Lezione 1

Tutorato di Ingegneria del Software

Simone Colli

3 novembre 2025

Indice

Contenuto della lezione

- Presentazione del progetto.
- Git e GitHub: flusso di lavoro, tipi di merge, risoluzione conflitti, convenzioni.
- Creazione repository GitHub, prima struttura della cartella di progetto,
- HTML/HTML5: struttura semantica di base.
- Creazione layout di base dell'app.

Presentazione del progetto

Il progetto consiste nello **sviluppo incrementale** di un'applicazione web per la gestione di attività (*Task Manager*). Il progetto verrà sviluppato a ogni lezione, aggiungendo funzionalità progressivamente, fino a ottenere un'applicazione completa e architetturalmente corretta.

L'applicazione permetterà di creare, modificare, completare e cancellare task. Ogni task dovrà avere un titolo, una descrizione, una data di scadenza, una priorità ed una (o più) categoria. I task potranno essere ordinati per priorità o data di scadenza, e raggruppati per categorie.
Il software sviluppato dovrà garantire la persistenza dei dati tramite un database, che verrà utilizzato per memorizzare le informazioni sui task e le loro categorie.

Il software dovrà essere sviluppato **senza l'uso di framework** esterni (es. Node, React, Laravel, ecc...), per permettere una comprensione più profonda dei concetti applicati.

Note introduttive

- Le slide e il materiale visto durante le lezioni sono disponibili su:
  - Pagina elly del corso.
  - Repository GitHub:
    `https://github.com/unipr-org/tutorati/IdS/2025/`
    (creata dopo la prima lezione)
- Per dubbi e domande, potete contattarmi via email a:
  `simone.colli@studenti.unipr.it`

Flusso di lavoro tipico

- **Branch naming**: `...`, `fix/...`, `docs/...`
- Ogni modifica va fatta su una branch, mai direttamente su `main`.
- La **Pull Request (PR)** serve per review e tracciabilità.
- Dopo il merge: **eliminare la branch** e sincronizzare (`git pull origin main`).

Panoramica sui tipi di merge

Obiettivo: integrare modifiche da una branch (es. `x`) in `main`.

Fast-forward (FF) `main` avanza linearmente: nessun merge commit, storia lineare.
Quando: `main` non ha nuovi commit dopo la creazione della feature branch.

Three-way merge Crea un *merge commit* con due genitori, preservando la storia parallela.
Quando: `main` e la feature branch sono divergenti.

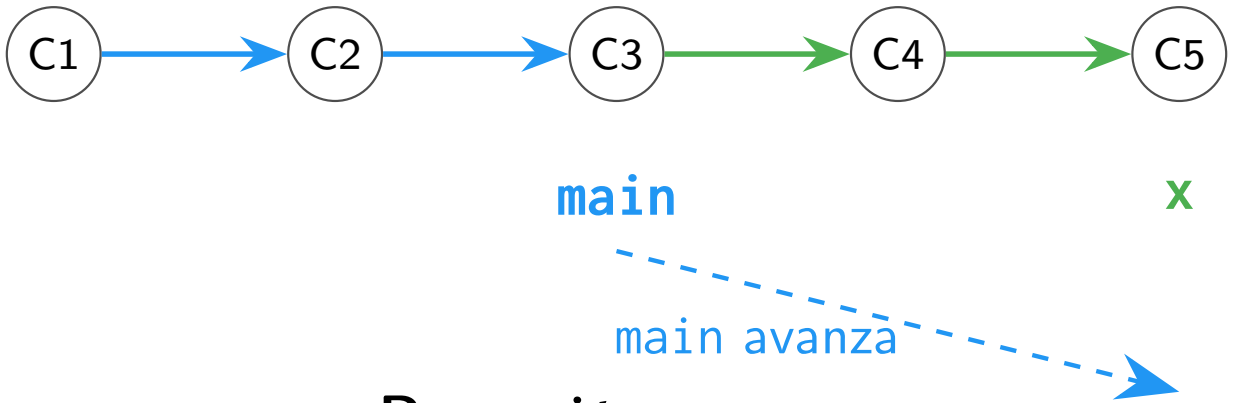
No fast-forward (–no-ff) Forza la creazione del merge commit anche se FF sarebbe possibile.
Perché: maggiore tracciabilità delle integrazioni di feature.

Squash merge Comprime tutti i commit della feature in uno solo su `main`.
Quando: storia lineare e pulita, sacrificando i dettagli dei singoli commit.

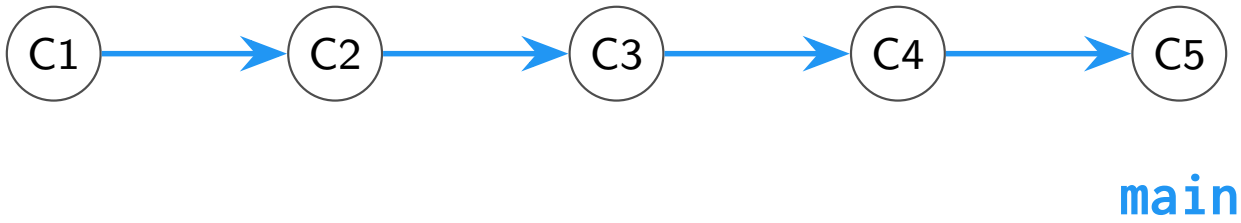
Rebase Riscrive la base della feature branch, spostando i commit sopra `main`.

Fast-forward (FF)

Prima del merge:

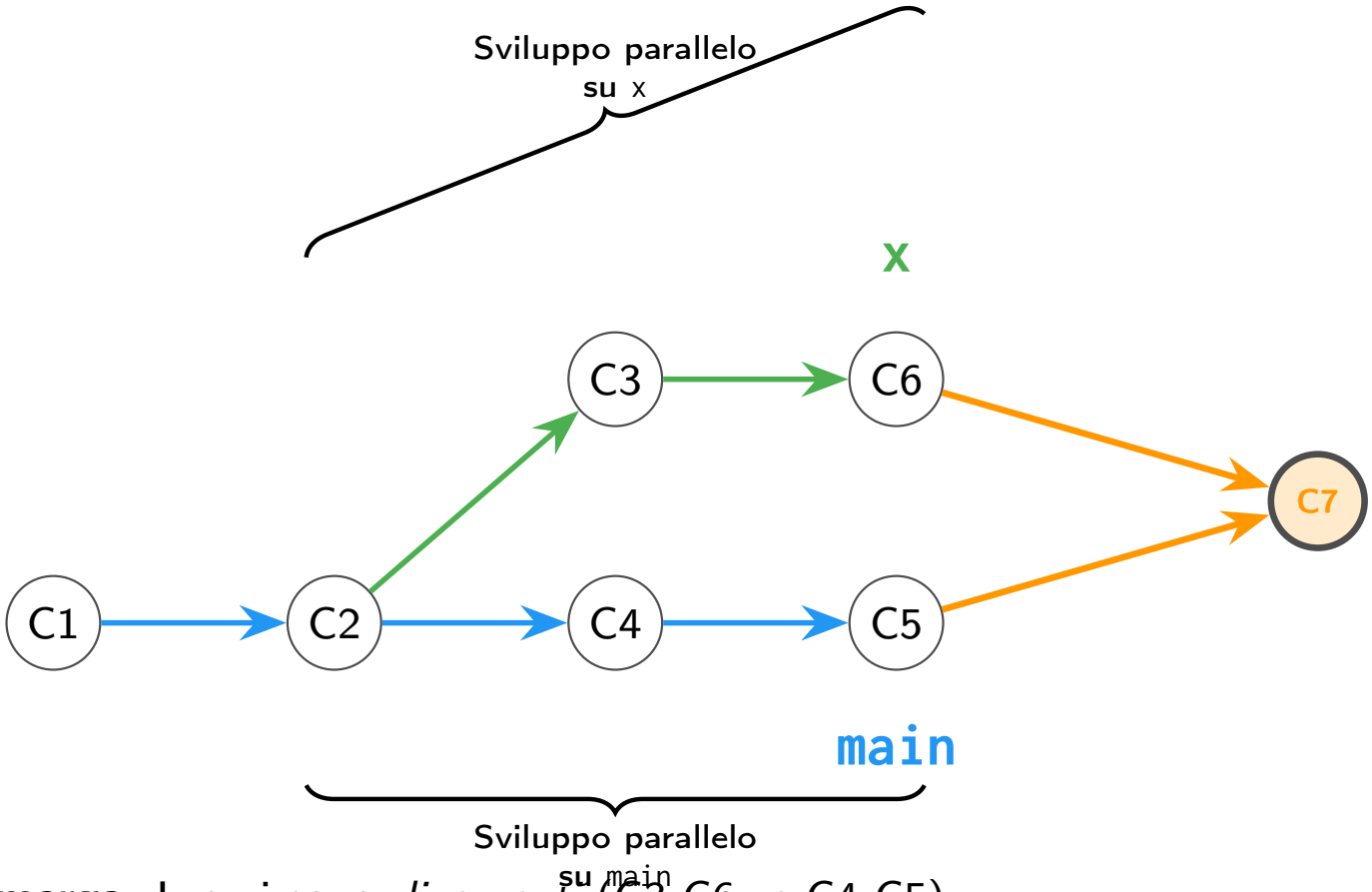


Dopo `git merge x`:



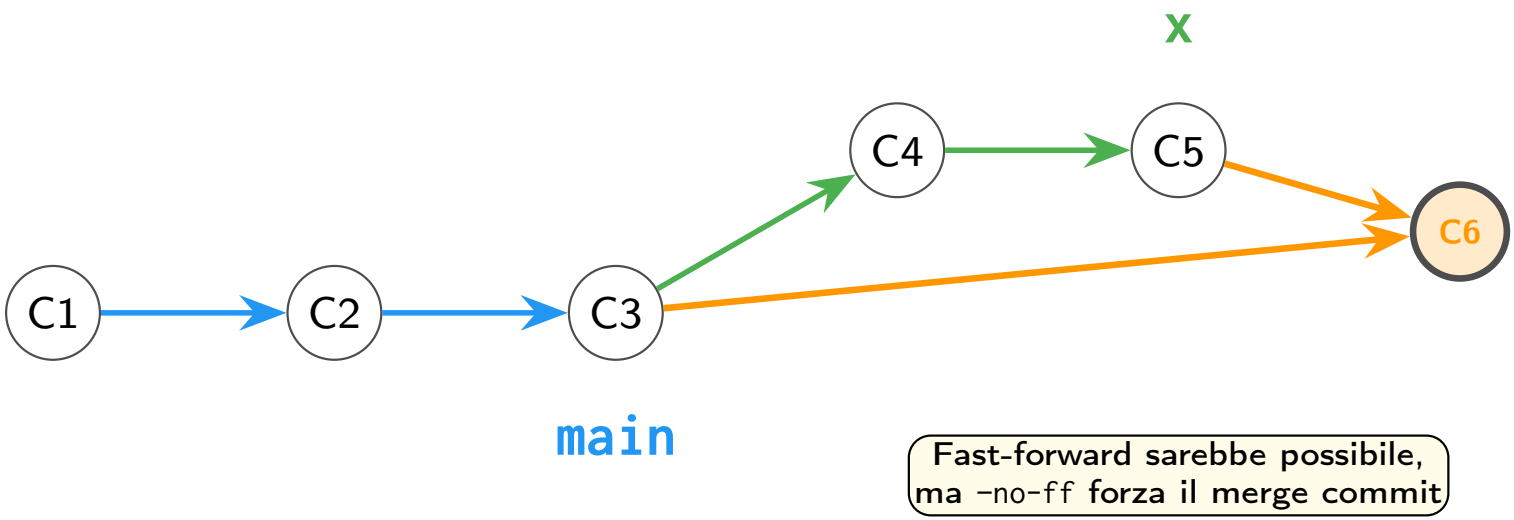
Fast-forward: Il puntatore `main` si sposta semplicemente in avanti. Nessun merge commit necessario perché la storia è lineare.

Three-way merge (merge commit)



Three-way merge: I rami sono *divergenti* (C3-C6 vs C4-C5).
Git crea un commit speciale (C7) con **due genitori**, integrando entrambe le linee di sviluppo.

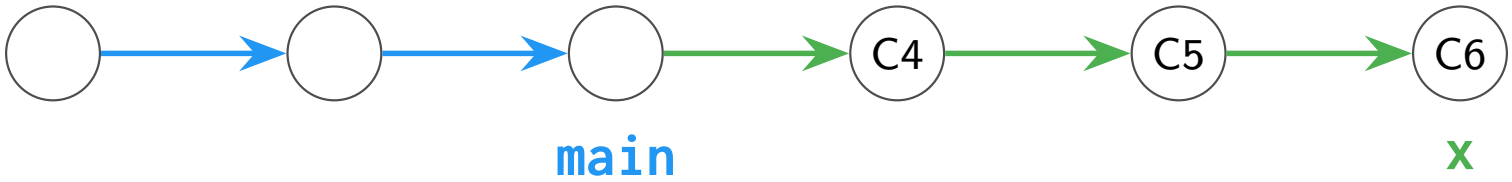
No fast-forward (–no-ff)



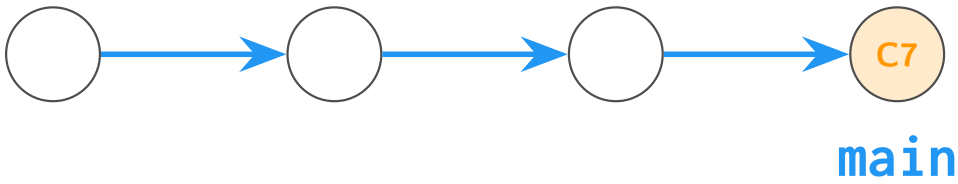
No fast-forward: Anche se la storia è lineare, si crea comunque un merge commit.
Vantaggio: Maggiore tracciabilità dell'integrazione della feature.

Squash merge

Prima del merge:

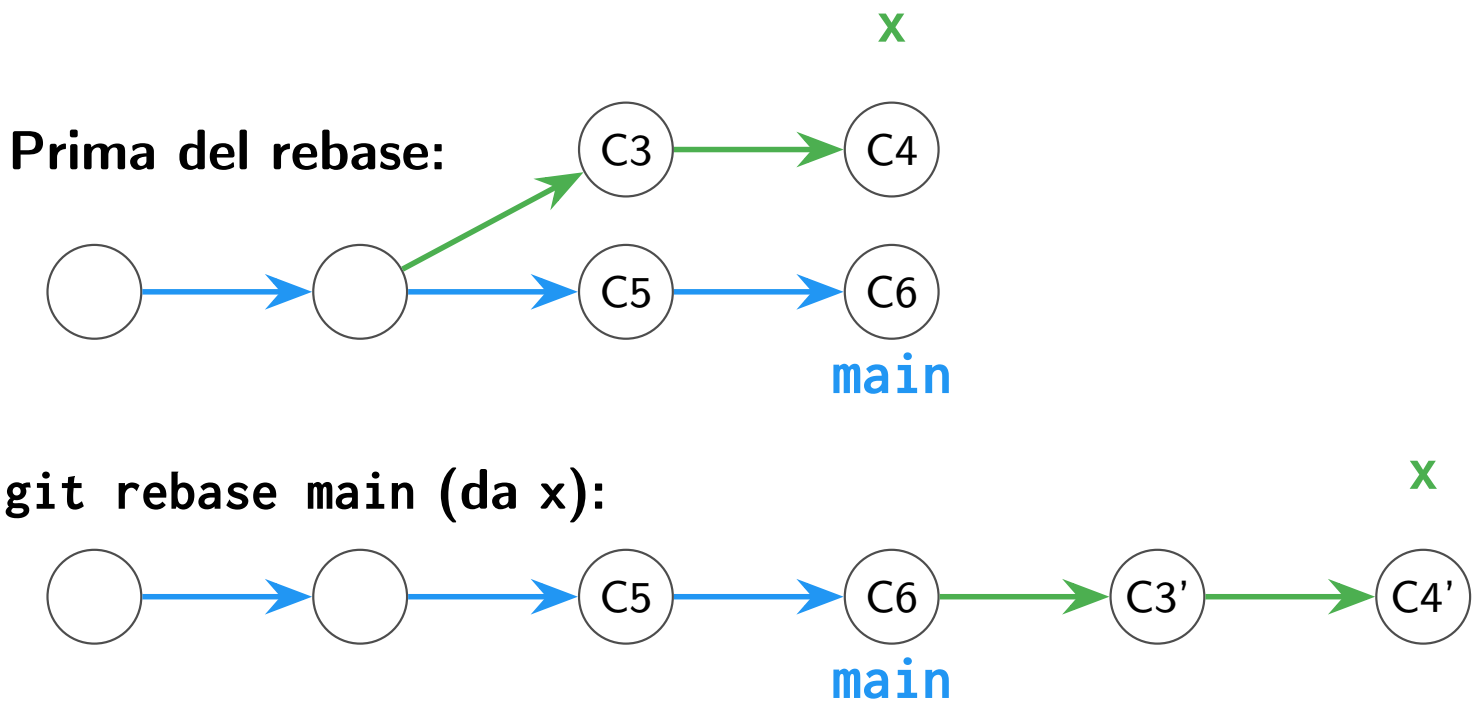


Dopo `git merge --squash x`:



Squash merge: Tutti i commit della feature (C4, C5, C6) vengono *compressi* in un unico commit (C7) su `main`. Storia lineare, ma si perdono i dettagli.

Rebase (alternativa al merge)



Rebase: I commit della feature (C3', C4') vengono *riscritti* sopra `main`.
Storia lineare, ma **attenzione**: non usare su branch già condivise!

Scelte pratiche

- **Progetto didattico/di squadra**: preferire **merge commit** o **no-ff** per visibilità dell'integrazione.
- **Repo pubblica rumorosa**: valutare **squash merge** per mantenere la storia pulita.
- **Pulizia locale prima della PR**: `git rebase -i` sulla propria branch (evitare su branch già condivise).

Flusso consigliato

1. **Branch di feature** `git checkout -b todo-layout`
2. **Commit piccoli e frequenti** con messaggi chiari.
3. **Apri una PR** verso `main`.
4. **Review & checks**.
5. **Merge**: merge commit / no-ff / squash.
6. **Elimina la branch** dopo l'integrazione.

Conventional commits (versione breve)

```
feat: aggiunto layout base
fix: corretto stile header
docs: aggiornato README
refactor: rinominati componenti
chore: aggiornato .gitignore
```

Consigliato per messaggi di commit chiari e standardizzati. Il contenuto dei messaggi deve essere informativo e coerente con il contenuto.

.gitignore e .gitkeep

- In `.gitignore`: file temporanei editor, artefatti di build, `.env`.
- `.gitkeep`: file vuoto per mantenere in repo cartelle altrimenti vuote.
- (Opz.) `.gitattributes`: normalizzare EOL, es. `* text=auto`.

Esercizio: creare la repo, README e prima branch

Obiettivo: Creare un repository GitHub per il progetto Task Manager.

1. **Crea repository su GitHub**
   - Nome: `task-manager` (o simile)
   - Visibilità: Public o Private
   - Aggiungi README.md, `.gitignore`, `.gitkeep`
2. **Clone locale e struttura**

```
git clone https://github.com/tuo-username/task-manager.git
cd task-manager
mkdir css js
touch index.html css/style.css js/script.js
```

Esercizio: creare la repo, README e prima branch

Obiettivo: Creare un repository GitHub per il progetto Task Manager.

1. **Crea branch per il layout**

```
git checkout -b initial-layout
git add .
git commit -m "feat: added initial project structure"
git push -u origin initial-layout
```

2. **Apri una pull request** su GitHub verso `main`

Gestione conflitti (procedura)

1. `git merge x`
2. Risolvi i file con marker <<<<<<< ======= >>>>>>>
3. `git add <file>` per ciascun file risolto
4. `git commit` per completare il merge
5. In caso di problemi: `git merge -abort`

Esercizio: Simulazione conflitto (5–10 minuti)

```
git checkout -b fake_feature
echo "<header>ToDo</header>" >> index.html
git add index.html && git commit -m "feat: added header"

git checkout main
echo "<footer>v1</footer>" >> index.html
git add index.html && git commit -m "feat: added footer"

git merge fake_feature
# risolvi i conflitti in index.html, poi:
git add index.html
git commit
```

Varianti: ripeti con `--no-ff`; prova `git rebase main` da header prima del merge.

Cheat-sheet essenziale

```
# Branch
git checkout -b x
git switch -c x   # alternativa moderna

# Merge (diverse opzioni)
git checkout main
git merge x             # three-way o FF
git merge --no-ff x     # forza merge commit

# Squash (via PR GitHub) o locale
git merge --squash x
git commit -m "feat: integra x"

# Rebase (attenzione su branch condivise)
git checkout x
git rebase main
```

Git pull: Sincronizzare con il repository remoto

Cos'è git pull? Comando che combina due operazioni: `git fetch` seguito da `git merge`. Scarica le modifiche dal repository remoto e le integra nel branch corrente.

Quando usarlo? Ogni volta che vuoi aggiornare il tuo branch locale con le ultime modifiche dal repository remoto, tipicamente prima di iniziare a lavorare o prima di fare un push.

Comportamento default Esegue un three-way merge se ci sono modifiche divergenti tra local e remoto, creando un merge commit.

Sintassi

```
git pull [remote] [branch]
```

Esempio: `git pull origin main`

Git pull: Opzioni principali

`git pull -ff-only` Esegue il pull solo se possibile un fast-forward. Se ci sono divergenze, il comando fallisce. Utile per evitare merge commit indesiderati.

`git pull -rebase` Invece di fare merge, esegue rebase delle tue modifiche locali sopra quelle remote. Mantiene una storia lineare.

`git pull -no-commit` Esegue il merge ma non crea automaticamente il commit permettendoti di rivedere le modifiche prima di confermare.

Best practice

Configurare il comportamento default di pull:

```
git config pull.rebase false # merge (default)
git config pull.rebase true # rebase
git config pull.ff only # fast-forward only
```

Git pull vs Git fetch + Git merge

Git pull

```
git pull origin main
```

Una singola operazione che:

- Scarica le modifiche
- Le integra automaticamente
- Può creare conflitti da risolvere

Git fetch + merge

```
git fetch origin
git merge origin/main
```

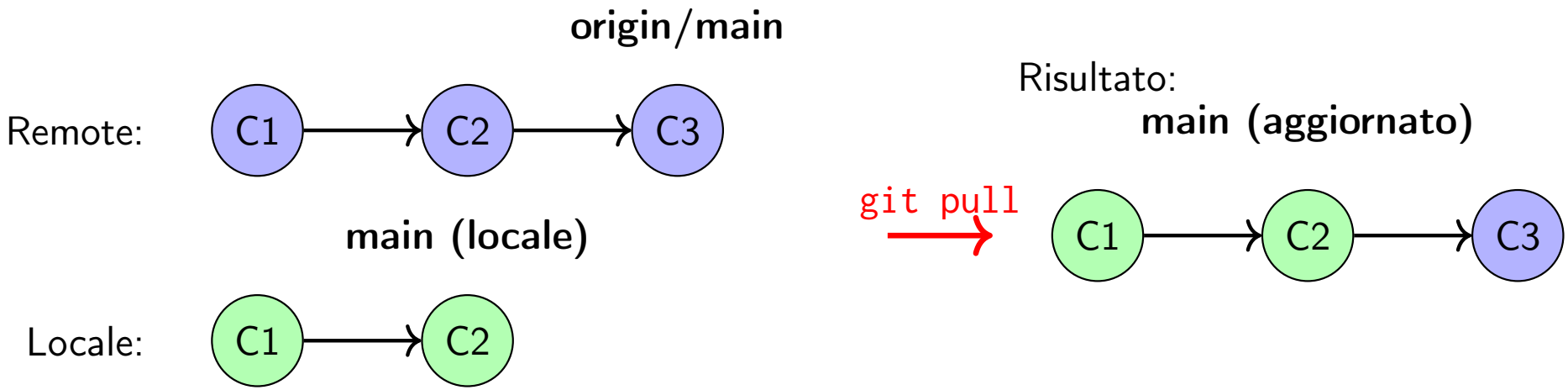
Due operazioni separate:

- Prima scarica le modifiche
- Permette di ispezionarle
- Poi decide come integrarle
- Più controllo sul processo

Quando usare fetch invece di pull?

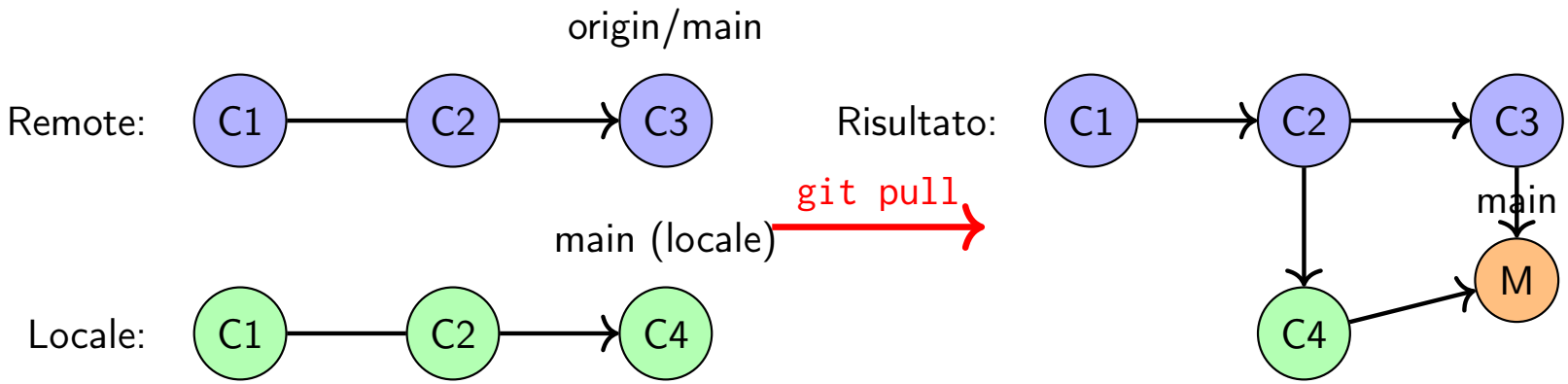
Usa `git fetch` quando vuoi vedere cosa è cambiato nel remoto prima di integrare le modifiche, o quando lavori su feature critiche e vuoi massimo controllo.

Git pull: Scenario tipico



In questo caso, il pull esegue un **fast-forward merge** perché non ci sono modifiche locali divergenti.

Git pull con conflitti



Quando ci sono modifiche divergenti, `git pull` crea un **merge commit** (M).

HTML5: Da Div alla semantica

Approccio HTML4

```
<div id="header">
  <div id="nav">...</div>
</div>

<div id="content">
  <div class="article">
    <div class="article-header"
    >
      <h2>Titolo</h2>
    </div>
    <div class="article-content
    ">
      <p>Contenuto...</p>
    </div>
  </div>
</div>
```

Approccio HTML5 (Semantico)

```
<header>
  <nav>...</nav>
</header>

<main>
  <article>
    <header>
      <h2>Titolo</h2>
    </header>
    <section>
      <p>Contenuto...</p>
    </section>
  </article>
</main>

<aside>...</aside>
<footer>...</footer>
```

Elementi semantici HTML5

`<header>`	Intestazione di pagina/sezione. Contiene titolo, logo, nav. *Può essere ripetuto*.
`<nav>`	Blocco di navigazione principale (menu, breadcrumb, filtri).
`<main>`	Contenuto principale della pagina. **Uno solo per pagina**.
`<article>`	Contenuto autonomo e riutilizzabile (post blog, task, prodotto).
`<section>`	Raggruppamento tematico di contenuti con heading.
`<aside>`	Contenuto correlato ma secondario (sidebar, note).
`<footer>`	Piè di pagina con info copyright, contatti, link. *Può essere ripetuto*.