Directory può trovarsi in 4 stati. È fondamentale saper leggere l'output di `git status`.

- **Untracked:** Il file esiste ma Git non lo osserva (nuovo file).
- **Tracked:** Git conosce il file. Si divide in:

- **Unmodified:** Identico all'ultimo commit.
- **Modified:** Modificato ma non ancora in Staging (Rosso).
- **Staged:** Modificato e aggiunto all'Index, pronto per il commit (Verde).

Codice 1.5: Esempio di git status

```
$ git status
On branch main

# File modificato ma non nell'Index (Rosso)
Changes not staged for commit:
  modified:   main.py

# File pronto per il commit (Verde)
Changes to be committed:
  new file:   utils.py
```

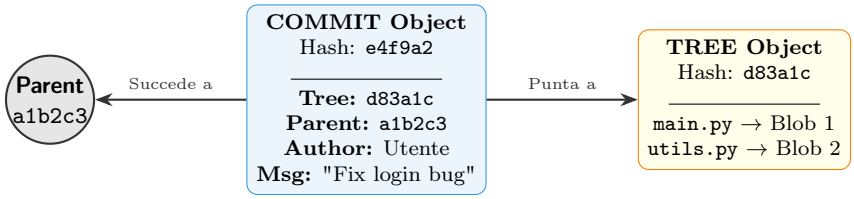
1.5 Il Commit

Il commit è l'operazione che trasforma il contenuto della Staging Area in uno snapshot permanente nel database di Git. Ogni commit è un oggetto immutabile. Una volta creato, non può essere modificato; qualsiasi operazione che sembra modificarlo (come `amend`) in realtà ne crea uno nuovo e scarta il vecchio.

1.5.1 Struttura di un Oggetto Commit

Un commit non contiene direttamente i file. Contiene metadati e puntatori. Ecco cosa viene scritto su disco (compresso) quando si digita `git commit`:

1. **Tree Hash:** Il puntatore alla radice del progetto (lo snapshot dei file).
2. **Parent Hash:** Il puntatore al commit (o ai commit) precedente/i.
3. **Author:** Chi ha scritto il codice e quando.
4. **Committer:** Chi ha effettivamente inserito il commit nel repo (utile se diverso dall'autore, es. rebase o patch).
5. **Message:** Il testo descrittivo.



1.5.2 1. Atomicità

> Un commit deve fare **una e una sola cosa**.

È buona prassi non mescolare la correzione di un bug critico con modifiche stilistiche (es. indentazione) di altri file. Se un commit compromette la stabilità (build), deve essere possibile rimuoverlo agevolmente senza perdere modifiche non correlate. Si consiglia l'uso di `git add -p` per separare le modifiche logiche.

1.5.3 2. Il Messaggio (Imperative Mood)

Lo standard internazionale per i messaggi di commit è il **Modo Imperativo Presente**. Git stesso usa questo tono nei messaggi automatici (es. "Merge branch...").

- **Corretto:** "Add inventory function" (Come se fosse un comando: "Aggiungi funzione...")
- **Comunemente usato:** "Added inventory" (Passato)
- **Errato:** "Inventory function code" (Vago)

3. Conventional Commits

Molte aziende adottano lo standard *Conventional Commits* per automatizzare il changelog. Struttura: `<tipo>: <descrizione>`

- `feat:` Nuova funzionalità

- `fix:` Correzione bug
- `docs:` Solo documentazione
- `chore:` Manutenzione (es. aggiornamento librerie)
- `refactor:` Modifica codice senza cambiare logica

1.5.4 Comandi Operativi

Codice 1.6: Comandi di Commit

```
# Standard (apre l'editor di testo per il messaggio)
git commit

# Shortcut (messaggio in linea - solo per messaggi brevi)
git commit -m "feat: implement user login"

# Skip Staging (Aggiunge e committa file TRACCIATI in un colpo solo)
# ATTENZIONE: Non include i file nuovi (Untracked)
git commit -a -m "fix: typo in readme"
```

1.5.5 Correggere l'ultimo Commit (–amend)

Può capitare di effettuare un commit dimenticando di aggiungere un file, oppure inserendo un messaggio errato.

Codice 1.7: Modificare l'ultimo commit

```
# 1. Aggiungere il file dimenticato
git add file_dimenticato.txt

# 2. Riscrivere l'ultimo commit includendo il nuovo file
git commit --amend --no-edit
# L'hash del commit cambierà!
```

Attenzione: Riscrivere la Storia

Non utilizzare mai `-amend` su commit già inviati a un repository remoto (Push). Questa operazione crea conflitti divergenti per gli altri collaboratori. Si raccomanda di usare `amend` esclusivamente per commit ancora locali.

1.6 Git Status: Analisi dello Stato Corrente

Il comando `git status` rappresenta il pannello di controllo principale per l'operatore. A differenza di altri comandi che agiscono sulla storia passata, questo comando fornisce uno snapshot delle differenze tra le tre aree logiche: Working Directory, Staging Area e HEAD.

1.6.1 Logica di Comparazione

Quando viene invocato, Git esegue due comparazioni simultanee:

1. **HEAD vs Staging Area:** Determina quali modifiche sono state pianificate per il prossimo commit (Changes to be committed).
2. **Staging Area vs Working Directory:** Determina quali modifiche sono state apportate ai file ma non ancora marcate per l'inclusione (Changes not staged).

Inoltre, scansiona la directory corrente alla ricerca di file che non esistono nel database (Untracked files).

1.6.2 Interpretazione dell'Output Standard

L'output predefinito è verboso e descrittivo, pensato per guidare l'utente nelle azioni successive.

Codice 1.8: Esempio di output verboso

```
$ git status
On branch main

# 1. Area Staging (Verde)
# Questi file entreranno nel prossimo commit
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   core.py

# 2. Working Directory (Rosso)
# Modifiche rilevate ma non ancora aggiunte
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
        modified:   utils.py

# 3. Non Tracciati (Rosso)
# File esistenti su disco ma ignoti a Git
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        temp_test.log
```

1.6.3 Esclusione dei File (.gitignore)

In un progetto reale, vengono generati file che non devono essere versionati (es. file di log, build artifacts, cartelle `bin/`, file di configurazione locali dell'IDE). Per escluderli sistematicamente, si utilizza il file speciale `.gitignore` nella radice del progetto.

Regole di Matching

Git ignora un file **solo se non è già tracciato**. Se un file è già stato committato in passato, aggiungerlo al .gitignore non avrà effetto finché non verrà rimosso dalla cache (`git rm -cached`).

1.6.4 Rimozione dal Versionamento (Un-track)

Una situazione frequente in ambito operativo riguarda i file che sono stati erroneamente aggiunti al repository (es. file di configurazione `.env` contenenti chiavi segrete, o cartelle di build `dist/`) e che successivamente si decide di ignorare. Come specificato in precedenza, aggiungere tali file al `.gitignore` è inefficace se essi sono già presenti nell'Index (stato *Tracked*).

Per rettificare questa condizione senza perdere i dati locali, si utilizza il flag `-cached`.

Distinzione

- `git rm <file>`: Rimuove il file dal versionamento **E** lo cancella fisicamente dal disco (Working Directory).
- `git rm --cached <file>`: Rimuove il file solo dal versionamento (Index). Il file rimane presente su disco ma diventa *Untracked*.

Nel pratico

Per smettere di tracciare un file ma conservarlo per l'uso locale, la sequenza corretta è:

Codice 1.9: Workflow per ignorare file già tracciati

```
# 1. Rimuovere il file dall'indice (diventa Untracked)
git rm --cached config.json

# 2. Aggiornare il .gitignore per prevenire future aggiunte
echo "config.json" >> .gitignore

# 3. Committare la rimozione (il file sparisce dal repo, resta sul PC)
git add .gitignore
git commit -m "chore: stop tracking config.json"
```

Questa operazione è fondamentale per la "sanitizzazione" del repository quando dati sensibili o binari pesanti sono stati committati per errore.

Codice 1.10: Esempio di file .gitignore

```
# Ignora tutti i file con estensione .log
*.log

# Ignora la directory node_modules ovunque si trovi
node_modules/

# Ignora i file di configurazione dell'ambiente locale
.env

# Eccezione: Ignora tutti i .txt tranne questo
*.txt
!important.txt
```

1.6.5 Short Format (Output)

L'output standard è spesso troppo lungo. L'opzione `-s` (o `-short`) fornisce una sintassi compatta a due colonne, simile a quella dei vecchi sistemi SVN.

- **Colonna Sinistra:** Stato della Staging Area (Verde).
- **Colonna Destra:** Stato della Working Directory (Rosso).

Codice	Significato	Descrizione
??	Untracked	File non tracciato (nuovo)
A	Added	File aggiunto all'index (nuovo)
M	Modified	Modificato nel WD, non nell'Index
M	Modified	Modificato nell'Index, pulito nel WD
MM	Mod + Mod	Modificato nell'Index E ulteriormente nel WD
D	Deleted	Cancellato nel WD, ancora presente nell'Index

Codice 1.11: Esempio reale di Short Status

```
$ git status -s
 M main.py     # Modificato nel WD (non staged)
M  utils.py    # Modificato nell'Index (staged)
MM api.py      # Staged, ma poi modificato ANCORA
?? test.log    # Untracked
```

Questa modalità permette di avere una visione d'insieme immediata dello stato di decine di file senza dover scorrere pagine di output verboso.

1.7 I Tre Alberi

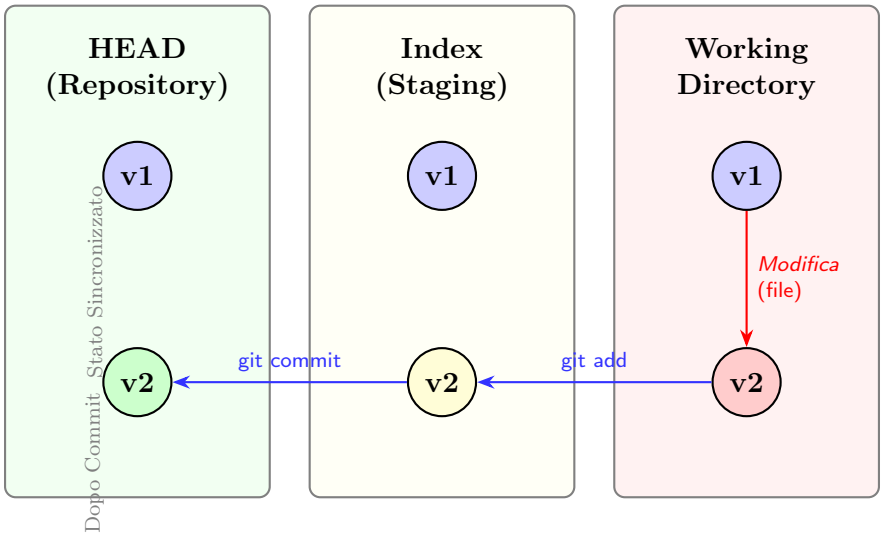
Per padroneggiare Git, è necessario abbandonare l'idea generica di salvataggio e adottare il modello mentale dei **Tre Alberi**. In Git, un albero (tree) è una collezione di file e directory in un determinato stato. Le operazioni principali ('add', 'commit', 'checkout', 'reset') non fanno altro che spostare dati tra questi tre alberi.

1.7.1 Definizione delle Entità

1. **HEAD (Ultimo Commit):** È il puntatore al ramo corrente. Rappresenta lo stato del progetto "congelato" nell'ultimo snapshot salvato. È la base di confronto per vedere cosa è cambiato.
2. **Index (Staging Area):** È un albero *provvisorio*. Non contiene i file fisici, ma una lista di riferimenti (hash) ai contenuti che si intende includere nel prossimo commit.
3. **Working Directory (Sandbox):** È l'unica area in cui i file sono scompattati e modificabili dall'utente. È volatile: le modifiche qui non esistono per Git finché non vengono promosse all'Index.

1.7.2 Visualizzazione del Flusso di Modifica

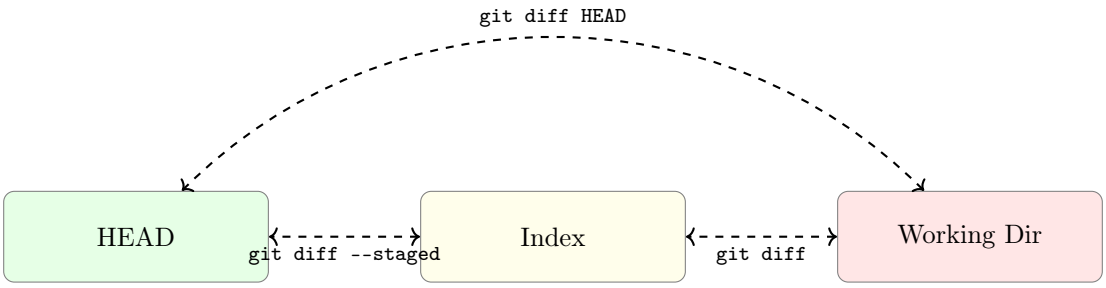
Analizziamo cosa accade a un singolo file (`file.txt`) durante un ciclo di modifica. Si noti come possano esistere contemporaneamente tre versioni diverse dello stesso file.



1.7.3 I Delta e il comando Git Diff

Una delle fonti di confusione maggiori riguarda il comando `git diff`. A seconda dei flag, esso confronta alberi diversi.

- `git diff` **(senza argomenti):** Confronta *Working Directory* vs *Index*. Mostra le modifiche apportate ma **non** ancora aggiunte.
- `git diff --staged`**:** Confronta *Index* vs *HEAD*. Mostra cosa è stato aggiunto e si sta per committare.
- `git diff HEAD`**:** Confronta *Working Directory* vs *HEAD*. Mostra tutte le modifiche dall'ultimo commit (sia staged che unstaged).



Caso d'Uso: file "Doppiamente Modificato"

È possibile che un file appaia sia in "Changes to be committed" (Verde) che in "Changes not staged" (Rosso). **Significato:** Esistono 3 versioni del file.

- **v1 (HEAD):** Versione originale salvata.
- **v2 (Index):** Una modifica aggiunta con `git add`.
- **v3 (WorkDir):** Un'ulteriore modifica apportata *dopo* l'add.

Eseguendo il commit in questa fase, verrà salvata la v2, non la v3.

1.8 Gestione delle Ramificazioni (Branching)

Il sistema di branching di Git è considerato una delle feature principali. A differenza di sistemi centralizzati dove un branch è una copia fisica della directory, in Git un branch è un'entità estremamente leggera.

1.8.1 Definizione Tecnica

Un branch in Git non è altro che un **puntatore mobile** a uno specifico commit. Tecnicamente, un branch è un semplice file di testo situato in `.git/refs/heads/` contenente l'hash di 40 caratteri dell'ultimo commit effettuato su quella linea di sviluppo.

Creare un branch, quindi, è un'operazione istantanea (costo O(1)), indipendentemente dalla dimensione del progetto, poiché consiste nello scrivere 41 byte su disco.

1.8.2 In Pratica...

Dal punto di vista operativo, un branch rappresenta un **ambiente di lavoro isolato**. Quando si crea un branch, si sta effettivamente dicendo a Git: *"Voglio divergere dalla linea principale per lavorare su qualcosa di specifico, senza che le mie modifiche interferiscano con il codice stabile o con il lavoro degli altri colleghi."*

L'Effetto sulla Working Directory

È fondamentale comprendere che Git gestisce i branch modificando fisicamente i file nella tua cartella di lavoro. Quando si cambia branch (operazione di `switch`), Git:

1. Rimuove i file che esistono nel vecchio branch ma non nel nuovo.
2. Ripristina i file che esistono nel nuovo branch ma non nel vecchio.
3. Aggiorna il contenuto dei file che esistono in entrambi ma con versioni diverse.

In pratica, la cartella di progetto si trasforma istantaneamente per riflettere lo stato di quel branch.

Scenari di Utilizzo Professionali

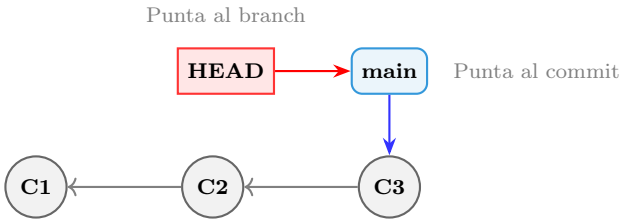
Nello sviluppo software moderno, i branch sono utilizzati principalmente per tre scopi:

- **Feature Branching:** Ogni nuova funzionalità (es. "Aggiungi carrello e-commerce") viene sviluppata in un ramo dedicato. Il codice `main` rimane pulito e compilabile mentre la feature è ancora "work in progress".
- **Hotfix (Correzioni Rapide):** Si immagini di lavorare su una feature complessa (branch `feature-x`) quando viene segnalato un bug critico in produzione.
  - Con i branch, si può salvare il proprio lavoro corrente, tornare al ramo `main`, creare un branch `hotfix`, correggere il bug, e poi ritornare sulla feature come se nulla fosse successo.
- **Experimentation:** Se si volesse provare a rifattorizzare una libreria senza sapere se funzionerà, si crea un branch. Se si fallisce, è possibile cancellare il branch e il puntatore sparisce. Nessun Undo necessario.

> Il branch `main` (o `master`) deve essere sempre **stabile** e dispiegabile in produzione (Deployable). Non si committa mai codice non testato o incompleto direttamente su `main`. Si lavora sui branch e si unisce (Merge) solo quando il lavoro è finito e verificato.

1.8.3 Il Ruolo di HEAD

Per capire dove ci troviamo, Git utilizza un puntatore speciale chiamato **HEAD**. Di norma, HEAD non punta direttamente a un commit, ma punta al nome del branch corrente. Quando viene eseguito un nuovo commit, il branch puntato da HEAD avanza automaticamente.



Interpretazione delle Decorazioni (Log Decorate)

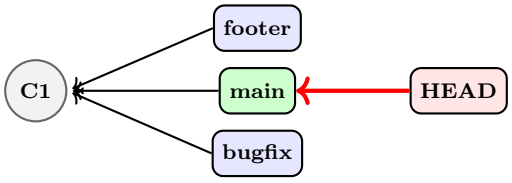
Quando si visualizza la storia (es. con `git log --oneline` (che verrà approfondito successivamente)), Git mostra tra parentesi i puntatori associati a ciascun commit. Una sintassi comune e spesso fraintesa è la seguente:

```
(HEAD -> main, footer, bugfix)
```

Questa riga fornisce tre informazioni distinte e precise sulla topologia del repository in quel momento:

1. **HEAD -> main:** Il puntatore `HEAD` è attualmente agganciato al branch `main`. Questo è il branch *Attivo*. Eseguendo un commit ora, solo `main` avanzerà.
2. **, footer, bugfix:** Anche i branch `footer` e `bugfix` puntano a questo stesso commit. Sono branch *Passivi*.
3. **Coincidenza:** I tre branch sono perfettamente sincronizzati (identici).

Questo accade tipicamente quando si creano nuovi branch senza ancora committarci sopra (es. `git branch footer`), oppure subito dopo un *Fast-Forward Merge*.



Nell'esempio visivo sopra, se l'utente esegue un commit, **solo** il rettangolo `main` (e di conseguenza `HEAD`) si sposterà verso destra (C2). I puntatori `footer` e `bugfix` rimarranno ancorati a C1.

1.8.4 Creazione e Cambio (Switching)

Fino alle versioni recenti di Git, il comando `checkout` veniva usato sia per cambiare branch che per ripristinare file. Per disambiguare, Git 2.23 ha introdotto `git switch`.

Comandi Operativi

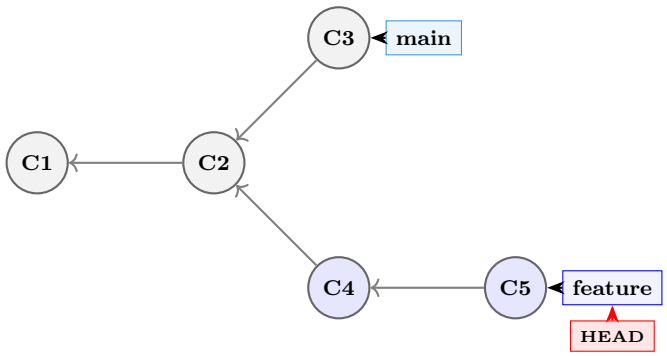
Codice 1.12: Gestione dei Branch

```
# Creare un nuovo branch (senza spostarsi)
git branch feature-login

# Spostarsi sul nuovo branch (aggiorna HEAD e Working Directory)
git switch feature-login
# (Vecchio comando: git checkout feature-login)

# Creare E spostarsi in un colpo solo (Best Practice)
git switch -c feature-login
# (Vecchio comando: git checkout -b feature-login)
```

1.8.5 Divergenza della Storia

Quando si lavora su un branch separato e si effettuano commit, la storia diverge. Se nel frattempo vengono effettuati commit anche sul branch principale (`main`), si crea una biforcazione nel grafo orientato aciclico (DAG).

Git gestisce questa complessità mantenendo puntatori distinti per ogni ramo.



Nello schema sopra:

- Il branch `main` è fermo al commit **C3**.
- Il branch `feature` è avanzato fino a **C5**.
- `HEAD` punta a `feature`, quindi la Working Directory contiene i file dello stato C5.
- I due rami condividono la storia fino a **C2** (Antenato Comune).

1.8.6 Gestione e Pulizia

I branch "feature" sono solitamente temporanei. Una volta terminato il lavoro (e integrato tramite merge), il puntatore deve essere rimosso per mantenere pulito il repository.

Codice 1.13: Cancellazione Branch

```
# Lista di tutti i branch locali (* indica il corrente)
git branch -v

# Cancellazione sicura (Safe Delete)
# Git impedisce la cancellazione se il branch non è stato ancora
    mergiato
git branch -d feature-login
```

```
# Cancellazione forzata (Force Delete)
# Necessaria se si vuole buttare via il lavoro senza integrarlo
git branch -D feature-login
```

1.8.7 Detached HEAD (Testa Distaccata)

Uno degli stati più temuti e meno compresi dagli sviluppatori alle prime armi è il cosiddetto *Detached HEAD*. Per comprenderlo, è necessario ricordare la regola aurea della navigazione in Git:

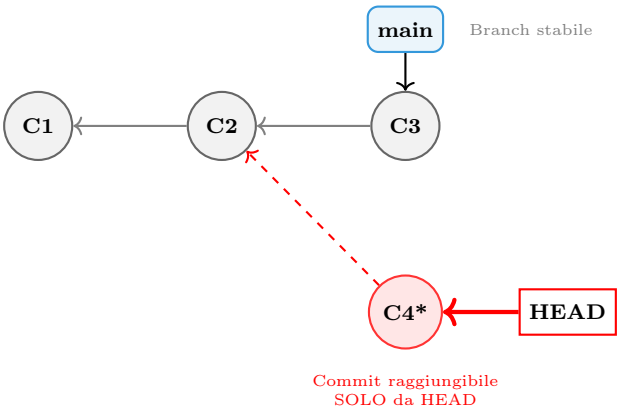
> **Normalmente**, HEAD punta a un nome di branch, e quel branch punta a un commit.

Quando si esegue un comando come `git checkout <hash>` (o `git switch -detach <hash>`), si forza HEAD a puntare **direttamente** a un commit, bypassando il branch. In questo momento, HEAD è "distaccato" da qualsiasi riferimento mobile.

Struttura di un Commit Orfano

Il pericolo non risiede nel *guardare* un vecchio commit (operazione sicura in sola lettura), ma nel **produrre nuovi commit** in questo stato.

Se si crea un commit mentre si è in *Detached HEAD*, il nuovo commit avrà un genitore, ma nessun branch che punta ad esso. L'unico riferimento che lo tiene "vivo" è HEAD stesso.



Nello schema sopra:

1. L'utente è tornato al commit **C2**.
2. Ha creato un nuovo commit **C4***.
3. Il branch `main` è rimasto fermo a **C3**.
4. **Il Rischio:** Se ora l'utente esegue `git switch main`, HEAD si sposterà su `main`. Il commit **C4*** non avrà più nessuno che punta ad esso. Diventerà "irraggiungibile" e verrà eliminato dal *Garbage Collector* di Git durante la manutenzione automatica.

Procedure di Recupero

Lavorare in *Detached HEAD* è utile per esperimenti rapidi. Tuttavia, se si decide di mantenere il lavoro svolto (il commit C4*), è necessario creare un branch che punti ad esso **prima** di spostarsi altrove.

Codice 1.14: Salvataggio di un Detached HEAD

```
# Situazione: ci si trova su (C4*)
$ git status
HEAD detached from 8f9a2b
nothing to commit, working tree clean

# Come soluzione, si crea un branch nella posizione corrente
git switch -c experiment-fix

# Risultato:
# Ora esiste un branch "experiment-fix" che punta a C4*.
# HEAD punta a "experiment-fix".
# Lo stato "Detached" è risolto.
```

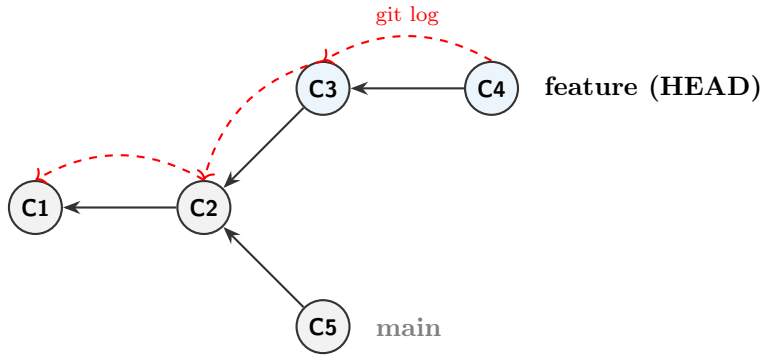
Se invece ci si è già spostati su `main` e si realizza di aver "perso" il commit C4*, l'unico modo per recuperarlo è utilizzare il comando `git reflog` per ritrovare l'hash perduto e creare un branch su di esso.

1.9 `git log`

La capacità di navigare ed estrarre informazioni dalla cronologia del progetto è fondamentale per il debugging e l'auditing del codice. Il comando `git log` non si limita a stampare a video i commit, ma esegue un attraversamento del Grafo Orientato Aciclico (DAG) partendo da un puntatore specifico (solitamente `HEAD`) e risalendo verso i genitori.

1.9.1 Reachability (Raggiungibilità)

Git mostra solo i commit "raggiungibili" dal punto di partenza. Se ci si trova su un branch laterale, `git log` non mostrerà i commit di `main` successivi alla divergenza, a meno che non vengano richiesti esplicitamente.



Il commit C5 viene ignorato
perché non è antenato di HEAD

1.9.2 Visualizzazione Topologica

Il formato predefinito è spesso troppo verboso per una visione d'insieme. Per analizzare la struttura delle ramificazioni e dei merge, si utilizzano flag specifici.

Codice 1.15: Visualizzazione del Grafo

```
# Visualizzazione compatta con grafo ASCII
git log --graph --oneline --all

# --graph: Disegna le linee di connessione tra i commit
# --oneline: Riduce ogni commit a una riga (Hash ridotto + Titolo)
# --all: Mostra TUTTI i branch e i tag, non solo quello corrente
# --decorate: Mostra i nomi dei branch e dei tag accanto agli hash
```

Alias

Per evitare di digitare comandi lunghi, è prassi configurare un alias globale:

`git config --global alias.lg "log --graph --oneline --all"`

Utilizzo: `git lg`

1.9.3 Filtraggio dei Dati (Data Mining)

In progetti con migliaia di commit, è necessario filtrare l'output per trovare informazioni specifiche.

1. Filtraggio per Quantità e Tempo

Codice 1.16: Filtri temporali e numerici

```
# Ultimi 5 commit
git log -n 5

# Commit effettuati nelle ultime 2 settimane
git log --since="2 weeks ago"

# Commit effettuati entro una data specifica
git log --until="2023-01-01"
```

2. Filtraggio per Contenuto

Questa è una delle funzionalità più potenti e meno conosciute. Permette di individuare **quando** una determinata stringa di codice è stata aggiunta o rimossa. Utile per trovare "chi ha introdotto questo bug" o "quando è stata rimossa questa funzione".

Codice 1.17: Ricerca nel codice sorgente

```
# Trova tutti i commit che hanno aggiunto o rimosso la stringa "
    password_hash"
git log -S "password_hash"

# Come sopra, ma accetta RegEx (Espressioni Regolari)
git log -G "^def login\("
```

3. Filtraggio per Percorso

Per visualizzare la storia di un singolo file o di una directory.

Codice 1.18: Storia specifica di un modulo

```
# Storia completa solo del file main.py
git log -- main.py

# Combinazione: Chi ha modificato main.py nel 2024?
git log --since="2024-01-01" --author="Rossi" -- main.py
```

Decodifica dei Simboli (Unified Diff)

Il formato di visualizzazione standard...

Codice 1.19: Esempio di output statistico

```
$ git log --stat -n 1
commit e4f9a2...
Author: Mario Rossi <m.rossi@email.com>
Date:   Mon Oct 12 14:00:00 2024 +0200

    feat: implement login system

 main.py  | 20 ++++++++++++++++++++
 utils.py |  5 +++++
 2 files changed, 25 insertions(+)
```

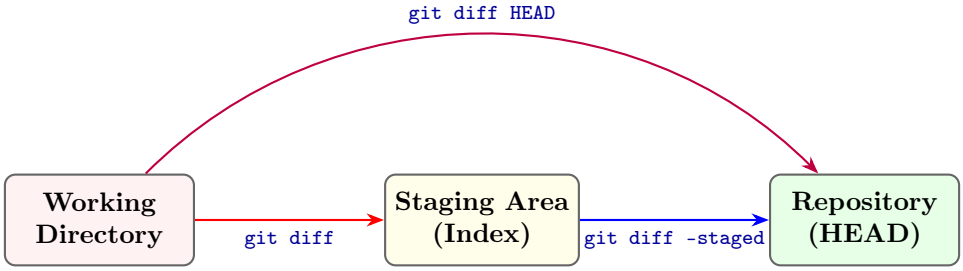
1.9.4 Git diff - Analisi delle Differenze)

Il comando `git diff` è un motore di comparazione testuale. La sua funzione non è limitata a mostrare "le ultime modifiche scritte", ma permette di calcolare il delta (le differenze) tra due qualsiasi entità del database di Git:

- Due commit arbitrari nella storia.
- Due rami (branch) divergenti.
- Lo stato attuale dei file su disco (Working Directory) rispetto all'ultimo commit o all'area di staging.

La Topologia

Per comprendere quale output verrà mostrato, bisogna sapere quali "aree" vengono messe a confronto.



Scenari di Comparazione

Oltre al controllo quotidiano delle modifiche locali, `git diff` è essenziale per l'auditing della storia e la revisione del codice tra rami.

1. Comparazione Storica (Commit vs Commit) Per analizzare l'evoluzione di un progetto (o di un singolo file) tra due punti specifici nel tempo:

Codice 1.20: Analisi differenziale tra commit

```
# Sintassi: git diff <old-sha> <new-sha>
git diff 8a2b3c 9f1d2e

# Focus su un singolo file (scelta consigliata per code review)
git diff 8a2b3c 9f1d2e -- src/main.js
```

2. Comparazione tra Rami (Branch vs Branch) Prima di effettuare un merge, è buona norma verificare quali modifiche il branch `feature` apporterà a `main`:

Codice 1.21: Analisi differenziale tra branch

```
# Mostra la differenza tra la punta (tip) dei due branch
git diff main feature-login

# Mostra solo i nomi dei file che cambiano (utile per overview)
git diff --name-only main feature-login
```

Esempi di Output Reali

Di seguito analizziamo l'output grezzo del comando per imparare a leggerne la sintassi nascosta.

Scenario A: Modifica di una riga esistente Questo è il caso più comune: si cambia il valore di una variabile o la logica di una funzione.

Codice 1.22: Output di una modifica standard

```
diff --git a/config.js b/config.js
index 5f2a1b..8c3d4e 100644
--- a/config.js
+++ b/config.js
@@ -10,4 +10,4 @@ function init() {
     console.log("System Start");
-    const max_retries = 3;
+    const max_retries = 5;
     connect_db();
 }
```

- **Prime due righe:** Metadati tecnici (hash dei file blob prima e dopo).
- `@@ -10,4 +10,4 @@`: Questo è l'**Hunk Header** (Intestazione del Blocco).
  - `-10,4`: Nel file originale, il blocco mostrato inizia alla riga 10 ed è lungo 4 righe.
  - `+10,4`: Nel file nuovo, il blocco inizia alla riga 10 ed è lungo 4 righe.
- **Contesto:** Le righe bianche (senza + o -) sono il codice circostante che Git mostra per aiutare l'orientamento.

Scenario B: Cancellazione o Creazione di un File Quando un file viene creato o cancellato, Git lo compara con il "nulla" (indicato tecnicamente come `/dev/null`).

Codice 1.23: Output di cancellazione file

```
diff --git a/old_script.py b/old_script.py
deleted file mode 100644
index 9a8b7c..0000000
--- a/old_script.py
+++ /dev/null
@@ -1,3 +0,0 @@
-print("Questo script è deprecato")
-exit(1)
```

- `+++ /dev/null`: Indica che il file di destinazione non esiste (il file è stato cancellato).
- `@@ -1,3 +0,0 @@`: Il file originale aveva 3 righe (visualizzate in rosso), il nuovo file ha 0 righe.

Best Practice: –color-words

Se le modifiche sono minime (es. correzione di un typo su una riga molto lunga), il diff standard può essere difficile da leggere. Utilizzando il flag:

```
git diff --color-words
```

Git non mostrerà le righe intere rimosse/aggiunte, ma evidenzierà solo le singole parole cambiate sulla stessa riga.

Decodifica dei Simboli (Unified Diff)

Il formato di visualizzazione standard (detto *Unified Diff*) utilizza i simboli + e -. È fondamentale comprendere che questi non indicano necessariamente un'aggiunta o una rimozione logica, ma chiamati Originale e Nuovo):

- `-` **(Meno):** Contenuto presente nel file *Originale* (spesso indicato come `a/`) che non esiste più nel file *Nuovo*.
- `+` **(Più):** Contenuto presente nel file *Nuovo* (spesso indicato come `b/`) che non esisteva nel file *Originale*.

Si ricorda che git ragiona per righe intere. Pertanto, la **modifica** di una singola parola viene rappresentata tecnicamente come la **distruzione completa** della vecchia riga e la **creazione** di una nuova riga con il contenuto aggiornato.

Codice 1.24: Esempio di Unified Diff

```
diff --git a/config.py b/config.py
index 8f9a..2b3c 100644
--- a/config.py  (Versione Precedente)
+++ b/config.py  (Versione Attuale)
@@ -5,3 +5,3 @@
 def connect():
-    timeout = 10  <-- Questa riga esisteva in A, non esiste in B
+    timeout = 30  <-- Questa riga esiste in B, non esisteva in A
     return True
```

1.10 `git merge`

L'operazione di *Merge* (unione) è il meccanismo fondamentale attraverso il quale storie divergenti vengono riconciliate. Dal punto di vista teorico, il merge non modifica la storia passata dei branch coinvolti, ma ne crea una convergenza.

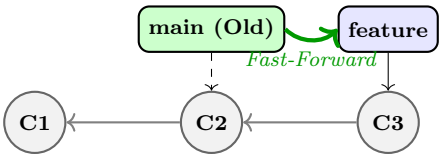
Il comando `git merge <branch-sorgente>` integra la storia del branch specificato nel branch corrente (`HEAD`). Git adotta strategie differenti in base alla topologia del grafo al momento dell'esecuzione.

1.10.1 Strategia 1: Fast-Forward

Questa è la modalità di integrazione più semplice ed efficiente. Si verifica quando esiste un percorso lineare diretto dal branch corrente al branch che si desidera unire.

Condizione Topologica: Il branch di destinazione (es. `main`) non ha subito modifiche rispetto all'antenato comune con il branch sorgente (es. `feature`). In termini di teoria dei grafi, il tip di `main` è un antenato diretto del tip di `feature`.

Azione del Sistema: Git non crea nessun nuovo commit di unione. Si limita a spostare il puntatore `HEAD` in avanti (fast-forward) per farlo coincidere con il puntatore del branch sorgente.



L'operazione è istantanea e non lascia traccia del fatto che i branch fossero separati ("linearizzazione della storia").

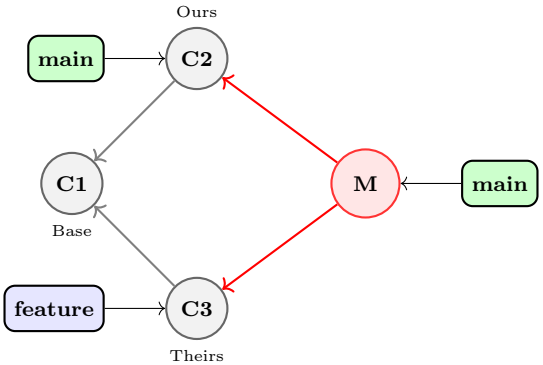
1.10.2 Strategia 2: Three-Way Merge (Merge Commit)

Questa strategia viene invocata quando la storia ha subito una biforcazione reale. Se entrambi i rami hanno registrato nuovi commit a partire dal loro antenato comune, non esiste un percorso lineare.

Algoritmo: Git deve considerare tre stati distinti per calcolare il risultato finale:

1. **Base:** L'antenato comune più recente (Lowest Common Ancestor).
2. **Theirs:** L'ultimo commit del branch che si sta integrando.
3. **Ours:** L'ultimo commit del branch corrente.

Azione del Sistema: Viene generato un nuovo commit speciale, detto *Merge Commit*. A differenza dei commit standard che hanno un solo genitore, un Merge Commit possiede **due genitori**: l'ultimo commit del ramo corrente e l'ultimo commit del ramo integrato.



Questo grafo a forma di diamante preserva la topologia dello sviluppo parallelo. È possibile forzare questa modalità anche quando sarebbe possibile un Fast-Forward utilizzando il flag `-no-ff`, pratica comune per mantenere traccia storica dell'esistenza di una feature branch.

1.10.3 Gestione dei Conflitti

Un conflitto di merge non è un errore del software, ma una richiesta di intervento umano. Si verifica quando Git, deterministico per natura, incontra una situazione ambigua che non può risolvere algoritmicamente senza rischiare la perdita di dati semantici.

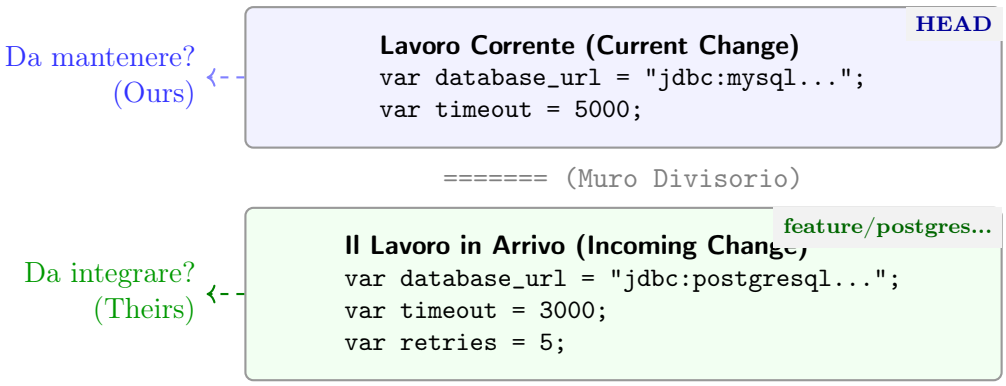
Struttura dei Marcatori di Conflitto

Quando Git rileva un conflitto, modifica fisicamente il contenuto dei file nella Working Directory. È vitale saper leggere questi blocchi per comprendere la provenienza delle modifiche.

Ecco come appare un conflitto reale e come interpretarlo:

```
<<<<<<< HEAD
var database_url = "jdbc:mysql://localhost:3306/prod";
var timeout = 5000;
=======
var database_url = "jdbc:postgresql://remote-server:5432/db";
var timeout = 3000;
var retries = 5;
>>>>>>> feature/postgres-migration
```

Visualmente, Git ha diviso il file in due settori contrapposti. Si può immaginare la struttura in questo modo:



La struttura è sempre composta da tre elementi chiave:

- **Marker di Apertura (`<<<<<<<`):** Indica l'inizio della sezione "HEAD" (ciò che si aveva prima del merge).
- **Separatore (`=======`):** linea divisoria. Non contiene codice, serve solo da confine.
- **Marker di Chiusura (`>>>>>>>`):** Indica la fine della sezione proveniente dal branch esterno (es. `feature/postgres-migration`).

Lo Stato Unmerged dell'Index

Durante un conflitto, il repository entra in uno stato sospeso. Eseguendo `git status`, i file non appaiono né come "staged" né come "modified", ma in una categoria speciale:

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

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

La dicitura `both modified` indica che entrambi i puntatori (HEAD e MERGE_HEAD) hanno apportato modifiche a quel file. In questo stato, il comando `git commit` è bloccato fino alla risoluzione.

"Protocollo" di Risoluzione del conflitto

La procedura corretta per risolvere un conflitto segue rigorosamente questi passaggi:

1. **Analisi:** Aprire il file ed esaminare le due varianti.
2. **Decisione:** Scegliere una delle seguenti opzioni:
   - Mantenere la versione locale (Ours).
   - Accettare la versione in arrivo (Theirs).
   - Creare una sintesi manuale delle due (es. mantenere entrambe le logiche).
3. **Pulizia:** Rimuovere fisicamente i marcatori `<<<<<<<`, `=======` e `>>>>>>>`. Il file deve tornare ad essere codice valido e compilabile.
4. **Mark as Resolved:** Eseguire `git add <file>`.

   Nota: In questo contesto, `git add` non serve solo a mettere il file in staging, ma comunica a Git: "Conflitto risolto manualmente, procedere".
5. **Finalizzazione:** Eseguire `git commit` (senza argomenti). Git aprirà l'editor con un messaggio precompilato "Merge branch ...". Salvare e chiudere per generare il nodo di merge nel grafo.

Merge Abort

Se durante la risoluzione dei conflitti la situazione diventa confusa o si realizza di aver sbagliato branch, è possibile annullare l'intera operazione e tornare allo stato immediatamente precedente al comando di merge:

```
git merge --abort
```

Questo comando ripristina `HEAD`, Index e Working Directory allo stato pre-merge.

1.11 Salvataggio Temporaneo (Stash)

Il meccanismo di *Stashing* risolve un problema operativo frequente: la necessità di cambiare contesto di lavoro (switch branch) mentre la Working Directory si trova in uno stato "dirty", ovvero con modifiche non ancora pronte per essere committate.

Git impedisce il cambio di branch se le modifiche locali entrano in conflitto con i file del branch di destinazione. Lo `stash` agisce come un'area di storage temporaneo (una struttura a pila LIFO - Last In, First Out) dove è possibile "parcheggiare" le modifiche attuali per recuperarle in un secondo momento.

Si immagini la seguente situazione tipica in un ambiente di sviluppo professionale:

1. **Il lavoro in corso:** è in sviluppo una funzionalità complessa sul branch `feature/nuovo-login`. Sono presenti file modificati a metà, variabili non ancora usate e il codice **non compila**.
2. **L'interruzione:** Arriva una segnalazione critica dalla produzione. Un bug bloccante deve essere risolto immediatamente sul branch `main`.
3. **Il Dilemma:**
   - Non si può fare `git commit`: verrebbe salvata nel repository una versione del codice "rotta" (broken build), violando le regole del team.
   - Non si può fare `git switch main`: Git impedisce di cambiare ramo perché le modifiche locali ai file andrebbero in conflitto o verrebbero sovrascritte dalla versione presente in `main`.

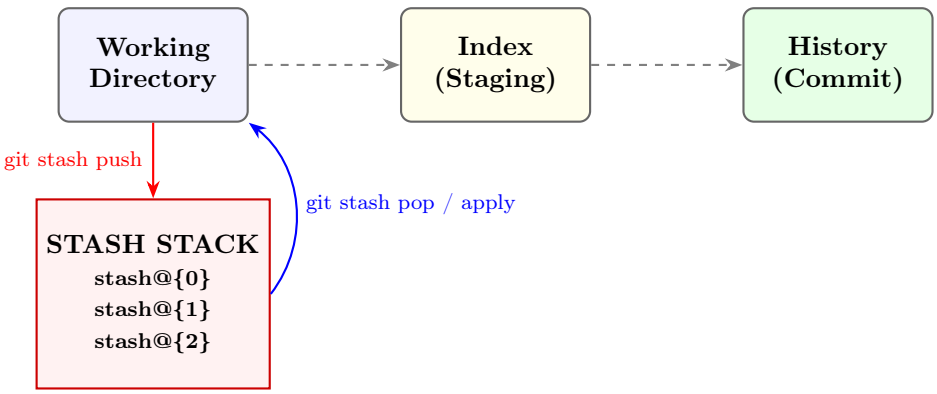
Git blocca l'operazione proteggendo il lavoro con il messaggio:

```
error: Your local changes to the following files would be overwritten by
checkout... Please commit your changes or stash them before you switch.
```

Soluzione: Eseguendo `git stash`, Git "congela" lo stato attuale (Working Directory + Index) e lo sposta in una memoria laterale. Lo spazio di lavoro torna immediatamente pulito (allineato all'ultimo commit), permettendo di passare a `main`, risolvere il bug, e successivamente tornare su `feature` per "scongelare" il lavoro esattamente nello stato in cui era stato lasciato.

1.11.1 Topologia dello Stash

A differenza delle tre aree standard (Working Directory, Index, Repository), lo Stash è un'area laterale indipendente dal branch corrente. Le modifiche salvate nello stash non sono visibili in nessun commit, ma sono salvate localmente nel database '.git'.



1.11.2 Operazioni Fondamentali

1. Creazione di uno Stash (Push)

Il comando base salva lo stato attuale dei file tracciati (modificati o nell'index) e riporta la Working Directory all'ultimo commit pulito (`HEAD`).

Codice 1.25: Salvataggio con messaggio

```
# Forma base (sconsigliata, usa un nome generico)
git stash

# Forma esplicita (CONSIGLIATA: assegna un nome per ritrovarlo)
git stash push -m "WIP: refactoring login logic"
```

Attenzione ai File Non Tracciati

Per default, `git stash` ignora i **nuovi file** (Untracked) che non sono mai stati aggiunti all'index. Per includere anche questi file nel salvataggio, è obbligatorio usare il flag `-u` (o `-include-untracked`).
`git stash push -u -m "Salva tutto inclusi nuovi file"`

2. Consultazione della Pila (List)

Essendo una pila, gli elementi sono indicizzati dal più recente (0) al più vecchio (n).

```
$ git stash list
stash@{0}: On main: WIP: refactoring login logic
stash@{1}: On feature-x: fix rapido css header
stash@{2}: On main: backup prima del pull
```

3. Recupero delle Modifiche (Pop vs Apply)

Esistono due metodi per riapplicare le modifiche parcheggiate. La distinzione è critica per la sicurezza dei dati.

- `git stash pop`: Applica le modifiche dell'ultimo stash (`stash@{0}`) alla Working Directory e, se l'operazione ha successo senza conflitti, **rimuove** lo stash dalla pila.
- `git stash apply`: Applica le modifiche ma **mantiene** una copia nello stash. È l'opzione più sicura se si vuole testare l'applicazione su più branch o se si temono conflitti complessi.

Codice 1.26: Applicazione specifica

```
# Applica l'ultimo stash e lo cancella
git stash pop

# Applica uno stash specifico (senza cancellarlo)
git stash apply stash@{2}
```

1.11.3 Gestione e Pulizia

Lo stash non si pulisce da solo. Accumulare troppi stash rende difficile il recupero.

- **Eliminare uno stash specifico:**

```
git stash drop stash@{1}
```

- **Svuotare l'intera pila (Attenzione!):**

  ```
  git stash clear
  ```

  Questa operazione è irreversibile e distrugge tutte le modifiche salvate.

Conflitti durante il Pop

Se, nel frattempo, il codice è cambiato in modo incompatibile con ciò che è salvato nello stash, `git stash pop` fallirà nel ripristino pulito. In questo caso:

1. Git segnalerà un **Merge Conflict**.
2. Le modifiche verranno applicate parzialmente (con i marcatori `<<<<<`).
3. **Importante:** Lo stash **non** verrà eliminato (il `pop` si trasforma automaticamente in un `apply`) per sicurezza, permettendo di riprovare in caso di errori nella risoluzione manuale.

1.12 Riscritura della Storia (Rebase)

Il comando `git rebase` rappresenta l'alternativa al *merge* per l'integrazione delle modifiche. Mentre il merge preserva la storia esattamente come è avvenuta (incluso il parallelismo temporale), il rebase la **riscrive** per linearizzarla.

Il termine "rebase" significa letteralmente "cambiare la base" di un ramo: si prende un branch che è nato da un certo commit e si simula che sia nato da un commit successivo.

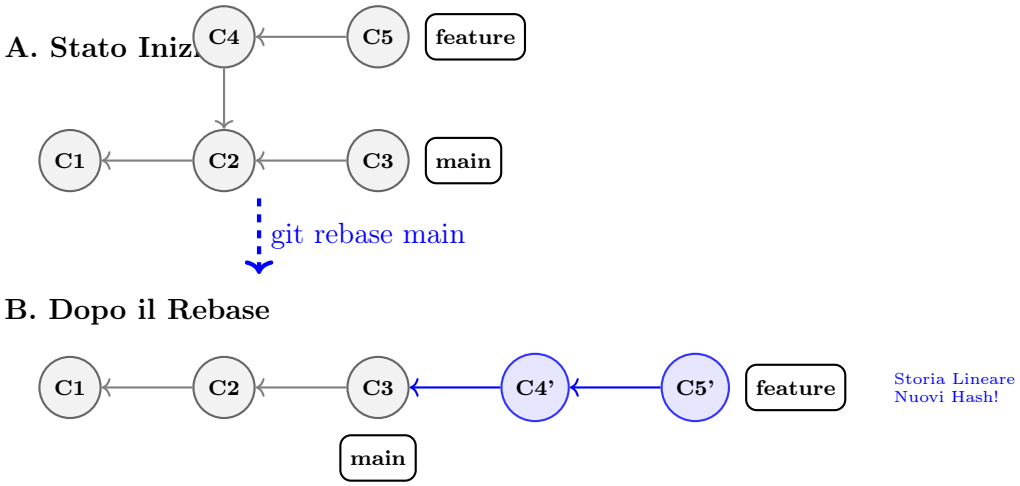
1.12.1 Topologia e Funzionamento

Tecnicamente, il rebase non "sposta" i commit esistenti. Git esegue le seguenti operazioni in sequenza:

1. Identifica l'antenato comune tra i due rami.
2. "Salva" temporaneamente le modifiche introdotte in ogni commit del branch corrente (come patch).
3. Porta il puntatore del branch corrente alla punta del branch di destinazione.
4. **Riapplica** (Replay) le modifiche salvate, una alla volta, creando **nuovi commit** con nuovi hash.

Visualizzazione della Linearizzazione

Di seguito è illustrata la trasformazione del grafo. Si noti come i commit originali $(C4, C5)$ vengano sostituiti da copie $(C4', C5')$.



Il vantaggio operativo è evidente: la storia del progetto diventa una linea retta, priva di biforcazioni e di "merge commit" rumorosi, facilitando la lettura cronologica e l'eventuale debug (bisect).

1.12.2 Interactive Rebase (-i)

Una delle funzionalità più potenti è il *Rebase Interattivo*. Questo comando permette non solo di spostare la base, ma di **modificare la storia locale** prima di condividerla.

Eseguendo:

```
git rebase -i HEAD~3
```

Git apre un editor di testo con la lista degli ultimi 3 commit, permettendo di scegliere un'azione per ciascuno:

- **pick:** Mantieni il commit così com'è.
- **reword:** Modifica solo il messaggio del commit.
- **squash:** Fondi questo commit con il precedente (utile per unire molti commit "WIP" in uno solo pulito).
- **drop:** Elimina completamente il commit dalla storia.

Questo strumento è essenziale per pulire la propria cronologia locale (sanitizzazione) prima di effettuare il push verso il repository remoto.

1.12.3 Attenzione al rebase

Il rebase è un'operazione distruttiva poiché altera gli identificatori (Hash SHA-1) dei commit. Esiste un imperativo categorico per il suo utilizzo:

Non eseguire mai il rebase su branch pubblici condivisi con altri sviluppatori.

Motivazione: Se si esegue il rebase di un branch (es. `main`) che altri hanno già scaricato:

1. Si crea una storia alternativa "B" (i commit ri-applicati).
2. I colleghi possiedono ancora la storia originale "A".
3. Quando cercheranno di sincronizzarsi, Git vedrà due storie divergenti senza un antenato comune recente, costringendo il team a risolvere conflitti complessi e duplicando i commit nel grafo finale.

Best Practice: Usare il rebase solo sui propri branch locali privati (feature branch) prima di integrarli nel ramo principale.

1.12.4 Merge o Rebase?

Come sappiamo, nel modello dei dati di Git, ogni stato del progetto è rappresentato da un nodo in un *Directed Acyclic Graph*. L'integrazione di rami divergenti richiede la riconciliazione di diverse linee temporali, operazione che può essere eseguita secondo due paradigmi logici: la preservazione della topologia parallela (Merge) o la linearizzazione semantica (Rebase).

Analisi git merge

Il processo di `git merge` è governato da un algoritmo di fusione a tre vie. Dato un ramo *feature* derivato dal ramo *main*, Git identifica tre nodi fondamentali:

1. **L'antenato comune (Merge Base):** L'ultimo commit condiviso prima della divergenza.
2. **Il commit di testa del target:** L'attuale `HEAD` del ramo ricevente.
3. **Il commit di testa del source:** L'ultimo commit del ramo da integrare.

Se le modifiche occorrono su file o linee differenti, Git genera automaticamente un **Merge Commit**. Questo nodo è strutturalmente unico poiché possiede due puntatori *parent*.

Analisi git rebase

Il `rebase` è formalmente definito come la ri-applicazione di una serie di modifiche sopra una nuova base. Matematicamente, se definiamo un commit come un set di differenze Δ rispetto al padre, il rebase estrae i Δ dal ramo divergente e li applica sequenzialmente sul tip del ramo target.

$$C' = C_{target} + \Delta_{source} \tag{1.1}$$

Tuttavia, poiché il contesto del codice circostante può essere mutato nel ramo target, ogni commit riapplicato genera un nuovo identificativo SHA-1. Il risultato è la distruzione della cronologia originale a favore di una struttura puramente lineare.

Analisi Comparativa e Criteri di Selezione

La scelta tra queste procedure non è meramente estetica, ma incide sulla manutenibilità del software a lungo termine.

- **Verificabilità Storica (Merge):** In contesti di auditing o sistemi critici, il merge è preferibile poiché documenta l'esatto istante e contesto in cui due flussi di lavoro si sono incontrati.

Parametro	**Git Merge**	**Git Rebase**
Modifica del DAG	Estensiva (aggiunge nodi)	Trasformativa (riscrive i nodi)
Risoluzione Conflitti	Centralizzata nel merge commit	Atomica (commit per commit)
Tracciabilità	Elevata (preserva i rami laterali)	Ridotta (appiattisce la storia)
Rischio Operativo	Basso	Alto (riscrittura della storia pubblica)

Tabella 1.1: Confronto sistematico tra integrazione additiva e lineare.

Come già spiegato in precedenza, si stabilisce come principio fondamentale che **non si deve mai eseguire il rebase di commit che sono stati pubblicati in un repository condiviso**. Poiché il rebase sostituisce i commit originali con versioni nuove e apparentemente identiche, gli altri sviluppatori che basano il proprio lavoro sui commit originali si troverebbero in una condizione di divergenza forzata della cronologia, portando a complessi conflitti di riconciliazione (*upstream rebase problem*).

Esempio di Divergenza

Si consideri uno scenario in cui uno sviluppatore S_1 effettua il rebase di un commit C già presente sul server, trasformandolo in C'. Un secondo sviluppatore S_2, che aveva già basato il proprio lavoro D sul nodo C, si troverà in una situazione di inconsistenza:

- **Storia Locale S_2:** $A \to B \to C \to D$
- **Storia Remota (S_1 dopo push force):** $A \to B \to C'$

Al successivo tentativo di sincronizzazione, Git interpreterà C e C' come due linee di sviluppo divergenti. L'unione automatica (*merge*) comporterà la duplicazione semantica delle modifiche (presenza sia di C che di C' nel grafo), costringendo S_2 a una risoluzione manuale complessa e soggetta a errori, vanificando i benefici della riscrittura operata da S_1.

Capitolo 2

GitHub e i Remote Repository

Fino a questo punto, l'analisi si è concentrata esclusivamente sull'uso di Git in locale (*Local Repository*). Tuttavia, la potenza di un sistema di controllo versione distribuito (DVCS) si manifesta pienamente quando il codice viene condiviso e sincronizzato attraverso una rete.

In questo capitolo si introduce il concetto di Remote Repository e si analizza il ruolo di piattaforme di hosting come GitHub.

2.1 Architettura Distribuita: Git vs GitHub

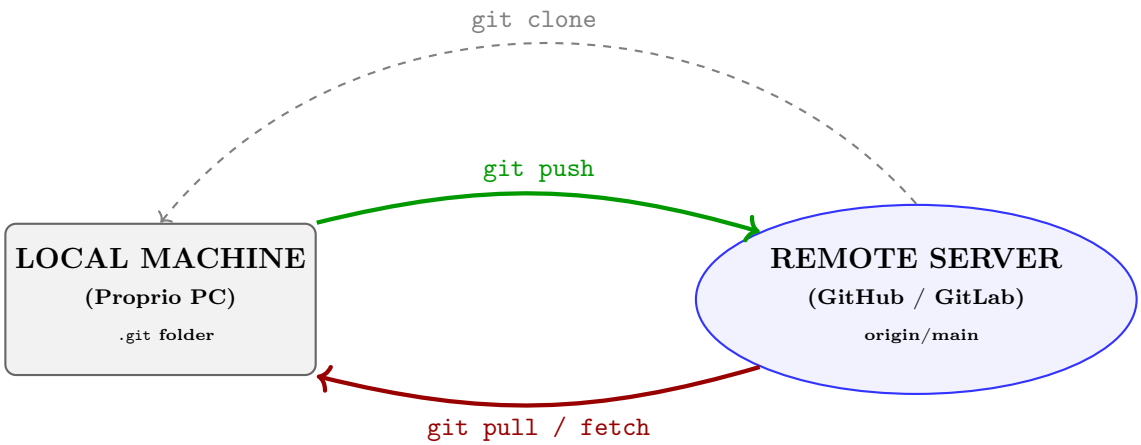
È un errore comune confondere lo strumento con il servizio. È necessario stabilire una distinzione tecnica rigorosa:

- **Git:** È il software (il motore) installato sulla macchina locale. Gestisce il versionamento, i branch e la storia dei commit. Funziona offline e non richiede alcuna connessione internet.
- **GitHub (o GitLab/Bitbucket):** È un servizio di hosting web che offre server remoti per ospitare repository Git. Aggiunge un livello di funzionalità "sociali" e di gestione del progetto (Pull Request, Issue Tracking, CI/CD) sopra il protocollo Git standard.

Mentre Git gestisce la storia del codice, GitHub gestisce la collaborazione tra sviluppatori.

2.1.1 Topologia del Workflow Remoto

L'interazione tra la macchina locale e il server remoto avviene tramite comandi specifici di sincronizzazione. Non esiste un collegamento live costante: le modifiche devono essere esplicitamente inviate (*Push*) o scaricate (*Pull*).



2.2 Configurazione dei Remotes

Per collegare un repository locale a uno su GitHub, è necessario definire un *Remote*. Tecnicamente, un remote è semplicemente un alias (un soprannome) salvato nella configurazione locale che punta a un URL specifico.

Il nome standard convenzionale per il repository remoto principale è `origin`.

2.2.1 Protocolli di Connessione

Esistono due metodi per identificare l'URL remoto. La scelta impatta sulla modalità di autenticazione:

1. **HTTPS:** `https://github.com/user/repo.git`
   Metodo universale. Richiede l'inserimento di credenziali (Username + Personal Access Token) ad ogni operazione di scrittura, a meno che non si usi un *Credential Helper*.

2. **SSH:** `git@github.com:user/repo.git`
   Utilizza una coppia di chiavi crittografiche (pubblica/privata). È lo standard professionale poiché permette l'autenticazione sicura senza dover inserire password ripetutamente.

Nota su SSH

Per utilizzare il protocollo SSH, è necessario generare una chiave locale (`ssh-keygen`) e caricare la chiave pubblica (`.pub`) nelle impostazioni del proprio profilo GitHub alla voce "SSH and GPG keys".

2.2.2 Configurazione dell'Autenticazione Crittografica (SSH)

L'interazione sicura con i repository remoti richiede l'instaurazione di un canale cifrato. Sebbene, come già spiegato, il protocollo HTTPS sia supportato, lo standard industriale per lo sviluppo software prevede l'utilizzo del protocollo **SSH (Secure Shell)** mediante autenticazione a chiave pubblica.

Questa metodologia elimina la necessità di inviare le credenziali (username e password) attraverso la rete a ogni operazione, sostituendole con una verifica crittografica basata su una coppia di chiavi:

- **Chiave Privata:** Risiede esclusivamente sulla macchina locale e non deve mai essere condivisa.
- **Chiave Pubblica:** Viene caricata sul server remoto (GitHub) e funge da identificativo univoco.

Per la documentazione ufficiale completa e aggiornata, si rimanda alle guide di riferimento GitHub:

- Generazione di una nuova chiave SSH
- Aggiunta della chiave all'account GitHub

1. Generazione della Coppia di Chiavi

Si raccomanda l'utilizzo dell'algoritmo **Ed25519**, il quale offre prestazioni e sicurezza superiori rispetto al legacy RSA.

Eseguire il seguente comando nel terminale, avendo cura di sostituire l'indirizzo email con quello associato all'account GitHub:

Codice 2.1: Generazione chiave Ed25519

```
ssh-keygen -t ed25519 -C "email.studente@dominio.com"
```

Il sistema richiederà il percorso di salvataggio (si consiglia di mantenere il default premendo *Invio*) e una *passphrase* opzionale per cifrare la chiave privata su disco.

2. Configurazione dell'SSH Agent Locale

Affinché il sistema operativo gestisca automaticamente l'identità crittografica durante le sessioni di terminale, è necessario avviare l'`ssh-agent` in background e aggiungervi la chiave privata appena generata.

Codice 2.2: Attivazione dell'Agent

```
# Avvio del processo agent in background
eval "$(ssh-agent -s)"

# Aggiunta dell'identità (chiave privata)
ssh-add ~/.ssh/id_ed25519
```

3. Associazione della Chiave Pubblica al Remote

L'ultimo passaggio richiede il trasferimento della chiave **pubblica** (file con estensione `.pub`) sui server di GitHub.

È necessario visualizzare e copiare il contenuto esatto della chiave pubblica:

Codice 2.3: Visualizzazione della chiave pubblica

```
# Comando universale (Linux/macOS/Git Bash)
cat ~/.ssh/id_ed25519.pub
```

Il contenuto visualizzato (una stringa che inizia con `ssh-ed25519`) deve essere copiato interamente negli appunti. Successivamente, tramite l'interfaccia web di GitHub:

1. Accedere alle **Settings** del profilo utente.
2. Selezionare la sezione **SSH and GPG keys**.
3. Cliccare su **New SSH key**, assegnare un titolo identificativo (es. *Workstation Lab 1*) e incollare la chiave nel campo apposito.

> È fondamentale non copiare mai il contenuto del file privo di estensione (la chiave privata) o condividerlo con terzi. La compromissione della chiave privata garantisce accesso completo a tutti i repository dell'utente.

Verifica della Connessione

Per confermare il corretto stabilimento del tunnel cifrato e l'avvenuto riconoscimento da parte del server, si esegua il test di connessione:

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

Un output contenente il messaggio *"You've successfully authenticated"* conferma che l'ambiente locale è correttamente configurato per le operazioni di *Push* e *Pull*.

2.3 `git remote`

Un repository Git locale è, per definizione, isolato. Per abilitare la collaborazione, è necessario configurare uno o più collegamenti verso server esterni. Il comando `git remote` gestisce il registro di questi collegamenti, mappando URL complessi a nomi mnemonici (alias) più semplici da utilizzare.

È importante sottolineare che l'aggiunta di un remote è un'operazione puramente locale: modifica il file di configurazione `.git/config` senza effettuare, in questa fase, alcuna connessione di rete verso il server.

2.3.1 Ispezione dei Remote Esistenti

Per visualizzare l'elenco dei server remoti configurati per il repository corrente, si utilizza il comando senza argomenti o con il flag di verbosità.

- **Lista semplice:** Restituisce solo i nomi degli alias.
- **Lista dettagliata (`-v`):** Mostra l'URL associato a ciascun alias e il tipo di operazione consentita (lettura/scrittura).

Codice 2.4: Ispezione dei remote con flag verbose

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

Nell'output precedente, `origin` è il nome standard convenzionale attribuito al server principale. Le diciture `(fetch)` e `(push)` indicano rispettivamente l'indirizzo da cui scaricare i dati e quello verso cui inviarli (solitamente coincidono).

2.3.2 Aggiunta di un nuovo Remote

La sintassi per registrare un nuovo repository remoto è la seguente:

```
git remote add <nome-alias> <url>
```

Requisiti per l'operazione: Affinché il comando abbia successo, devono essere soddisfatte le seguenti condizioni logiche:

1. Il comando deve essere eseguito all'interno di una directory già inizializzata come repository Git.
2. Il `<nome-alias>` deve essere univoco nel contesto locale (non è possibile avere due remote chiamati `origin`).
3. L'`<url>` deve rispettare la sintassi dei protocolli supportati (HTTPS o SSH), sebbene la validità dell'indirizzo venga verificata solo al primo tentativo di connessione.

Codice 2.5: Esempio di aggiunta remote

```
# Aggiunta del repository principale
$ git remote add origin git@github.com:universita/progetto-lab.git

# Aggiunta di un repository secondario (es. di un collega)
$ git remote add luigi https://github.com/luigi/progetto-lab.git
```

Rinominare e Rimuovere

Nel ciclo di vita di un progetto, può essere necessario modificare la topologia dei collegamenti remoti.

Rinominare un alias (`rename`) Se si desidera cambiare l'etichetta locale di un remote (ad esempio, da `origin` a `backup`):

```
$ git remote rename origin backup
```

Questa operazione aggiorna automaticamente anche tutti i rami a tracciamento remoto (remote-tracking branches) associati.

Rimuovere un alias (`remove`) Qualora un server non sia più necessario o l'URL sia cambiato drasticamente:

```
$ git remote remove origin
# Sintassi alternativa deprecata ma diffusa:
# git remote rm origin
```

Nota Tecnica sulla Rimozione

Il comando `remove` agisce esclusivamente sulla configurazione locale. **Non cancella** il repository sul server remoto, né elimina i file locali. Rimuove semplicemente il "puntatore" e i rami di riferimento (es. `origin/main`) dalla memoria del repository locale.

2.3.3 Gestione degli Errori Comuni

Durante la configurazione dei remote, l'errore più frequente riguarda la duplicazione degli alias.

Codice 2.6: Errore di duplicazione

```
$ git remote add origin git@github.com:nuovo/repo.git
fatal: remote origin already exists.
```

Analisi e Risoluzione: Questo errore indica che l'alias `origin` è già occupato. Git impedisce di sovrascriverlo accidentalmente. Per risolvere, l'utente deve scegliere una delle seguenti strade:

1. Usare un nome diverso per il nuovo remote (es. `upstream`).
2. Rimuovere il vecchio remote esistente tramite `git remote remove origin`.
3. Cambiare l'URL del remote esistente (senza rimuoverlo) tramite il comando `set-url`:

```
$ git remote set-url origin git@github.com:nuovo/repo.git
```

2.3.4 Stato del Repository Locale e Vincoli di Sincronizzazione

È fondamentale chiarire un concetto architetturale: il comando `git remote add` è un'operazione puramente logica di configurazione. Esso non controlla, non tocca e non modifica i file presenti nella directory di lavoro (*Working Directory*).

Di conseguenza, **non è necessario che la cartella locale sia vuota**. Tuttavia, lo stato dei file presenti determina come avverrà la prima sincronizzazione. Si configurano tre scenari principali:

Scenario A: Progetto Locale verso Remote Vuoto (Standard)

Questo è il caso più comune per nuovi progetti.

- **Locale:** Contiene codice sorgente e una storia di commit.
- **Remote:** È appena stato creato su GitHub ed è completamente vuoto (senza file `README`, `.gitignore` o licenza).

Risultato: L'operazione è sicura. Al primo `push`, il contenuto locale popolerà il server remoto, rendendoli identici.

Scenario B: Cartella Vuota verso Remote Esistente

Si verifica quando si vuole recuperare un progetto già avviato altrove, ma invece di usare `git clone` (che crea una nuova cartella), si inizializza manualmente una cartella esistente.

- **Locale:** Contiene solo la cartella nascosta `.git`.
- **Remote:** Contiene già file e commit.

Risultato: L'operazione è sicura. Al primo `pull`, i file remoti verranno scaricati nella cartella locale.

Scenario C: Storie non Correlate

Questo è il caso critico a cui prestare attenzione (garbage nella cartella locale).

- **Locale:** Contiene dei commit (es. un progetto iniziato localmente).
- **Remote:** Contiene *altri* commit (es. un file `README.md` creato automaticamente da GitHub all'apertura del repository).

In questo caso, sebbene `git remote add` non dia errori, il primo tentativo di unire i due mondi (`git pull` o `git merge`) fallirà con l'errore:

```
fatal: refusing to merge unrelated histories
```

Gestione delle Storie Non Correlate

Git, di default, si rifiuta di unire due progetti che non condividono un antenato comune (il primo commit). Se ci si trova in questa situazione intenzionalmente, è necessario forzare l'unione con il flag: `-allow-unrelated-histories`.
Tuttavia, la *best practice* è evitare questa situazione:

1. Se il progetto esiste già in locale, creare il repository remoto **completamente vuoto**.
2. Se il progetto esiste già in remoto, usare `git clone` in una nuova cartella locale.

2.4 `git push`

Il comando `git push` è l'operazione che sincronizza il repository remoto con quello locale, trasferendo i commit mancanti e aggiornando i puntatori dei branch remoti.

A differenza di un semplice trasferimento file, il push è una transazione che ha successo solo se la storia locale è coerente con quella remota.

2.4.1 Sintassi e Flusso di Lavoro

La sintassi formale del comando richiede di specificare la destinazione e il ramo sorgente:

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

Dove:

- `<remote>`: L'alias del server (solitamente `origin`).
- `<branch>`: Il ramo locale che si desidera inviare (es. `main` o `feature-x`).

Il Primo Push e l'Upstream Tracking

Quando si invia un branch per la prima volta, è necessario stabilire una relazione di tracciamento (*tracking*) tra il ramo locale e quello remoto corrispondente. Questo si ottiene con il flag `-u` (o `-set-upstream`).

Codice 2.7: Primo invio con configurazione upstream

```
$ git push -u origin main
```

Effetti del comando:

1. Crea il ramo `main` sul server remoto `origin` (se non esiste).
2. Carica i commit.
3. Configura il file `.git/config` locale in modo che i futuri comandi `git push` e `git pull` sappiano automaticamente a quale ramo remoto fare riferimento senza doverlo specificare ogni volta.

Dalla seconda volta in poi, sarà sufficiente digitare:

```
git push
```

2.4.2 Fast-Forward

Git protegge l'integrità della storia remota applicando una regola ferrea: il push viene accettato **solo se** il commit locale è un discendente diretto dell'ultimo commit presente sul server. Questa condizione è detta *Fast-Forward*.

Se un altro collaboratore ha effettuato un push sullo stesso ramo durante lo sviluppo locale, le due storie sono divergenti. In questo scenario, Git rifiuterà il push per prevenire la sovrascrittura del lavoro altrui.

Codice 2.8: Errore di Non-Fast-Forward

```
$ git push
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to '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.
```

Procedura di Risoluzione: Non è possibile forzare l'invio in questo stato (a meno di azioni distruttive). La procedura corretta prevede:

1. Scaricare le modifiche remote (`git pull` o `git fetch`).
2. Integrare le storie (tramite *Merge* o *Rebase*).
3. Ripetere il `git push` con la storia unificata.

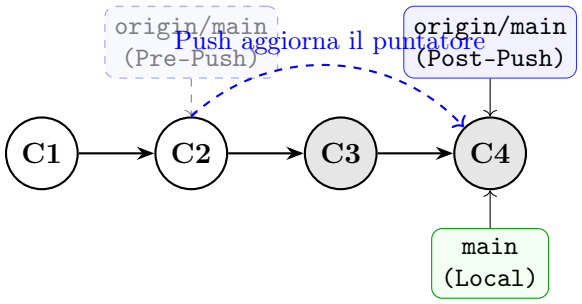


Figura 2.1: **Fast-Forward Push:** I commit C3 e C4 esistono solo in locale. Poiché C4 discende direttamente da C2 (il vecchio stato remoto), Git accetta il push e sposta semplicemente il puntatore `origin/main` in avanti.

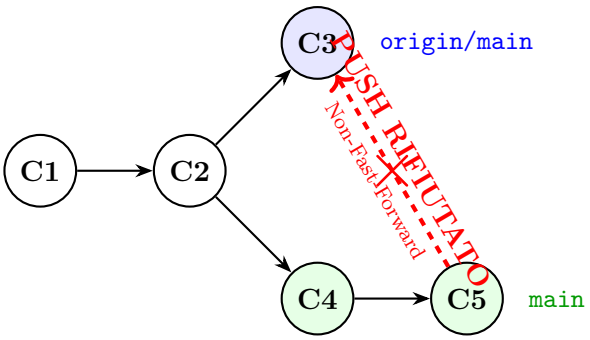


Figura 2.2: **Push Rifiutato:** Il repository remoto è avanzato al commit C3. Il nostro lavoro locale (C4, C5) parte da C2, ignorando l'esistenza di C3. Git impedisce il push per non sovrascrivere la storia remota.

2.4.3 Force Push

Esistono scenari eccezionali (come quello mostrato nella figura qua sopra, ad esempio dopo un `rebase` locale o la rimozione di dati sensibili) in cui la storia locale e remota sono diverse e si desidera sovrascrivere il server.

Il flag `-force` disabilita i controlli di sicurezza e impone lo stato locale sul remoto, cancellando eventuali commit presenti solo sul server.

L'uso indiscriminato di `git push --force` è la causa principale di perdita di dati nei team di sviluppo. Se si sovrascrive un ramo pubblico (es. `main`), tutti i collaboratori che hanno basato il loro lavoro sui commit cancellati si troveranno con repository corrotti.

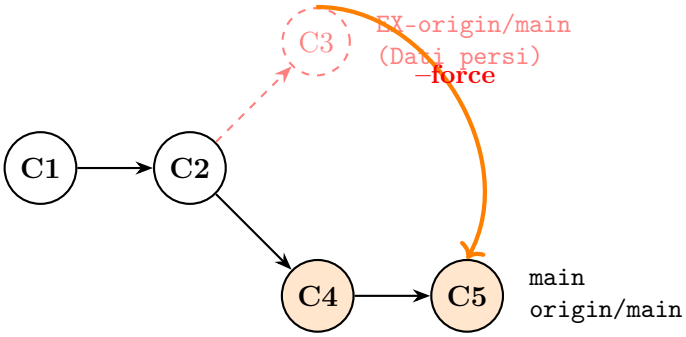


Figura 2.3: **Effetti del Force Push:** L'esecuzione di `git push --force` impone lo stato locale (C5) al server. Il commit C3, che era precedentemente sul server, viene scollegato dalla storia e i dati contenuti in esso vanno persi per tutti i collaboratori.

alternativa sicura: `--force-with-lease`

Git offre inoltre un'opzione di sicurezza superiore:

```
git push --force-with-lease
```

Questo comando esegue la sovrascrittura **solo se** nessuno ha aggiunto nuovi commit sul ramo remoto nel frattempo. Se il server contiene modifiche che il client non ha ancora rilevato (non ha fatto *fetch*), il comando fallisce, proteggendo il lavoro dei colleghi.

2.4.4 Push di Tag

È importante notare che il comando standard `git push` trasferisce solo i commit e i branch, ma **non i Tag** annotati (l'argomento verrà trattato in seguito). Per inviare le versioni (release) al server, è necessario un comando esplicito:

```
# Invia un singolo tag
git push origin v1.0.0

# Invia tutti i tag presenti in locale
git push origin --tags
```

2.4.5 Configurazione del Tracciamento (Upstream)

Quando si crea un nuovo branch locale (es. `feature-login`) e si tenta di eseguire un semplice `git push`, Git blocca l'operazione restituendo un errore:

```
fatal: The current branch feature-login has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin feature-login
```

L'Ambiguità della Destinazione

Git è progettato per essere esplicito. Anche se esiste un solo remote (`origin`), Git non assume automaticamente che il ramo locale `main` debba corrispondere al ramo remoto `origin/main`. Si potrebbe voler inviare il ramo `dev` locale verso un ramo chiamato `beta` sul server.

Finché non viene stabilita una relazione formale di **tracciamento (tracking)**, Git non sa dove indirizzare i dati.

Soluzione: Impostare l'Upstream

Per risolvere l'ambiguità, è necessario dire a Git: "Da ora in poi, collega il ramo locale X al ramo remoto Y".

Esistono due modi per farlo durante il primo push:

1. sintassi estesa : È il comando suggerito nell'errore stesso. È molto esplicito ma lungo da digitare:

```
git push -set-upstream origin <nome-branch>
```

2. scorciatoia (`-u`) Poiché questa operazione è necessaria ogni volta che si pubblica un nuovo ramo, Git offre un flag breve che è esattamente equivalente al precedente:

```
git push -u origin <nome-branch>
```

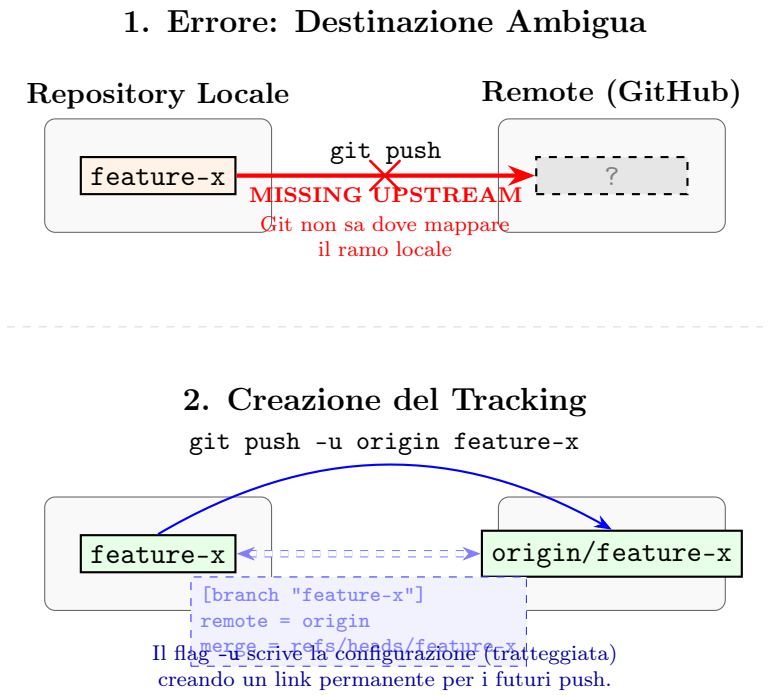


Figura 2.4: **Il concetto di Upstream Tracking.** In alto: senza configurazione, Git non può indovinare la corrispondenza tra rami locali e remoti. In basso: il comando con `-u` non solo trasferisce i dati, ma crea un legame persistente (configurazione) che orienta automaticamente i trasferimenti successivi.

Cosa succede realmente

L'utilizzo del flag `-u` non è solamente una comodità momentanea. Esso modifica permanentemente il file di configurazione locale `.git/config`, aggiungendo le seguenti righe:

```
[branch "main"]
    remote = origin
    merge = refs/heads/main
```

2.5 `git clone`

Il comando `git clone` è lo strumento standard per duplicare in locale un repository remoto esistente. Questa operazione non si limita a scaricare l'ultima versione dei file sorgente (come avverrebbe tramite il download di un archivio ZIP), ma trasferisce l'intero database crittografico di Git, includendo la cronologia completa dei commit, tutti i branch e i tag.

Al termine dell'esecuzione, la directory risultante è a tutti gli effetti un repository Git completo, capace di operare offline per l'ispezione della storia e la creazione di nuovi commit.

2.5.1 Sintassi Fondamentale

La forma più comune del comando richiede unicamente l'URL (Uniform Resource Locator) del repository bersaglio:

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

L'URL può utilizzare i protocolli HTTPS (es. `https://github.com/user/repo.git`) o SSH (es. `git@github.com:user/repo.git`). Come discusso nelle sezioni precedenti, l'utilizzo di SSH è raccomandato per evitare l'inserimento ripetuto delle credenziali.

Comportamento Predefinito

Se eseguito senza ulteriori argomenti, il comando esegue automaticamente una serie di operazioni sequenziali:

1. Crea una nuova directory nel percorso corrente, assegnandole lo stesso nome del repository remoto (privato dell'estensione `.git`).
2. Inizializza la nuova directory come repository Git (creazione della cartella nascosta `.git`).
3. Configura automaticamente un *remote* locale, nominandolo per convenzione `origin`, che punta all'URL fornito.
4. Scarica l'intero database degli oggetti Git dal server.
5. Effettua il *checkout* (l'estrazione dei file nella *Working Directory*) del branch predefinito del repository remoto (generalmente `main` o `master`).

2.5.2 Struttura del comando

Per comprendere appieno l'architettura di Git, è utile scomporre l'operazione di clonazione. Dal punto di vista logico, eseguire `git clone <url>` è l'equivalente della seguente sequenza di comandi a basso livello:

Codice 2.9: Meccanica interna di un git clone

```
mkdir nome-repo
cd nome-repo
git init
git remote add origin <url>
git fetch origin
git checkout -b main --track origin/main
```

Questa scomposizione evidenzia come il *clone* sia essenzialmente un'operazione di bootstrap che prepara l'ambiente di lavoro, configurando il tracciamento (*tracking*) iniziale senza richiedere interventi manuali.

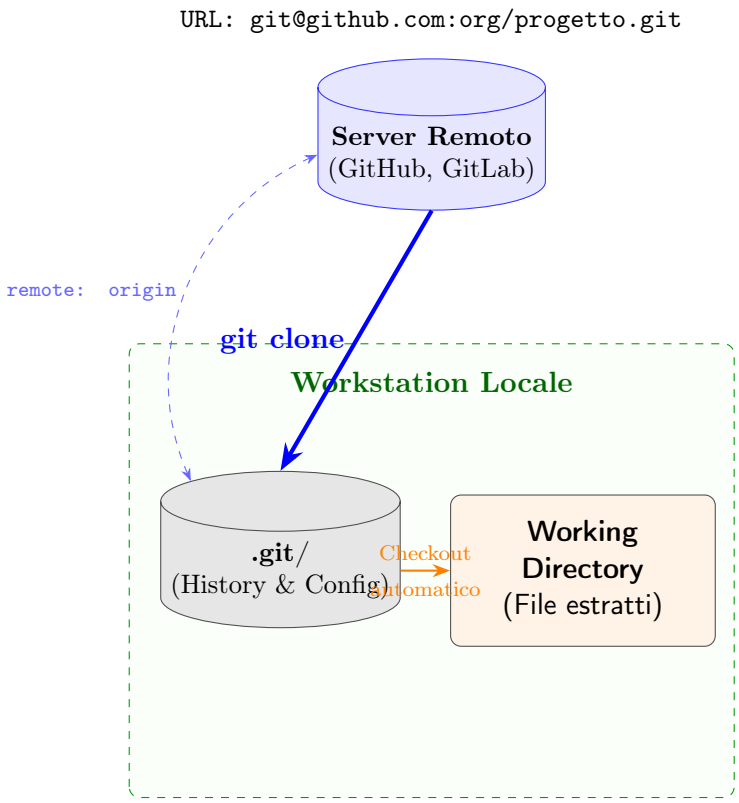


Figura 2.5: **Architettura del comando Clone.** Il processo trasferisce l'intero database degli oggetti dal server remoto alla cartella locale `.git`, stabilisce il collegamento `origin` e infine estrae i file del branch principale nella *Working Directory* per permettere la modifica.

2.5.3 Ottimizzazione delle Risorse

In contesti industriali o in presenza di repository con una storia decennale (es. il kernel Linux), il download dell'intero database può richiedere quantità proibitive di banda e spazio su disco. Git fornisce parametri specifici per mitigare questo problema.

Clonazione Superficiale (Shallow Clone)

L'argomento `--depth` permette di troncare la cronologia, scaricando solo un numero specifico di commit recenti.

```
# Clona scaricando unicamente l'ultimo commit del branch predefinito
git clone --depth 1 <url>
```

Questa modalità è particolarmente utile nei sistemi di *Continuous Integration* (CI/CD), dove il server di automazione necessita unicamente del codice sorgente attuale per compilare o testare l'applicativo, senza alcun interesse per la storia passata.

Clonazione di un Branch Specifico

Di default, `git clone` scarica i riferimenti a tutti i branch remoti. Se si desidera isolare un singolo ramo (ad esempio, una specifica release in manutenzione), si utilizzano le opzioni `--branch` e `--single-branch`:

```
git clone --branch v2.0-bugfix --single-branch <url>
```

L'utilizzo combinato di queste due opzioni istruisce Git a scaricare *esclusivamente* la storia lineare necessaria per ricostruire il ramo specificato, ignorando il resto del repository e ottimizzando drasticamente i tempi di trasferimento.

> **Nota**
>
> Un errore comune consiste nell'eseguire il clone all'interno di un repository Git già esistente. Sebbene il sistema operativo consenta l'operazione, la creazione di un repository dentro un altro repository (senza l'uso esplicito dei *submodules*) genera conflitti di tracciamento difficili da risolvere. Si raccomanda di eseguire `git clone` sempre all'interno di directory di lavoro non versionate.

2.5.4 Potenziali limitazioni derivanti dai Repository Parziali

L'utilizzo di `--single-branch` crea un ambiente di lavoro isolato, definito tecnicamente "a visione limitata". Poiché il comando `git merge` richiede l'accesso alla storia di entrambi i rami coinvolti per calcolare l'antenato comune (Three-Way Merge), è impossibile fondere un ramo che non è stato scaricato.

Se si tenta di eseguire `git merge origin/main` in un repository clonato singolarmente, Git restituirà l'errore:

```
merge: origin/main - not something we can merge
```

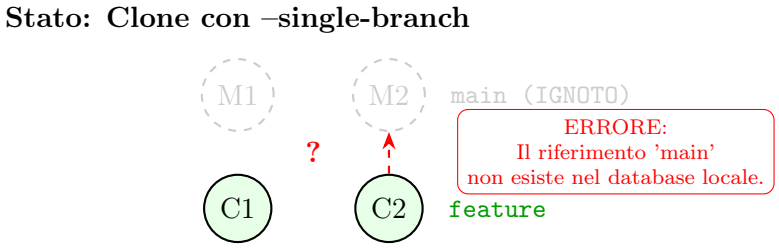


Figura 2.6: **Cecità del Clone Parziale.** Quando si clona un singolo ramo, i commit degli altri rami (rappresentati in grigio tratteggiato) non vengono scaricati sul disco. Git non può unire C2 con M2 perché non possiede fisicamente l'oggetto M2 né i suoi antenati.

Esistono due strategie per gestire questa situazione, a seconda del flusso di lavoro adottato.

A: Merge lato Server (Pull Request)

Nel moderno sviluppo software, spesso non è necessario effettuare il merge localmente. Se lo scopo della clonazione parziale è correggere un bug rapidamente:

1. Si lavora sul ramo isolato.
2. Si effettua il push delle modifiche sullo stesso ramo remoto.
3. Si utilizza l'interfaccia web del provider (GitHub/GitLab) per aprire una **Pull Request**.

In questo scenario, sarà il server (che possiede tutti i rami) a farsi carico dell'operazione di merge. L'operazione di Pull Request verrà approfondita in seguito.

B: Espansione della Configurazione

Qualora sia indispensabile effettuare il merge localmente (ad esempio per testare l'integrazione), è necessario istruire il repository locale a vedere anche gli altri rami.

Non basta un semplice `git fetch` (di cui si parlerà nella prossima sezione), poiché la configurazione del clone parziale limita esplicitamente il download al singolo ramo. Bisogna modificare la configurazione dei *remote*:

Codice 2.10: Ripristino della visibilità completa

```
# 1. Modifica la configurazione per tracciare tutti i rami
git config remote.origin.fetch "+refs/heads/*:refs/remotes/origin/*"

# 2. Scarica le informazioni mancanti
git fetch origin

# 3. Ora il ramo main è visibile e mergiabile
git merge origin/main
```

2.6 `git fetch`

Nel modello distribuito di Git, lo stato del repository remoto non viene aggiornato in tempo reale sul computer locale. Per visionare le modifiche apportate dai colleghi, è necessario interrogare esplicitamente il server.

Il comando `git fetch` assolve a questa funzione: contatta il remote specificato, scarica tutti i dati (commit, file, riferimenti) che non sono presenti in locale, e aggiorna i cosiddetti **Remote-Tracking Branches** (es. `origin/main`).

2.6.1 Principio di Sicurezza (Non-Distruttivo)

La caratteristica distintiva di `fetch` è la sua totale sicurezza.

- **Aggiorna il Database:** I nuovi commit vengono salvati nella cartella `.git`.
- **Non tocca la Working Directory:** I file su cui si sta lavorando rimangono invariati.
- **Non altera la posizione dei branch locali:** Il puntatore `main` non si sposta.

In termini pratici, `git fetch` è un'operazione di *presa visione*. Permette di analizzare cosa è successo sul server (eventualmente usando `git log origin/main`) senza essere costretti a unire subito il lavoro altrui col proprio.

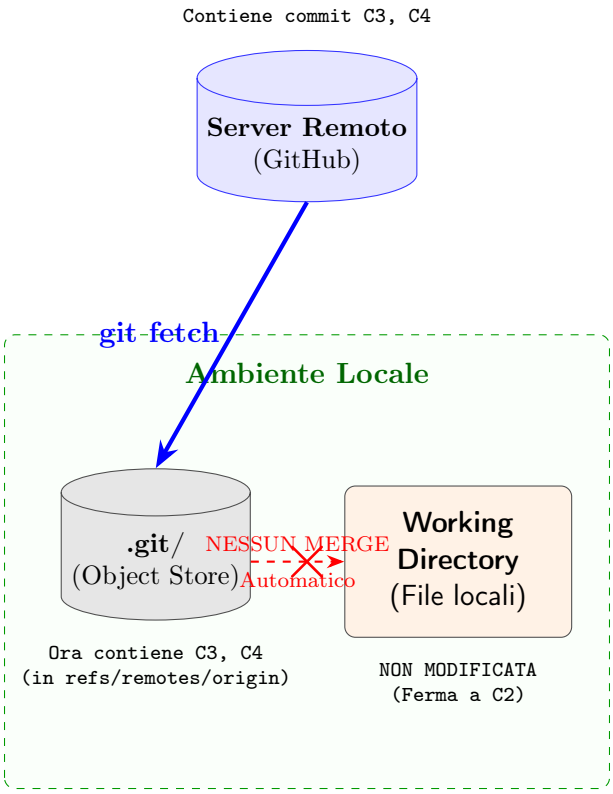


Figura 2.7: **Funzionamento del Fetch.** Il comando scarica i dati dal server nel database locale nascondendoli nei *Remote-Tracking Branches*. Notare la barriera rossa: i file visibili all'utente nella cartella di lavoro non vengono toccati, rendendo l'operazione sicura in qualsiasi momento.

2.6.2 Manutenzione dei Riferimenti (`--prune`)

Un aspetto avanzato del fetch riguarda la pulizia. Se un ramo viene cancellato sul server (es. dopo il merge di una Pull Request), Git locale non lo rimuove automaticamente per evitare perdite accidentali di riferimenti.

Per sincronizzare anche le cancellazioni e rimuovere i riferimenti ai rami remoti ormai inesistenti, si utilizza il flag `-prune` (o `-p`):

```
git fetch --prune
# Output tipico:
# - [deleted]         (none)     -> origin/feature-login
```

2.7 `git pull`

Il comando `git pull` costituisce il meccanismo standard per l'aggiornamento del repository locale con le modifiche presenti sul server remoto. A differenza del semplice download, questa operazione implica un'alterazione immediata dello stato della *Working Directory*.

Dal punto di vista architetturale, `git pull` non è un comando atomico, bensì uno script di convenienza (macro) che esegue sequenzialmente due operazioni fondamentali:

$$\texttt{git pull} = \underbrace{\texttt{git fetch}}_{\text{Download}} + \underbrace{\texttt{git merge}}_{\text{Integrazione}} \tag{2.1}$$

2.7.1 Dinamica dell'Operazione

Quando viene invocato, il comando procede attraverso le seguenti fasi:

1. **Fase di Fetch:** Contatta il server remoto e scarica i nuovi commit, aggiornando i riferimenti dei rami remoti (es. `origin/main`). Questa fase è sicura e non genera conflitti.

2. **Fase di Merge:** Tenta immediatamente di fondere il ramo remoto appena scaricato nel ramo locale corrente. È in questa fase che possono emergere conflitti di codice se le modifiche locali sono incompatibili con quelle remote.

2.7.2 Comparazione dei Flussi di Dati

Per comprendere appieno l'impatto di questo comando, è utile confrontare il flusso dei dati rispetto alle operazioni di clonazione e prelievo (fetch).

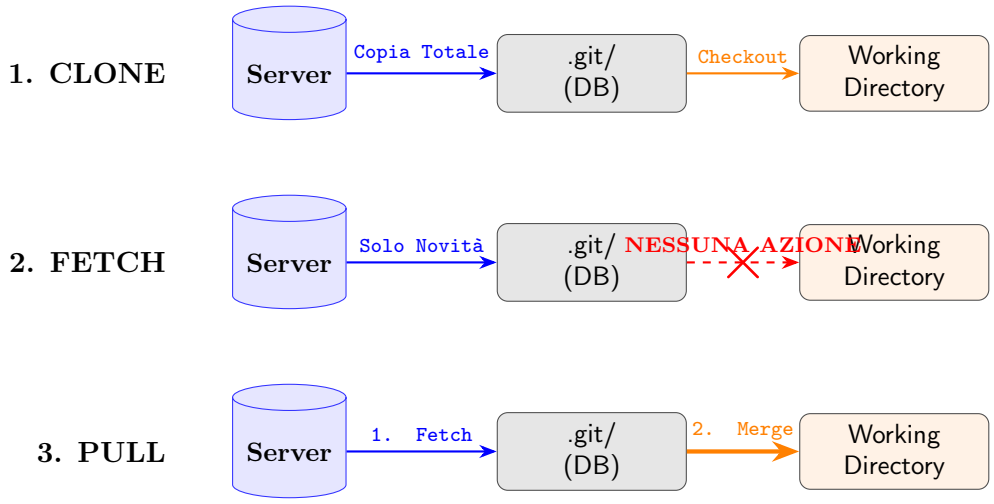


Figura 2.8: **Confronto dei flussi di sincronizzazione.** *Clone* inizializza l'intero ambiente. *Fetch* aggiorna solo il database nascosto, isolando l'utente dalle modifiche. *Pull* aggiorna il database e forza immediatamente l'aggiornamento dei file di lavoro, esponendo l'utente a potenziali conflitti.

2.7.3 Conseguenze del `git pull`

Poiché `git pull` implica un'integrazione, di default esso crea un **Merge Commit** ogni volta che la storia locale diverge da quella remota. In team numerosi, questo comportamento può portare a una cronologia "sporca", ricca di ramificazioni non necessarie.

È possibile (e spesso raccomandato) modificare l'integrazione utilizzando il flag `-rebase`:

```
git pull --rebase
```

In questa modalità, l'equazione 2.1 cambia:

$$\texttt{git pull --rebase} = \texttt{git fetch} + \texttt{git rebase}$$

Anziché creare un nodo di unione, Git sposta temporaneamente i commit locali, aggiorna il ramo con le novità dal server, e riapplica i commit locali in cima alla nuova storia, mantenendo la cronologia lineare.

Pull Fast-Forward Only

Per evitare fusioni accidentali impreviste, è possibile istruire Git ad accettare l'aggiornamento solo se questo può avvenire in modo lineare (senza conflitti e senza merge commit):

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

Se l'operazione richiede un merge reale, il comando fallisce, lasciando all'utente la responsabilità di decidere come procedere manualmente.

2.7.4 Misconcetti sul `git pull`

Una delle fonti di maggiore confusione nei neofiti riguarda la natura stessa dell'operazione di merge invocata da `git pull`. La domanda sorge spontanea: *"Se sto lavorando sullo stesso ramo (main), cosa sto unendo esattamente?"*.

È fondamentale comprendere che in Git non esiste il concetto di "file condiviso simultaneamente". Esistono invece due rami distinti che, pur condividendo lo stesso nome logico, possiedono storie indipendenti:

1. **Il Ramo Locale (`refs/heads/main`):** Rappresenta la linea temporale delle modifiche effettuate sulla workstation locale.
2. **Il Ramo Remoto (`refs/remotes/origin/main`):** Rappresenta lo stato del server, che è stato scaricato nel database locale durante la fase di *fetch*.

Quando `git pull` esegue il merge, non sta "aggiornando" i file in senso letterale, ma sta riconciliando la divergenza tra questi due rami.

Esempio

Si consideri lo scenario in cui, partendo da un commit comune, sia l'utente locale che un collega remoto hanno prodotto nuovi commit. Agli occhi di Git, la topologia del grafo è identica a quella di due feature branch separati.

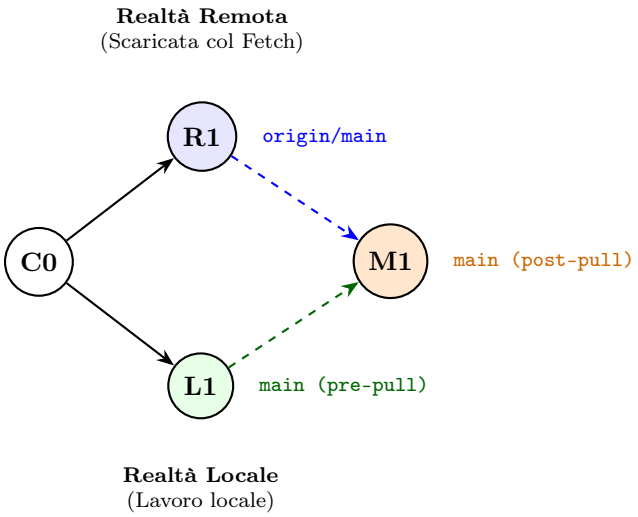


Figura 2.9: **Natura duale del Pull.** Nonostante il nome logico sia lo stesso ("main"), Git tratta la versione remota e quella locale come due linee di sviluppo divergenti che necessitano di un nodo di unione (M1) per essere riconciliate.

Come illustrato in Figura 2.9, il commit **M1** è un vero e proprio *Merge Commit*. Esso possiede due genitori:

- Il commit **L1** (l'ultimo stato del proprio lavoro locale).
- Il commit **R1** (l'ultimo stato del server remoto).

Di conseguenza, i conflitti che possono emergere durante un `git pull` sono tecnicamente identici ai conflitti di un merge tra rami diversi. Il fatto che entrambi i puntatori si chiamino "main" è un dettaglio semantico irrilevante per l'algoritmo di Three-Way Merge di Git.

Capitolo 3

Collaborazione Distribuita e Modelli di Sviluppo

Fino a questo momento, abbiamo trattato Git come uno strumento per il versionamento individuale o per la sincronizzazione tra una macchina locale e un server di backup. Tuttavia, la vera potenza di Git, amplificata da piattaforme come GitHub, risiede nella sua natura collaborativa.

In questo capitolo esploreremo le dinamiche di lavoro in team, partendo da una distinzione fondamentale che spesso confonde i neofiti: la relazione tra la piattaforma tecnica e la licenza del software.

3.1 GitHub e l'Equivoco dell'Open Source

Nell'immaginario collettivo, GitHub è spesso sinonimo di Open Source. Sebbene la piattaforma ospiti la più grande comunità di software libero al mondo (incluso il kernel Linux, React, TensorFlow), questa associazione non è esclusiva né vincolante.

È necessario distinguere nettamente tra la **visibilità** del codice e i **diritti di utilizzo** dello stesso.

3.1.1 Codice Proprietario vs. Codice Aperto

GitHub funge da infrastruttura neutra, supportando due modelli di sviluppo diametralmente opposti:

1. **Sviluppo Proprietario (Closed Source):** La stragrande maggioranza delle aziende (da startup a colossi come Microsoft e Google) utilizza GitHub per ospitare il codice commerciale. Questi repository sono configurati come **Private**.
    - Il codice è inaccessibile al pubblico.
    - L'accesso è ristretto ai soli dipendenti o collaboratori autorizzati tramite invito.
    - La proprietà intellettuale è strettamente protetta e non vi è alcuna intenzione di condivisione con la comunità.
2. **Sviluppo Open Source (FOSS):** I progetti configurati come **Public** sono visibili a chiunque disponga di una connessione internet. Tuttavia, la semplice visibilità non rende un software "Open Source" nel senso legale del termine.

Se si pubblica un codice su un repository pubblico GitHub **senza specificare una licenza**, per le leggi internazionali sul diritto d'autore, quel codice è **"All Rights Reserved"**. Ciò significa che gli altri utenti possono *leggerlo* (perché reso pubblico), ma non hanno alcun diritto legale di scaricarlo, modificarlo, compilarlo o utilizzarlo nei propri progetti.

Affinché un progetto sia realmente Open Source, il proprietario deve includere un file `LICENSE` (es. MIT, Apache 2.0, GNU GPL) che conceda esplicitamente al pubblico i diritti di modifica e ridistribuzione.

3.1.2 Modello Social Coding

Indipendentemente dal fatto che il progetto sia chiuso o aperto, GitHub ha introdotto un paradigma definito *Social Coding*. Questo modello abbatte le barriere burocratiche alla contribuzione:

- In passato, per contribuire a un progetto, era necessario chiedere via email l'accesso in scrittura al server centrale.
- Con GitHub, chiunque può creare una copia personale del progetto (Fork), modificarla liberamente, e proporre le modifiche all'autore originale (Pull Request).

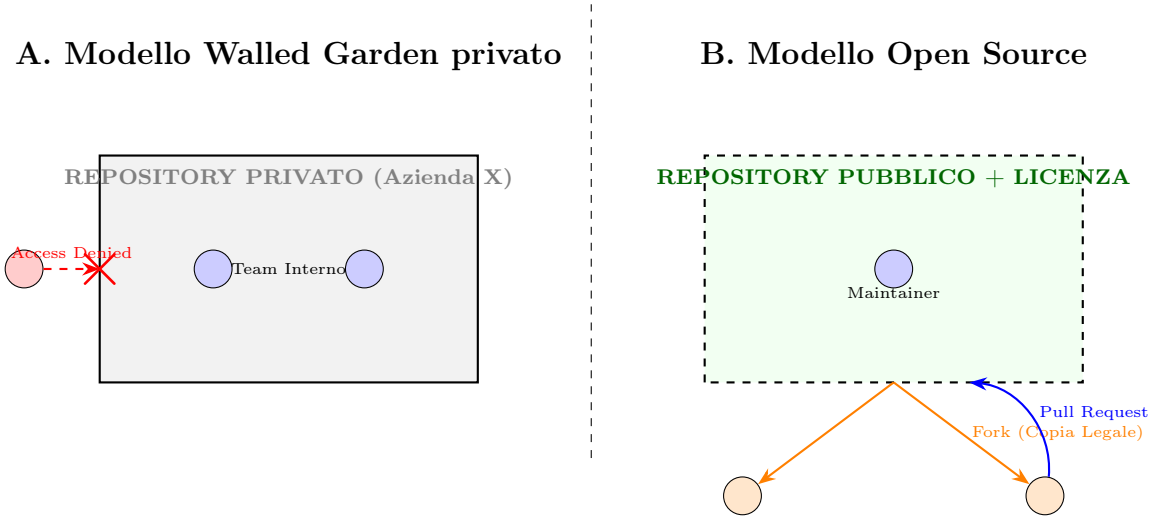


Figura 3.1: **Architettura della Collaborazione.** A sinistra: L'uso aziendale di GitHub, dove il codice è un asset segreto protetto da firewall logici. A destra: L'uso Open Source, dove la licenza (es. MIT) permette legalmente la duplicazione (Fork) e la contribuzione esterna.

In conclusione, GitHub è lo strumento abilitante, ma è la configurazione del repository (Privato vs Pubblico) e la scelta della licenza a determinare la natura del progetto. Un'azienda può utilizzare GitHub per anni senza mai produrre una sola riga di codice Open Source.

3.2 Il Flusso di Contribuzione Standard

Indipendentemente dalla natura del progetto (Open Source o Aziendale), una collaborazione efficace non inizia mai con la scrittura del codice, bensì con la definizione dell'obiettivo.

Il flusso di lavoro professionale su GitHub segue una rigida sequenza logica progettata per minimizzare lo spreco di risorse e prevenire conflitti architetturali. Questa sequenza può essere riassunta nel paradigma: **Discussione → Assegnazione → Esecuzione**.

3.2.1 L'Unità di Lavoro: GitHub Issues

Il punto di partenza per qualsiasi modifica al codice è la creazione (o l'individuazione) di una **Issue**. Sebbene il termine possa suggerire esclusivamente la segnalazione di problemi ("bug"), nel contesto di GitHub le Issues rappresentano l'unità atomica di pianificazione. Esse vengono utilizzate per:

- **Bug Reporting:** Segnalazione dettagliata di malfunzionamenti, corredata da passaggi per riprodurre l'errore.
- **Feature Request:** Proposte per l'aggiunta di nuove funzionalità o miglioramenti a quelle esistenti.
- **Refactoring:** Pianificazione di modifiche strutturali al codice che non alterano il comportamento esterno.
- **Domande e Discussioni:** Spazio per chiarimenti sull'architettura o sulle scelte progettuali.

In un contesto aziendale, le Issues fungono da "Ticket" di lavorazione (spesso integrate con Project Board stile Kanban); in un contesto Open Source, rappresentano il forum pubblico dove la comunità decide la direzione del progetto.

3.2.2 Protocollo di Avvio Lavori

Prima di aprire il proprio editor e iniziare a digitare, è imperativo seguire un protocollo di verifica per garantire che il proprio contributo sia utile e ben accetto.

1. Ricerca Preliminare (Avoid Duplicates)

La prima azione è sempre la ricerca. È altamente probabile che il bug riscontrato o la funzionalità desiderata siano già stati segnalati da un altro utente. Aprire una Issue duplicata disperde le informazioni e aumenta il carico di lavoro per i manutentori (*Maintainers*). Se esiste già una discussione aperta, è buona norma aggiungere un commento per confermare l'interesse o fornire ulteriori dettagli tecnici.

2. Definizione dello Scope e Assegnazione

Una volta aperta la Issue, si avvia la fase di negoziazione. I manutentori del progetto valuteranno la proposta:

- La funzionalità è in linea con la visione del progetto?
- L'approccio tecnico suggerito è corretto?

Solo dopo aver ricevuto un feedback positivo ("Green Light"), si procede all'assegnazione. GitHub permette di assegnare ufficialmente una Issue a un utente (*Assignee*). Questo passaggio è cruciale: segnala alla comunità che **qualcuno ci sta già lavorando**, evitando che due sviluppatori sprechino tempo a risolvere lo stesso problema parallelamente.

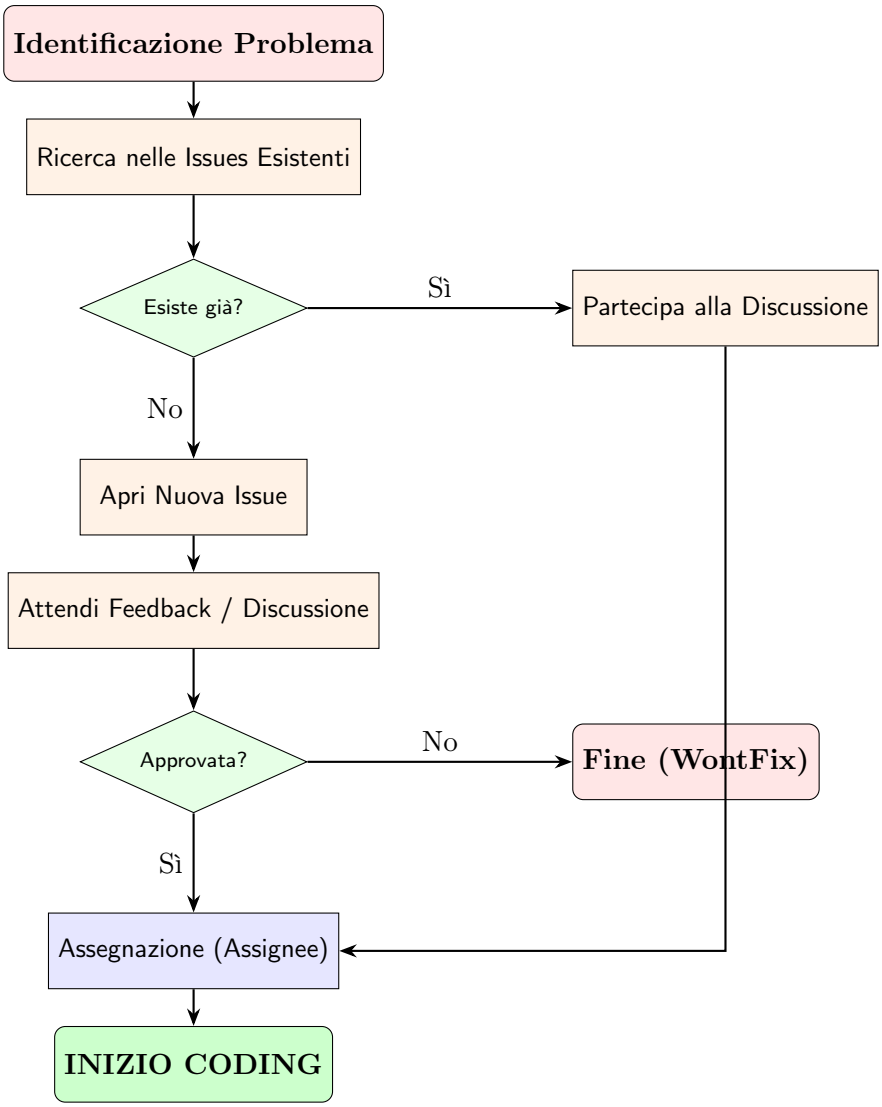


Figura 3.2: **Diagramma di Flusso Pre-Contribuzione.** Si noti come la fase di scrittura del codice avvenga solo al termine di un processo decisionale. Scrivere codice senza una Issue associata o assegnata è considerato una pratica scorretta (Bad Practice).

Best Practice: Nessuna PR senza Issue

Una regola non scritta (ma ferrea) in molti progetti è: *"No Pull Request without an Issue"*. Una PR che appare dal nulla, senza una discussione precedente che ne giustifichi l'esistenza, ha un'alta probabilità di essere chiusa o ignorata, poiché manca di contesto e validazione preliminare.

3.3 Dualità del Fork: Contribuzione vs Divergenza

Il meccanismo del *Forking* non è vincolato esclusivamente alla contribuzione verso il progetto genitore (Upstream). Quando si effettua un fork, GitHub crea una copia legale e completa del repository nel proprio spazio utente. Da quel momento, si possiede la totale sovranità su quella copia.

Esistono quindi due percorsi distinti che un Fork può intraprendere:

3.3.1 1. Il Fork Temporaneo (Workflow di Contribuzione)

È lo scenario descritto in precedenza. L'obiettivo è migliorare il progetto originale.

- **Durata:** Il fork è strumentale e spesso effimero.
- **Regole:** Si deve rispettare lo stile di codice e le linee guida del progetto originale.
- **Destinazione:** Le modifiche tornano alla base tramite **Pull Request**.

3.3.2 2. Il Fork Permanente (Hard Fork / Sviluppo Indipendente)

In questo scenario, lo sviluppatore decide di prendere il codice esistente come base di partenza per creare un prodotto diverso o una variante personale che non verrà mai riunita all'originale.

- **Libertà Totale:** Non è necessario aprire Issues o chiedere permessi. Essendo i proprietari della copia, è possibile stravolgere il codice, cambiare l'architettura o aggiungere feature che il progetto originale ha rifiutato.
- **Esempi Celebri:**
  - *LibreOffice* è nato come Hard Fork di *OpenOffice*.
  - *MariaDB* è un fork di *MySQL*.
  - *Ubuntu* deriva da *Debian*.

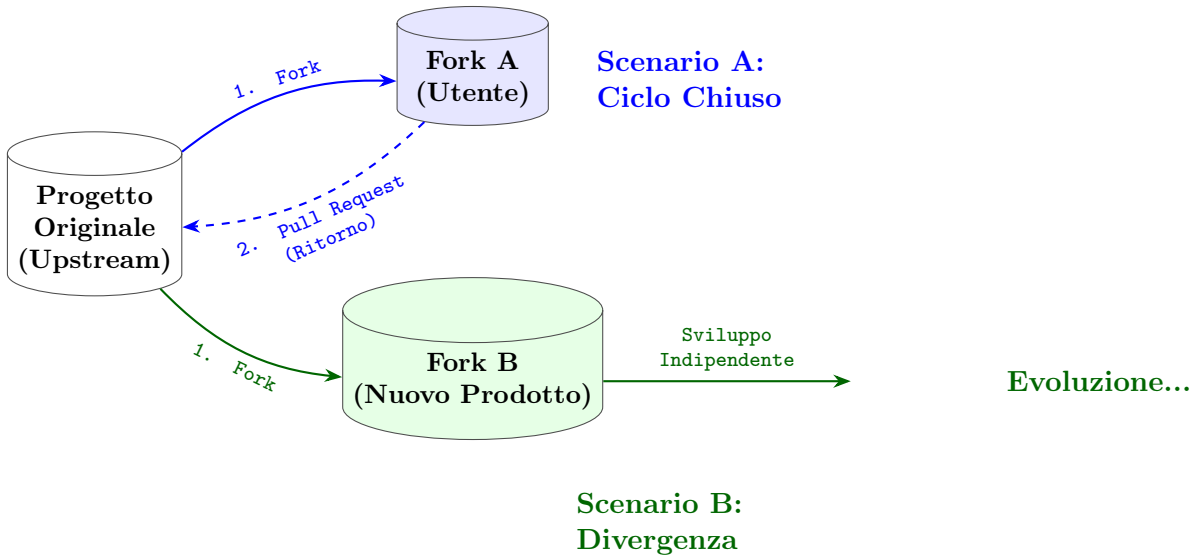


Figura 3.3: **Destini di un Fork.** Nel Scenario A, il fork serve solo come area temporanea per le modifiche destinate a tornare al padre. Nel Scenario B (Hard Fork), il legame si rompe e nasce un software completamente nuovo e indipendente.

Vincolo delle Licenze

Anche se tecnicamente è possibile gestire liberamente il proprio Fork, legalmente si deve rispettare la licenza del progetto originale. Se il progetto originale usa una licenza "Copyleft" (come la **GPL**), anche il proprio Fork dovrà rimanere Open Source e mantenere la stessa licenza. Se usa una licenza permissiva (come la **MIT**), si ha più libertà, incluso (a volte) chiudere il codice. Ma l'attribuzione all'autore originale è quasi sempre obbligatoria.

3.4 Caso di Studio: Il Ciclo di Vita di una Contribuzione

Per consolidare le nozioni teoriche esposte, si analizzerà ora la procedura completa per contribuire a un progetto esterno. Si ipotizzi il seguente scenario:

- **Progetto Target:** Una libreria Python open source per il calcolo statistico, denominata `OpenStats`.
- **Repository Originale (Upstream):** `university-libs/openstats`.
- **Il Problema:** La funzione `calculate_average()` genera un errore critico (divisione per zero) quando riceve in input una lista vuota.

Il flusso di lavoro si articola in cinque fasi operative distinte.

3.4.1 Fase 1: Apertura della Issue (Segnalazione)

Prima di scrivere qualsiasi riga di codice, è necessario documentare il malfunzionamento. L'utente si reca nel repository `university-libs/openstats`, verifica che il problema non sia già stato segnalato e apre una nuova **Issue**.

Una segnalazione professionale deve contenere:

1. **Titolo Chiaro:** Es. *"ZeroDivisionError in calculate_ average with empty list"*.
2. **Passaggi per Riprodurre (Reproduction Steps):** Il codice minimo necessario per innescare l'errore.
3. **Comportamento Atteso:** Cosa dovrebbe succedere (es. restituire 0 o `None`).

Dopo la discussione, un maintainer del progetto conferma il bug e assegna la Issue all'utente.

3.4.2 Fase 2: Preparazione dell'Ambiente (Fork & Clone)

Non avendo diritti di scrittura diretta sul repository `university-libs`, l'utente deve creare una copia personale.

1. **Fork:** Tramite l'interfaccia web di GitHub, si clicca sul pulsante "Fork". Questo crea una copia speculare nel profilo dell'utente: `utente-studente/openstats`.
2. **Clone:** Si scarica la copia locale sulla propria macchina di sviluppo.

```
# Clonazione del PROPRIO fork (non dell'originale)
git clone https://github.com/utente-studente/openstats.git
cd openstats
```

3.4.3 Fase 3: Isolamento e Correzione (Branching)

È considerata una *bad practice* lavorare direttamente sul ramo `main`, anche nel proprio fork. Si deve creare un ramo dedicato (Topic Branch) con un nome descrittivo.

```
# Creazione e attivazione del ramo di fix
git checkout -b fix/division-by-zero
```

A questo punto si effettua la modifica al codice sorgente (file `stats.py`):

Codice 3.1: Esempio di correzione del bug

```
def calculate_average(data):
    # Fix: Gestione del caso lista vuota
    if not data:
        return 0.0

    return sum(data) / len(data)
```

Dopo aver verificato la correzione con dei test locali, si procede al commit delle modifiche:

```
git add stats.py
git commit -m "Fix division by zero in average calculation (Closes #42)"
```

Automazione tramite Keywords

Si noti la stringa `(Closes #42)` nel messaggio di commit. GitHub riconosce parole chiave speciali come *Closes*, *Fixes* o *Resolves* seguite dal numero della Issue. Quando questo codice verrà integrato nel ramo principale, la Issue #42 verrà chiusa automaticamente.

3.4.4 Fase 4: Pubblicazione e Proposta (Push & Pull Request)

Le modifiche risiedono ora solo sul computer locale. È necessario caricarle sul proprio fork remoto (Origin).

```
# Push del ramo verso il proprio fork su GitHub
git push -u origin fix/division-by-zero
```

Subito dopo il push, l'interfaccia web di GitHub mostrerà un banner: *"Compare & pull request"*. Cliccando su tale pulsante, si avvia la procedura formale di richiesta integrazione:

- **Base Repository:** `university-libs/openstats` (branch `main`).
- **Head Repository:** `utente-studente/openstats` (branch `fix/division-by-zero`).

3.4.5 Visualizzazione del Flusso Architetturale

La complessità di questa operazione risiede nel coinvolgimento di tre copie distinte del repository. La Figura 3.4 illustra il percorso circolare dei dati.

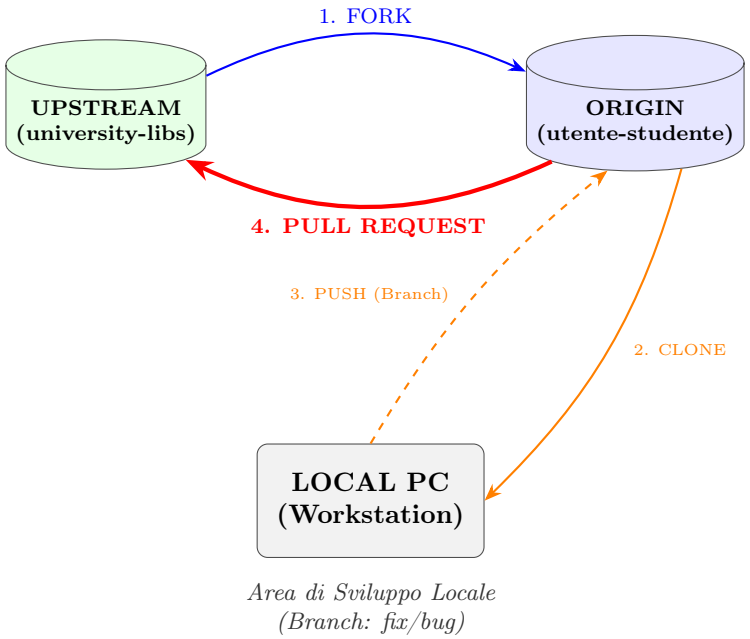


Figura 3.4: **Topologia della Pull Request.** Il ciclo prevede quattro passaggi distinti. Si noti come le operazioni di scrittura (Push) avvengano esclusivamente verso il proprio Fork (Origin), mentre l'Upstream riceve solo la richiesta di integrazione finale.

Una volta aperta la Pull Request, inizia la fase di Code Review. I manutentori potrebbero richiedere modifiche. In tal caso, l'utente dovrà semplicemente modificare i file in locale, fare un nuovo commit e un nuovo push sullo stesso ramo: la Pull Request su GitHub si aggiornerà automaticamente con le nuove modifiche.

3.5 Manutenzione del Fork

Una volta effettuato il fork, la copia personale (`origin`) e il progetto originale (`upstream`) iniziano a vivere vite separate. Mentre si lavora localmente, il progetto originale continua ad evolversi grazie ai contributi di altri sviluppatori.

Se il proprio sviluppo dura più di qualche giorno, è matematico che il proprio codice diventi obsoleto (*stale*), portando a inevitabili conflitti al momento della Pull Request. Per evitare ciò, è necessario configurare una connessione diretta con il repository originale per scaricare gli aggiornamenti.

3.5.1 Configurazione del Remote "Upstream"

Di default, quando si clona un fork, Git configura un solo server remoto chiamato `origin` (il vostro fork). È necessario aggiungere manualmente l'indirizzo del repository originale, che per convenzione viene chiamato **upstream**.

```
# Verifica dei remote attuali
git remote -v
# Output: origin  https://github.com/tuo-utente/repo.git (fetch/push)

# Aggiunta del repository originale come 'upstream'
git remote add upstream https://github.com/autore-originale/repo.git

# Verifica finale
git remote -v
```

```
# Output:
# origin    https://github.com/tuo-utente/repo.git (fetch/push)
# upstream  https://github.com/autore-originale/repo.git (fetch/push)
```

3.5.2 La Procedura di Sincronizzazione

La sincronizzazione non avviene mai direttamente tra i due repository su GitHub. I dati devono passare attraverso il computer locale dell'utente. La procedura corretta per aggiornare il proprio ramo `main` con le ultime novità dell'autore originale è la seguente:

```
# 1. Scaricare i metadati dal repository originale (senza unire nulla)
git fetch upstream

# 2. Spostarsi sul ramo principale locale
git checkout main

# 3. Unire le modifiche scaricate nel ramo locale
git merge upstream/main
# (A questo punto il PC è aggiornato, ma il GitHub no)

# 4. Inviare gli aggiornamenti al fork su GitHub
git push origin main
```

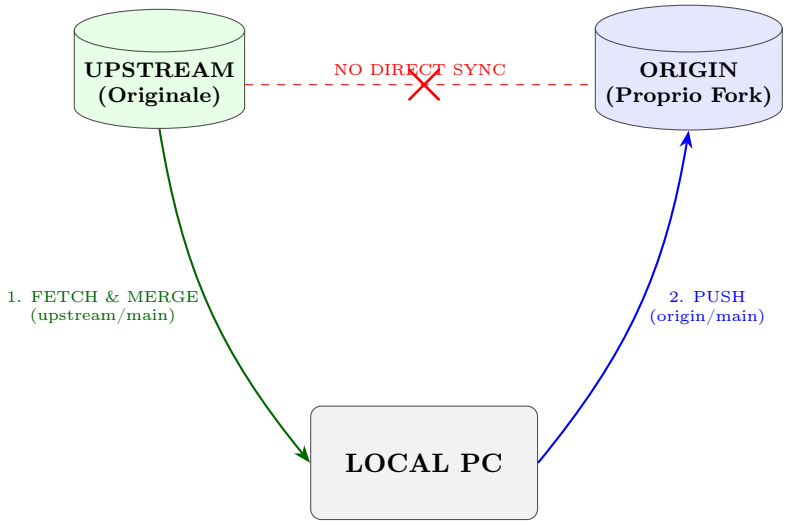


Figura 3.5: **Triangolazione degli Aggiornamenti.** Un errore comune è pensare che il tasto "Sync Fork" sulla web UI sia l'unico modo. Da riga di comando, i dati fluiscono dall'Upstream al PC locale, e solo successivamente vengono spinti verso il Fork (Origin).

3.6 Gestione dei Conflitti di Merge

Nel contesto di un sistema di versionamento distribuito, la parallelizzazione del lavoro comporta inevitabilmente situazioni in cui le modifiche apportate da due attori distinti risultano incompatibili. Tale evenienza è definita **Conflitto di Merge**.

Git dispone di algoritmi sofisticati per l'unione automatica (*auto-merging*), capaci di integrare modifiche che interessano file diversi o porzioni distanti dello stesso file. Tuttavia, quando due rami propongono variazioni concorrenti alla medesima riga di codice, l'algoritmo sospende l'operazione, richiedendo un intervento manuale esplicito per determinare quale versione debba prevalere.

3.6.1 Struttura di un Conflitto

Quando Git rileva un'incompatibilità durante un comando di `merge` o `pull`, modifica fisicamente i file interessati, inserendo dei marcatori testuali speciali che delimitano le aree di divergenza. Il file non viene sovrascritto ciecamente, ma viene "annotato" con entrambe le versioni del codice.

Un file in stato di conflitto presenta la seguente struttura tripartita:

Codice 3.2: Esempio di file con marcatori di conflitto

```
def calcola_totale(prezzo, tasse):
<<<<<<< HEAD
    # Versione presente nel ramo locale (destinazione)
    return prezzo * (1 + tasse / 100)
=======
    # Versione proveniente dal ramo remoto (sorgente)
    return prezzo + (prezzo * tasse)
>>>>>>> feature/nuova-formula
```

La sintassi dei marcatori è così definita:

- `<<<<< HEAD`: Indica l'inizio della porzione di codice presente nel ramo corrente (quello su cui ci si trovava prima del merge).
- `=======`: Agisce da separatore tra le due varianti.
- `>>>>> branch-name`: Indica la fine del blocco in conflitto e specifica la provenienza della modifica entrante.

3.6.2 Protocollo di Risoluzione

La risoluzione di un conflitto non è un'operazione tecnica automatizzabile, bensì una decisione logica e semantica. L'operatore deve seguire una procedura in quattro fasi:

1. **Identificazione:** Il comando `git status` elenca i file nello stato *"Unmerged paths"*.
2. **Modifica Manuale:** Si apre il file con un editor di testo. L'operatore deve analizzare le due varianti e decidere come procedere:
   - Mantenere solo la versione locale (HEAD).
   - Accettare solo la versione remota (Incoming).
   - Creare una sintesi delle due (spesso necessario se entrambe le modifiche sono valide ma parziali).

   È imperativo rimuovere completamente i marcatori (`<<`, `===`, `>>`) prima di salvare.
3. **Mark as Resolved:** Una volta salvato il file "pulito", è necessario comunicare a Git che il conflitto è stato risolto aggiungendo il file all'area di staging:

   ```
   git add nome_file.py
   ```

4. **Commit di Fusione:** La procedura si conclude generando un commit che sancisce l'unione delle due storie.

   ```
   git commit -m "Merge branch 'feature' - Resolved conflicts in
       calculation"
   ```

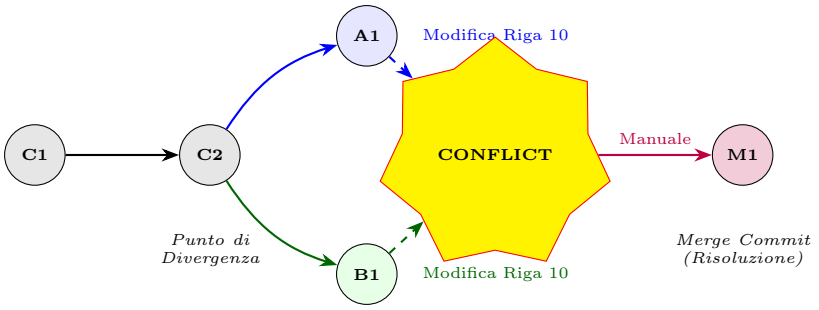


Figura 3.6: **Dinamica del Conflitto.** Quando due rami modificano concorrentemente la stessa area semantica (es. Riga 10) partendo da un antenato comune (C2), Git non possiede l'autorità logica per scegliere un vincitore. Il processo di fusione si arresta (stella gialla) finché l'utente non genera il commit di risoluzione (M1).

Nota su `-abort`

Se durante la risoluzione di un conflitto ci si rende conto di aver commesso un errore o di non avere le competenze per decidere quale versione mantenere, è possibile annullare l'intera operazione di merge e tornare allo stato precedente con il comando: `git merge -abort`

3.7 L'importanza del File README.md

Nello sviluppo software moderno, la qualità del codice è condizione necessaria ma non sufficiente per il successo di un progetto. Un repository privo di documentazione, per quanto tecnicamente eccellente, risulta inaccessibile a terzi e, nel lungo periodo, persino al suo stesso autore.

Il file `README.md` rappresenta l'ingresso principale (*entry point*) di ogni repository. GitHub lo riconosce automaticamente e ne renderizza il contenuto nella pagina principale del progetto. La sua funzione è duplice:

- **Funzione Descrittiva:** Spiega lo scopo del software ("Cosa fa").
- **Funzione Operativa:** Istruisce l'utente sull'utilizzo ("Come si usa").

In ambito accademico e professionale, la presenza di un README curato è spesso il primo indicatore della maturità ingegneristica di uno sviluppatore.

3.7.1 Standard Tecnologico: Markdown

Il formato standard per la documentazione su piattaforme di versionamento è il **Markdown** (estensione `.md`). Si tratta di un linguaggio di markup leggero che permette di formattare il testo utilizzando una sintassi semplice e leggibile, che viene poi convertita in HTML dal motore di rendering di GitHub.

A differenza di formati binari (come `.docx` o `.pdf`), il Markdown è testo puro (*plain text*), rendendolo perfettamente compatibile con il versionamento di Git: le modifiche alla documentazione possono essere tracciate riga per riga esattamente come il codice sorgente.

3.7.2 Anatomia di un README Professionale

Un documento efficace deve rispondere a domande precise. La struttura standard adottata dalla comunità open source prevede le seguenti sezioni:

1. **Titolo e Descrizione:** Un "Elevator Pitch" che riassume in due righe il problema risolto dal software.

2. **Prerequisiti (Dependencies):** Quali librerie o interpreti sono necessari (es. Python 3.9+, Node.js).
3. **Installazione:** La sequenza esatta di comandi da terminale per configurare l'ambiente.
4. **Utilizzo (Usage):** Esempi concreti di codice o screenshot che mostrano il software in azione.
5. **Contributi:** Linee guida per chi vuole collaborare (spesso rimanda a un file `CONTRIBUTING.md`).
6. **Licenza:** Il quadro legale di utilizzo (es. MIT, GPL, Apache).

3.7.3 Esempio di Sintassi Markdown

Di seguito viene riportato un esempio pratico di come si struttura il codice sorgente di un README, mettendo in evidenza gli elementi sintattici più comuni: intestazioni, blocchi di codice e liste.

Codice 3.3: Esempio di file README.md raw

````
# Nome del Progetto
> Una breve descrizione o slogan del progetto.

## Indice
- [Installazione](#installazione)
- [Utilizzo](#utilizzo)

## Installazione
Clona il repository e installa le dipendenze:

```bash
git clone [https://github.com/utente/progetto.git](https://github.com/
    utente/progetto.git)
cd progetto
pip install -r requirements.txt
```

## Utilizzo
Esempio di utilizzo:

```python
import progetto
progetto.main()
```

## Contributi
Per contribuire al progetto, segui le linee guida riportate in [
    CONTRIBUTING.md](CONTRIBUTING.md).

## Licenza
Questo progetto è rilasciato sotto la licenza MIT.
````

Un buon README deve rendere il progetto **autosufficiente**. Se un nuovo sviluppatore deve contattare l'autore originale per capire come compilare il progetto, la documentazione ha fallito il suo scopo primario.