

# VCS-GIT

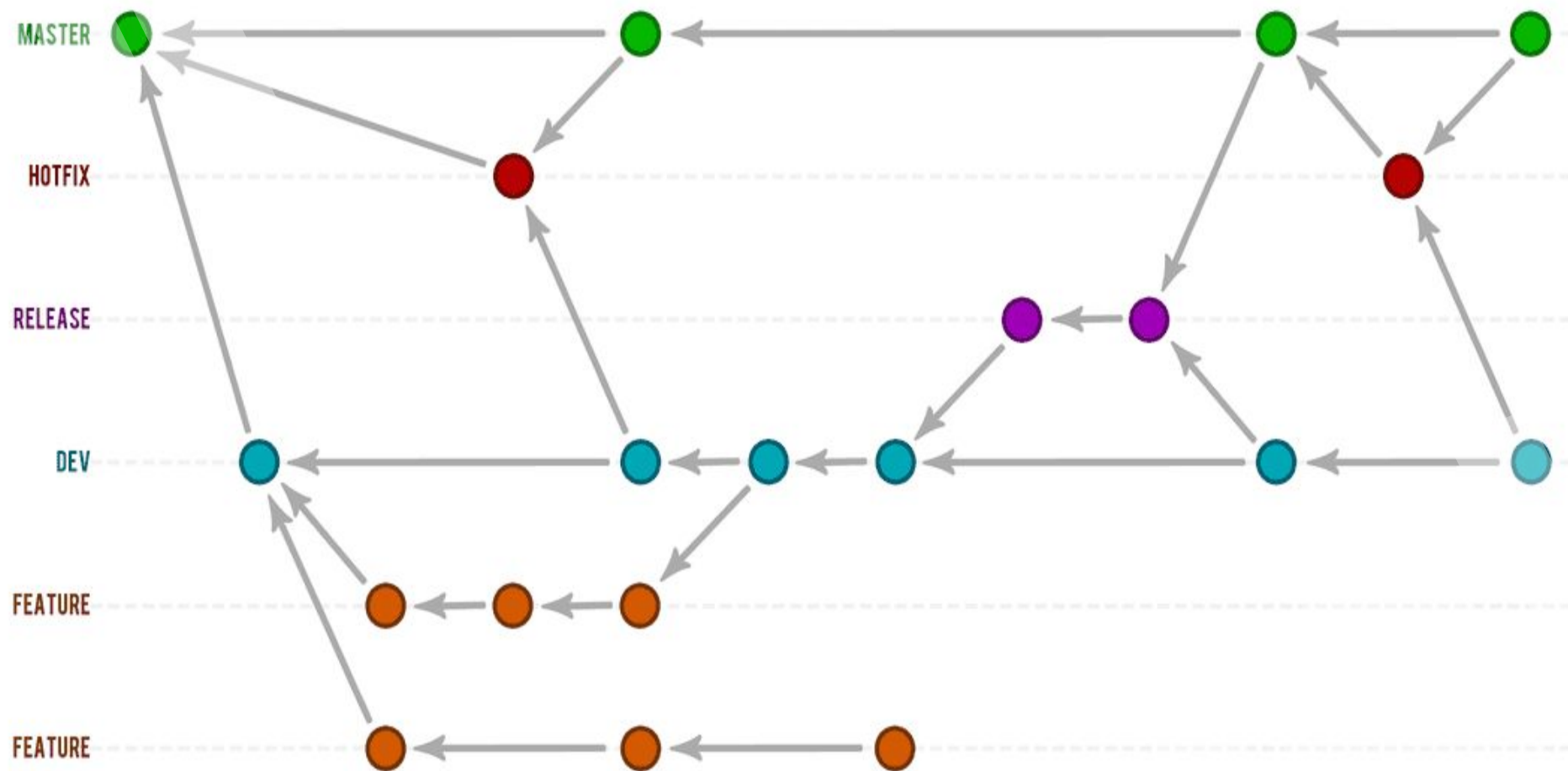
## 5-Branch, merge, fork

Prof. Marcello Missiroli



## DI CHE SI PARLA?

In questo (ultima) parte, vedremo un po' di argomenti avanzati di Git. I più sono basati con il concetto di branch/merge, che permettono uno sviluppo più agile e parallelo. Vedremo anche un accenno ai cosiddetti "flussi di lavoro" che si posso usare con git. Infine un accenno sugli aspetti avanzati di logging e "contabilità".





1.

**Branch**



## Di che si tratta

Finora tutte le modifiche confluiscono in un unico prodotto finale, e ogni volta può essere necessario risolvere dei conflitti - magari inutilmente se il lavoro non è definitivo.

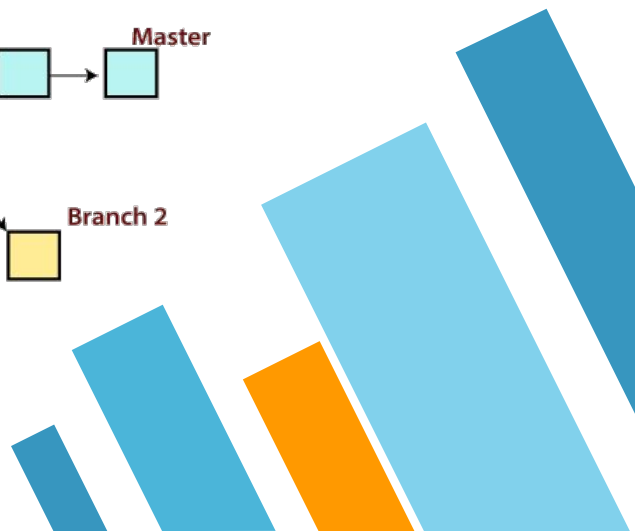
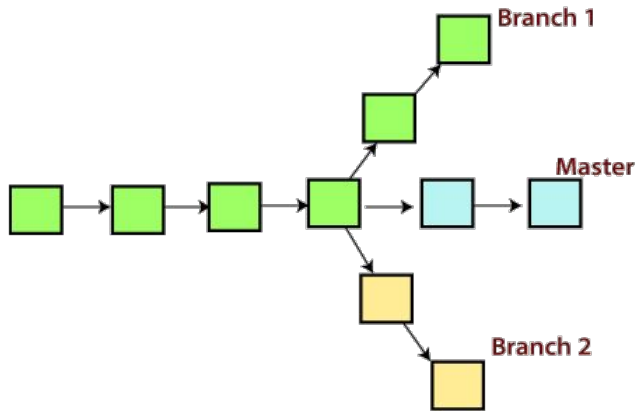
Si può però lavorare su versioni separate dei file per evitare di “sporcare” il lavoro degli altri (o proprio)-. IN pratica si ha la libertà di lavorare, provare, “sporcare” il codice senza compromettere lavori di altri e/o definitivi.

# I branch

Per questo si usano i branch, una diramazione del codice; i commit relativi a questa diramazione resteranno del tutto isolati rispetto alla versione principale, quella chiamata “main” o “master”, dandoci totale libertà di lavoro. Quando si è pronti, si può la diramazione al tronco principale, e solo in quel momento ci dovremo occupare degli eventuali conflitti.

## Vantaggi

I branch sono veloci da creare, occupano pochissimo spazio. È buona abitudine imparare a farne uso. E basta imparare due soli comandi (branch e merge).





# 2.

## Branch locali





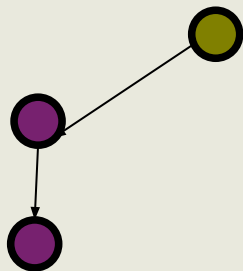
## Di nuovo il librone sacro

Questa volta Bob e Tim decidono di lavorare su aspetti separati: Bob si occuperà dei precetti, mentre Tim si occuperà dei comandamenti. Per farlo, lavoreranno su branch locali per non intasare il server e il log.

precetti

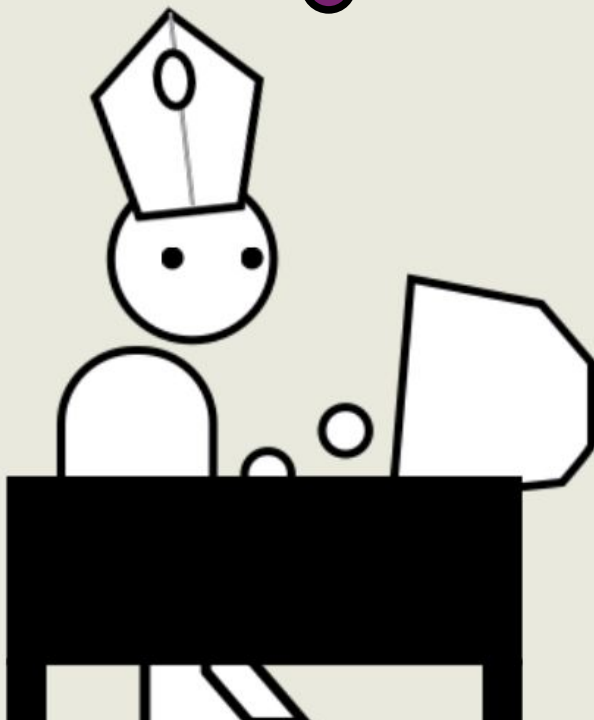
main

“Precetti” è ora il  
branch attivo. I  
comandi hanno effetto  
solo su questo branch

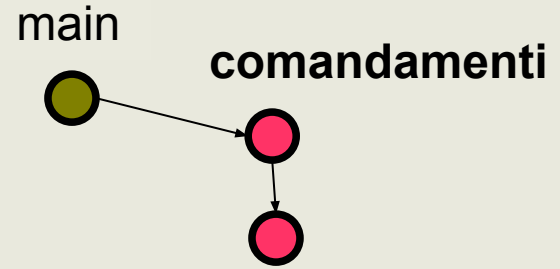


```
$ git checkout -b  
precetti
```

```
$ git commit -am  
"modifica"
```




```
$ git checkout -b  
comandamenti  
  
$ git add  
comandamenti.txt  
$ git commit -m "inizio"  
$ git branch  
main  
* comandamenti
```





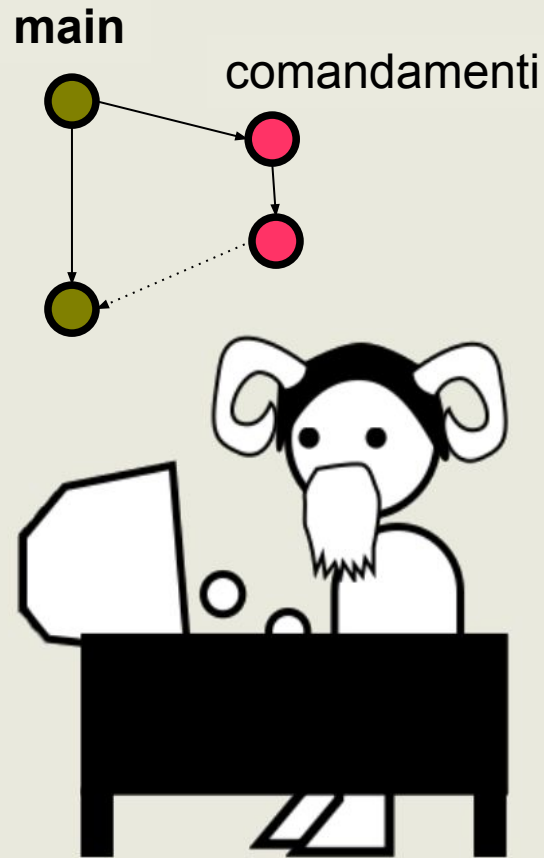
# Merge

Al termine di una porzione significativa del lavoro del branch, è opportuno ricongiungere i rami. L'operazione è detta Merge (fusione). Come nel caso del pull (che è di fatto un merge), si possono generare conflitti che vanno risolti.



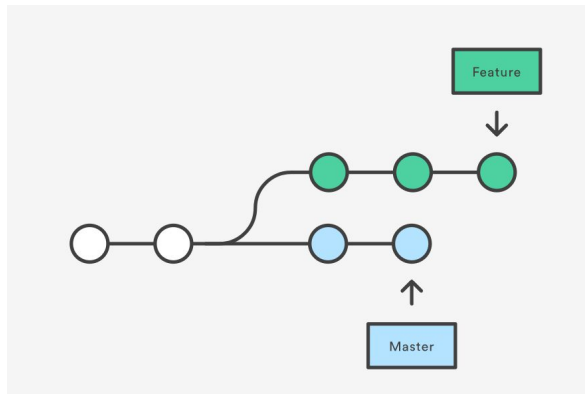
```
$ git checkout main
```

```
$ git merge comandamenti
```



## Fusione: come?

Per fondere i vari branch esistono due modalità base: merge e rebase. La differenza è come effettuano la fusione.



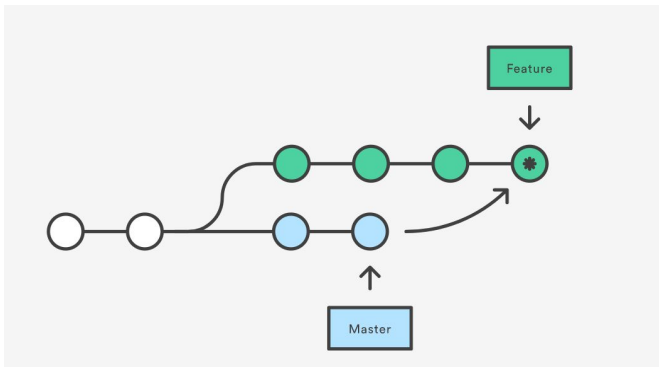
# Merge

git merge master feature

Il comando consente di posizionarsi sul ramo *feature* e di ricevere le commit presenti in *master*. Questo tipo di operazione **non è invasiva** sulla struttura pre-esistente dei rami.

In questo caso il risultato potrebbe essere di non facile interpretazione per gli altri sviluppatori del team. Questo perchè sarebbero necessari **numerosi merge** per poter allineare il branch.

•



# Rebase

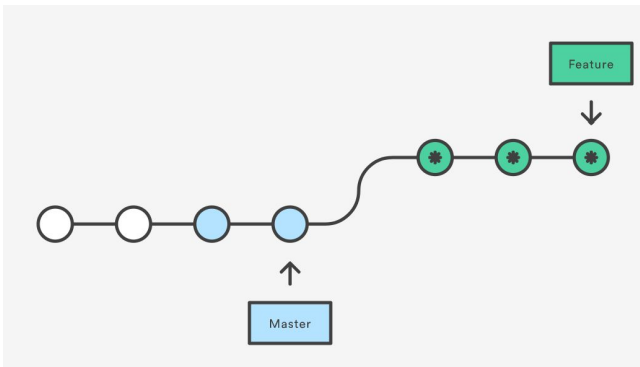
git checkout feature

git rebase master

Dopo essersi posizionato sul branch *feature*, il comando rebase accoda l'intero ramo *feature* a *master*, di fatto "riscrivendo la storia" dei commit. Il risultato è un un flusso di lavoro molto più lineare, eliminando tutta la cronologia di commit sviluppata nel ramo *feature*.

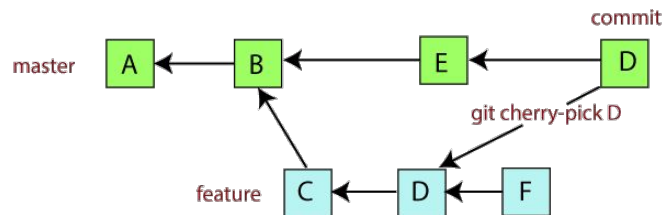
Potrebbero però verificarsi danni irreversibili, puoché si perde, lo storico delle modifiche che sono avvenute prima dell'operazione di rebase.

Morale: mai eseguire un rebase su rami pubblici.



## Altre opzioni

**git cherry pick** (lett: spiluccatura): Permette di “pescare” commit presi da qualsiasi branch e applicarlo al branch attuale.





## Rivedere la storia dei commit

Dato che i commit sono “cheap”, il log può diventare molto verboso specie in progetti grandi o longevi. Può quindi avere senso “comprimere” tanti commit in uno solo.

Si può fare in vari modi

```
git rebase -i HEAD~5
```

Apri una sessione interattiva che permette di comprimere gli ultimi cinque commit

```
git merge --squash feature/login
```

Combina i commit del branch in un unico commit (che va effettuato successivamente)

```
git commit - - amend
```

Modifica l'ultimo commit (ideale per correzioni)


# 4.

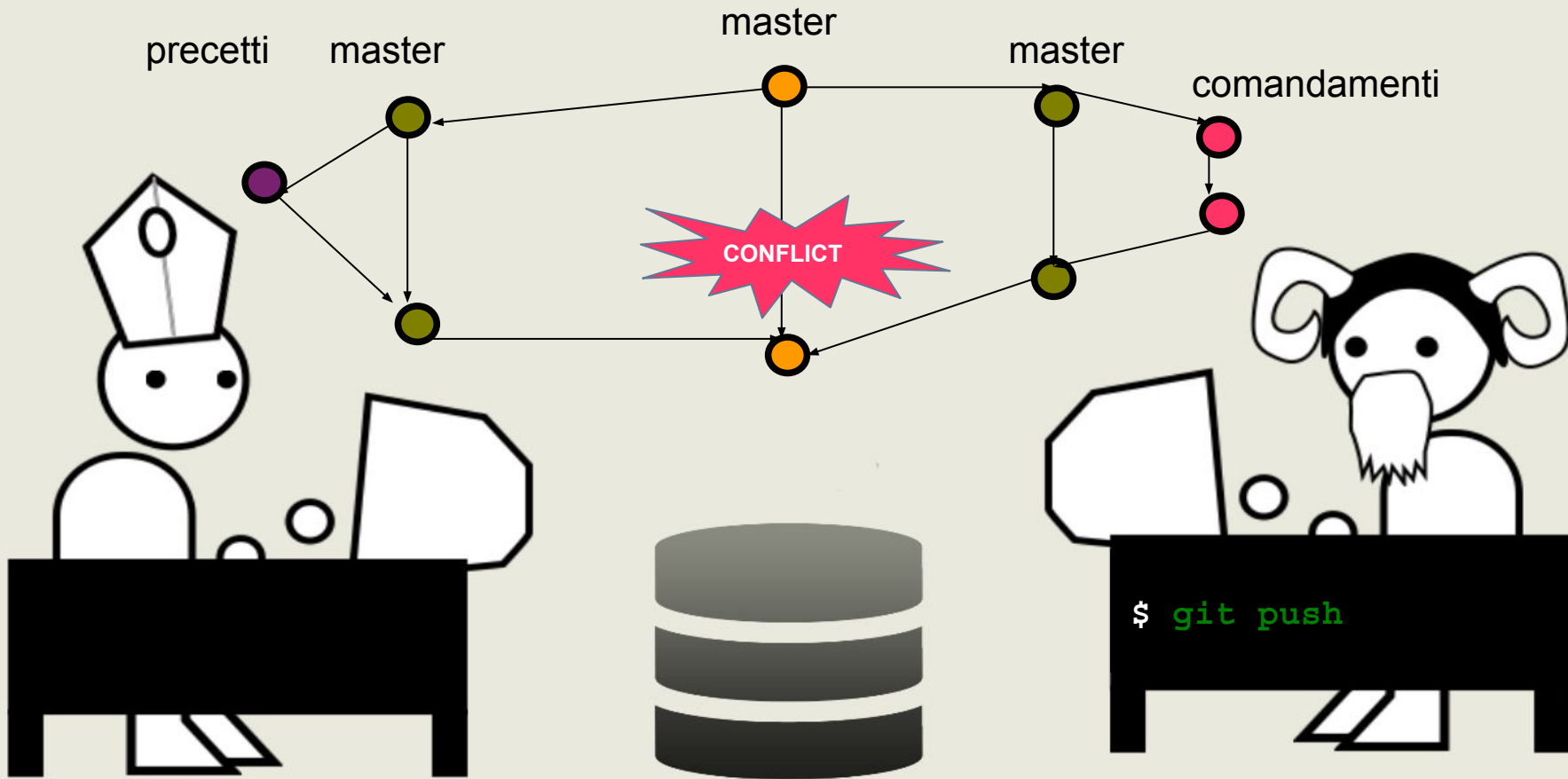
## Lavorare con i remoti

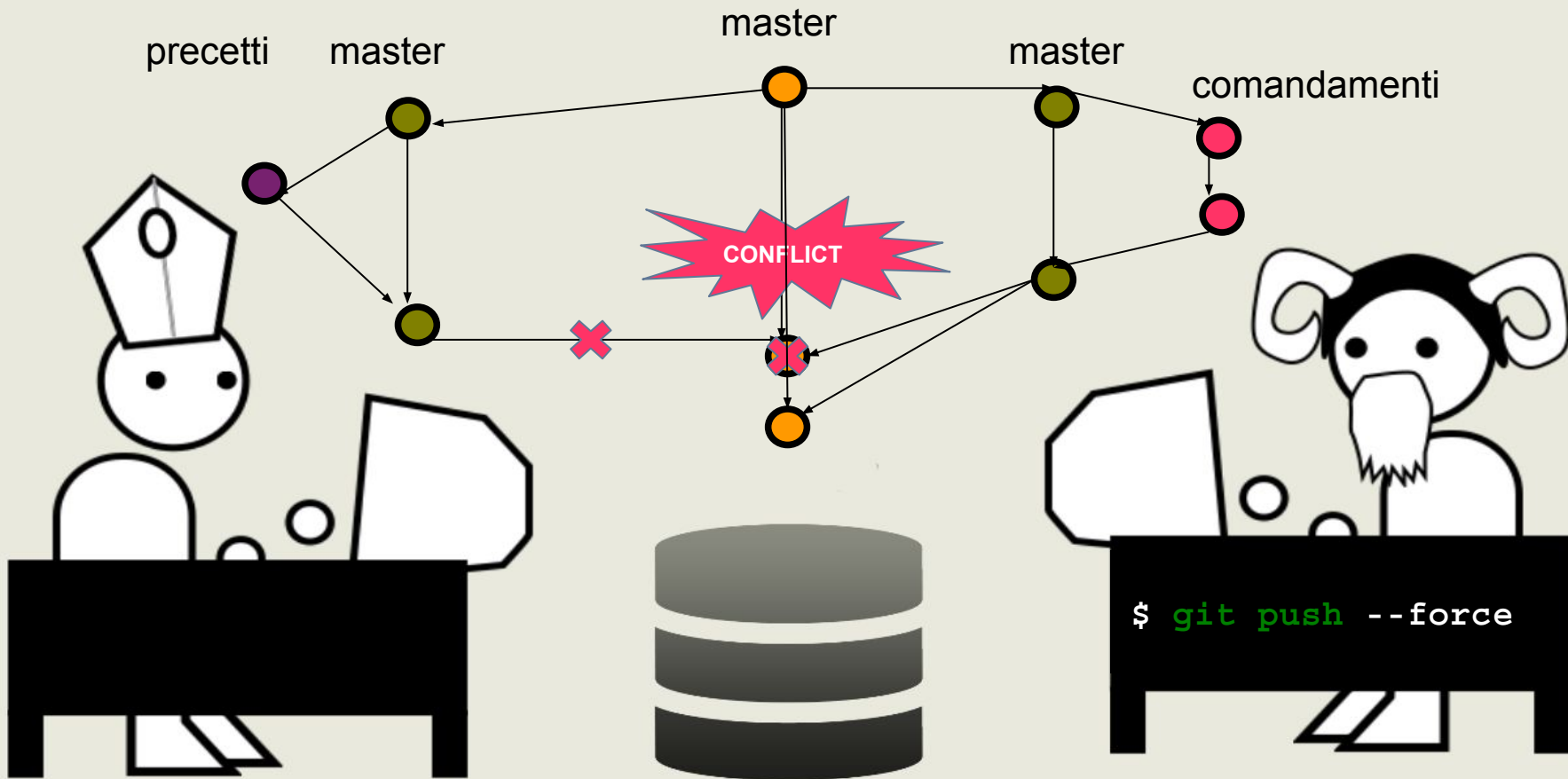


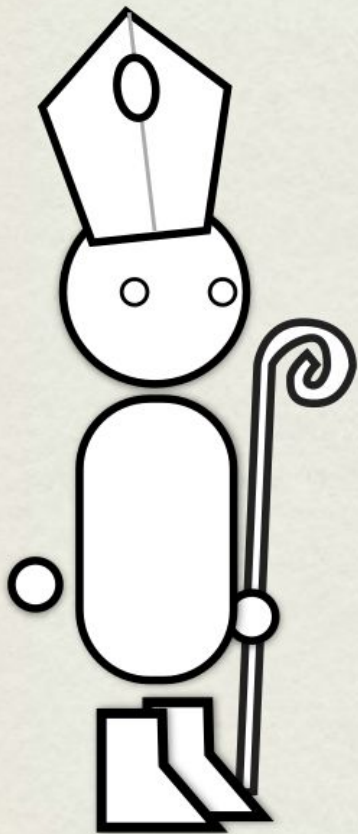
## Fusione su remoto

Dopo molto lavoro, è opportuno riunire tutto il lavoro nel repository remoto. Questo può essere fonte di conflitti (spesso non solo tecnici, ma anche umani)

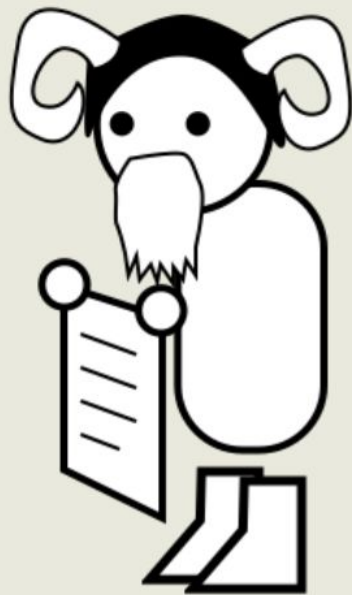
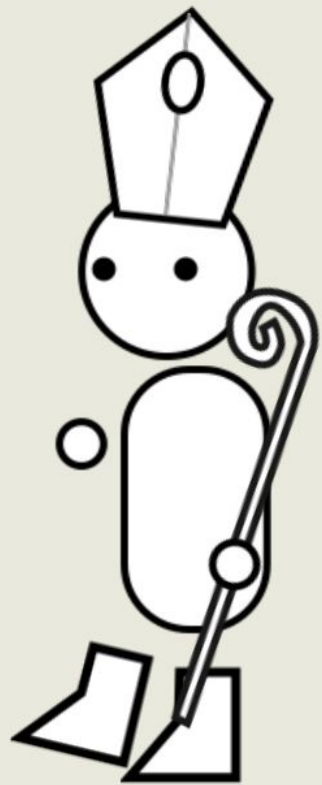








“Non lavarsi non sarà mai considerato peccato”



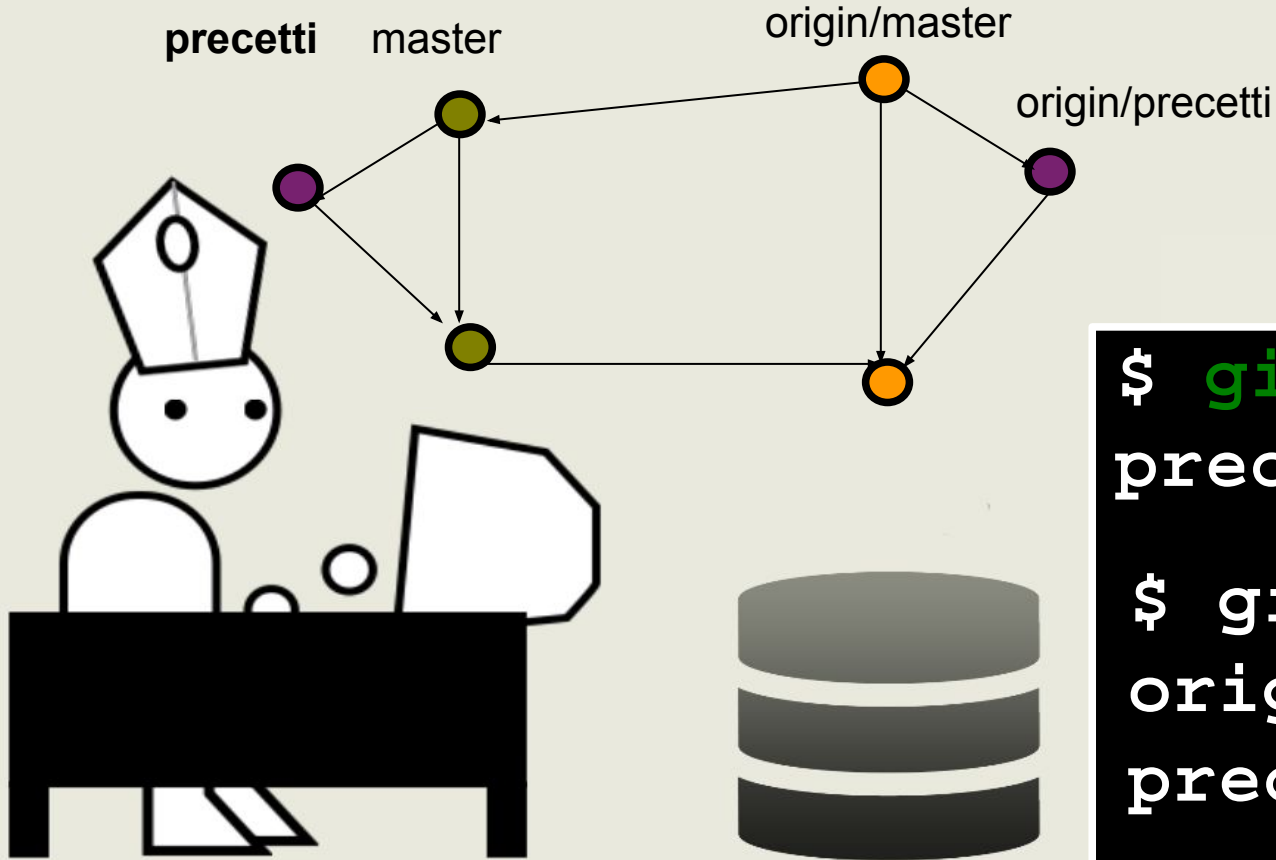


## Branch remoto

Tipicamente, il repo remoto segue il branch main (o master). Nulla però vieta di far tracciare altri branch.







```
$ git checkout  
precetti
```

```
$ git push -u  
origin  
precetti
```

# Trucchetti

- » **Cancellare** un branch locale : `git branch -d [nome]`
- » **Cancellare** un branch remoto: `git push origin --delete [nome]`
- » **Rinominare** un branch locale: `git branch -M [nome]`
- » **Git log** con tutti branch, “cool”: `git log --oneline --decorate --graph --all`



# 5.

## Flussi di lavoro

Vari modi di lavorare in gruppo



## 'N chesso?

Esistono tanti gruppi e altrettanti modi di lavorare. Git è particolarmente flessibile e si adatta agli stili di lavoro di chiunque (e non viceversa). Vediamone alcuni.



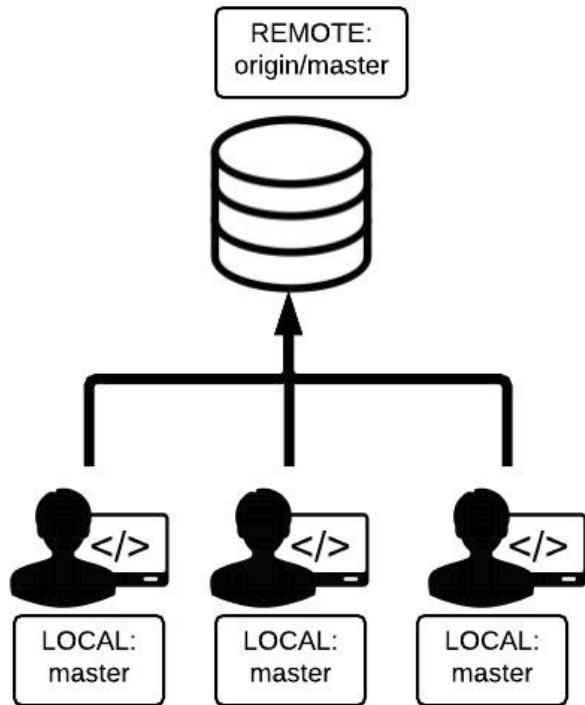
# Centralized Workflow

Il server remoto (su Github) è il server di riferimento (“custode della verità”).

Ogni sviluppatore clona il repository e sviluppa LOCALMENTE

Quando uno sviluppatore termina una sezione significativa, pulla e integra localmente. Quindi reinvia al repo remoto.

É il sistema visto finora.



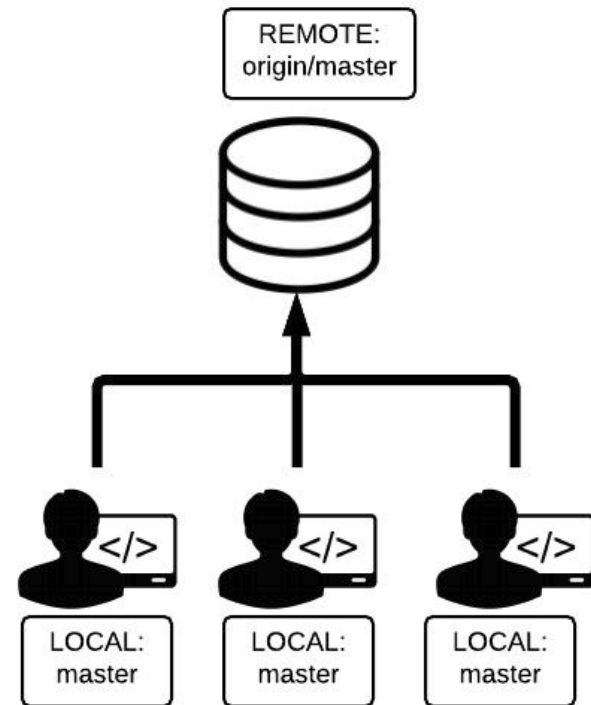
# Centralizzato

## Vantaggi

- » Facile da capire
- » Ottimo per team piccoli

## Svantaggi

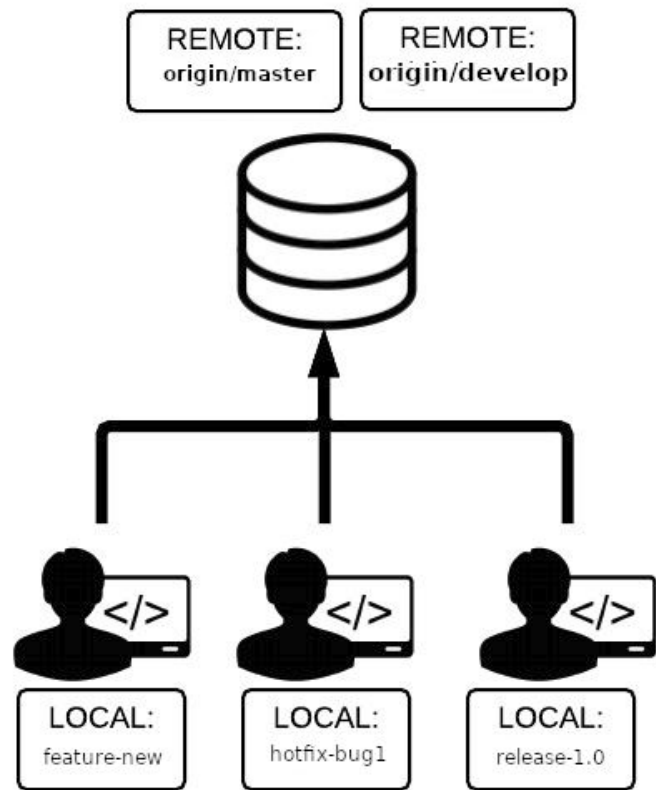
- » Alla lunga gestire i conflitti diventa problematico
- » Non sfrutta la natura distribuita di Git's



# Gitflow

È un flusso molto famoso proposto da Vincent Driessen nel 2010. Prevede due branch di durata infinita:

- » master/main – contiene il codice pronto (in gergo, “di produzione”).
- » develop – branch di sviluppo

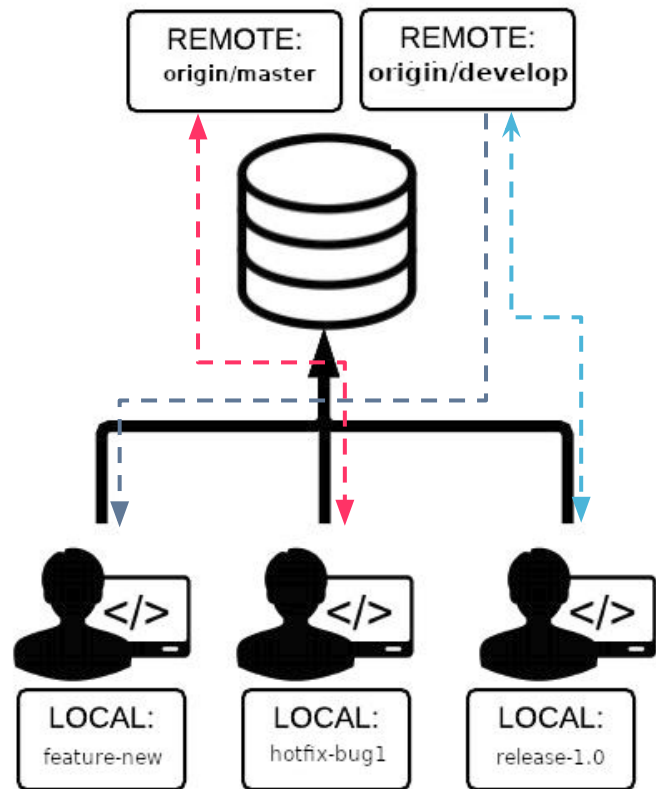


## Gitflow (2)

Ogni sviluppatore crea branch temporanei secondo una terminologia ben precisa:

- » feature-\* — per sviluppare nuove funzionalità. Figlia di develop.
- » hotfix-\* — per riparare ad errori del master. Da mergiare in master o develop
- » release-\* — in preparazione di un rilascio. Figlia di develop, da riunire a master e develop.

. [Ulteriori fettagli qui.](#)





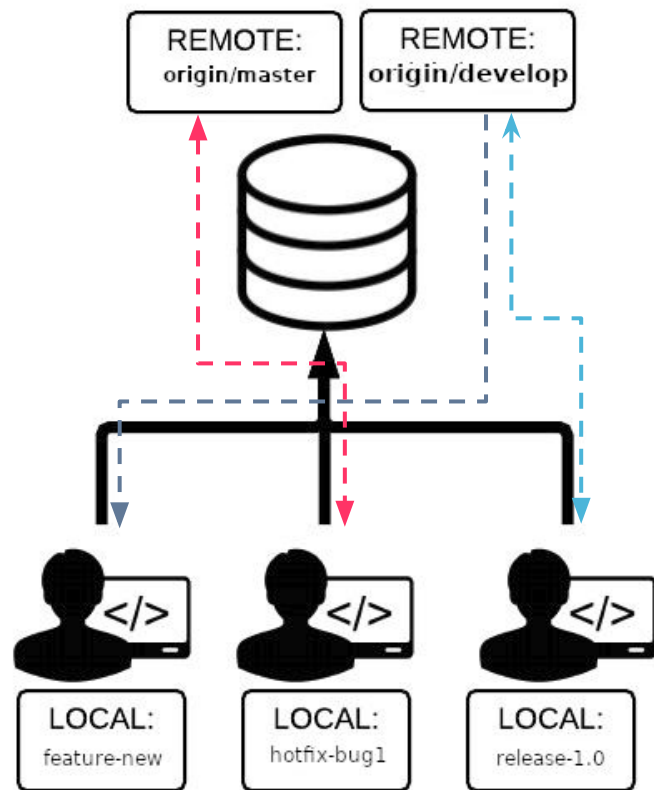
# Flusso Git

## Vantaggi

- » Ideale se occorrono rilasci regolari.
- » Presenza di tool
- » Master ha solo codice stabile..

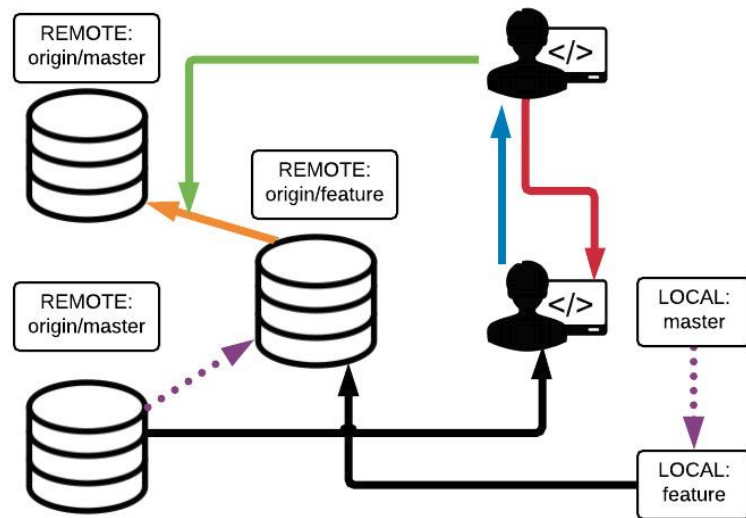
## Svantaggi

- » La storia passata diventa intricata e illeggibile
- » Non funziona bene con versioni multiple
- » Benvenuti a Merge Hell. Popolazione: 1 (tu).
- »



# Feature branch workflow

Il repo remoto è il server di riferimento. Ogni sviluppatore forka il repository, quindi clona localmente; se è fuori sincronia si ribasa alla situazione corrente. Uno sviluppatore crea un branch locale (replicato anche nel server remoto). Quando ha finito controlla chiedendo ai colleghi se il push si può fare. Quindi si procede al push



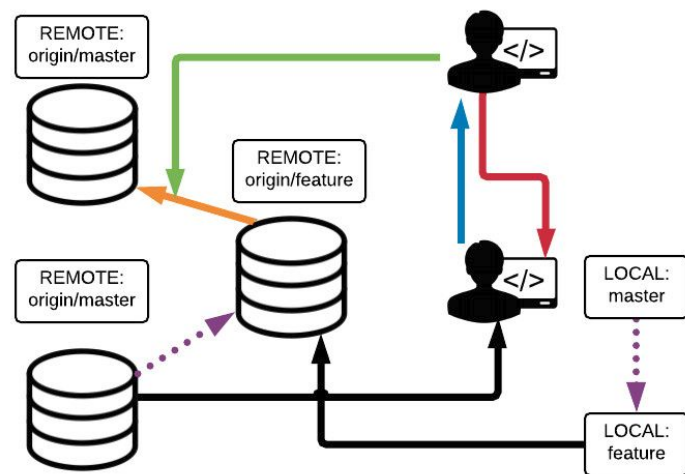
# Feature branch


## Vantaggi

- » Promuove la collaborazione e la proprietà condivisa.
- » Master resta stabile


## Svantaggi

- » Le branch più vecchie tendono ad avere problemi di merge.





# Forking & pull/merge request



## Che sarebbe?

Il forking è una caratteristica presente in tutti i siti di social coding che permette di **duplicare** un repository esistente, inserendolo tra i propri progetti (quindi con libertà totale di accesso) pur mantenendo un legame con il repository originario (nome in codice “upstream”)

## E quindi?

Questa relazione è alla base del cosiddetto sistema di **pull request** (Github, BitBucket)/**merge request** (Gitlab).

Dopo aver forkato un repository e aver realizzato delle modifiche, si invia una richiesta di inserire la modifica invece di farlo direttamente. Questi potrà accettare la modifica oppure rifiutarla, inviando magari un commento.

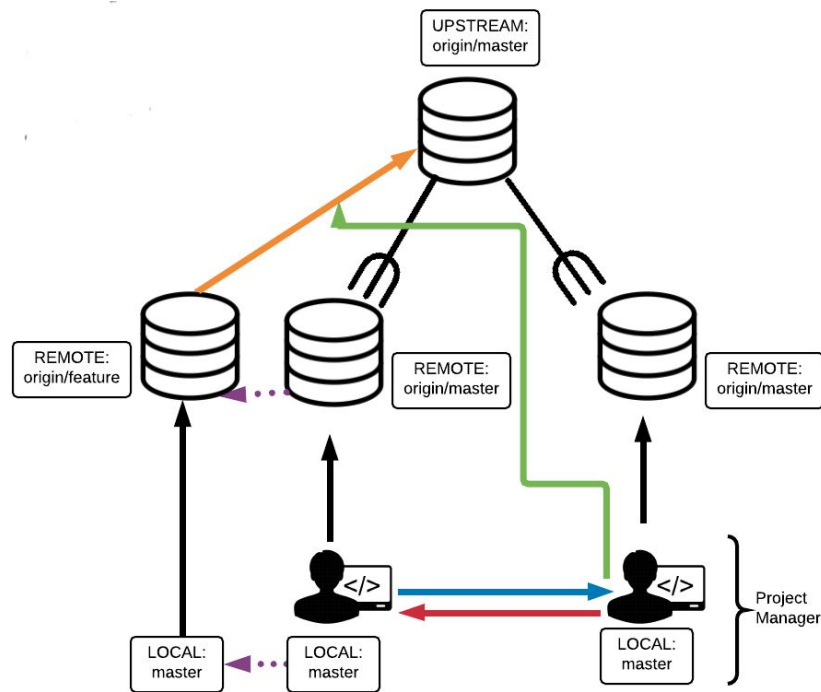
# Forking Workflow

Esiste un server  
(UPSTREAM) di riferimento

Ogni sviluppatore forka il  
server. Quindi clona  
localmente il suo repository

Lavora indisturbato. Quando  
si sente pronto, invia una  
pull request a un collega  
(Project master)

Se tutto va bene, il Project  
manager integra le modifiche



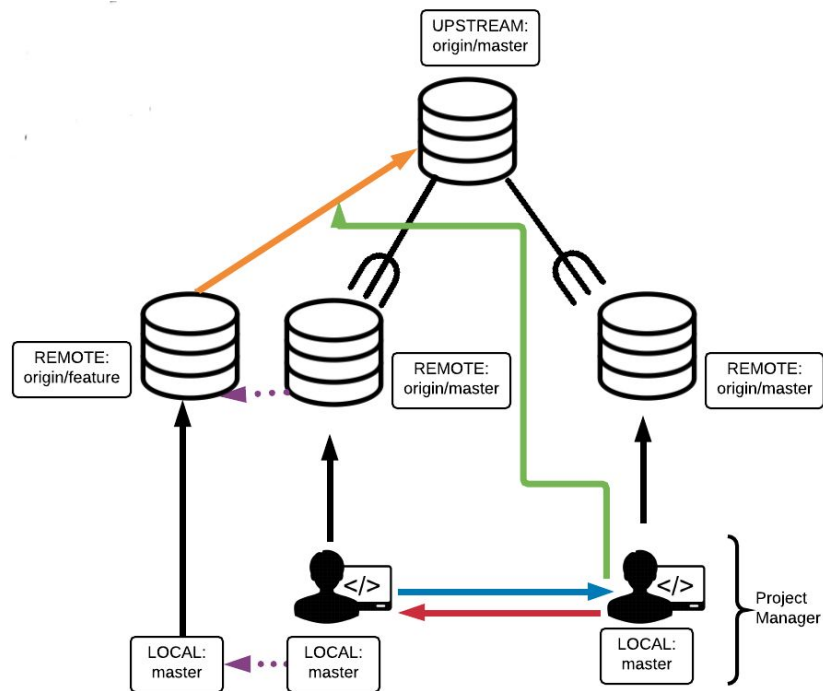
# Forking Workflow

## Vantaggi

- » Il repo upstream resta impeccabile, tramite un ferreo controllo degli accessi in scrittura.

## Svantaggi

- » Più complicata
- » Implica una struttura gerarchica





# Rivediamo qualche concetto

## Branch

Una diramazione della linea di sviluppo

## Checkout

Permette di passare ai diversi branch, nonché crearli e distruggerli

## Merge

Commit verso il server remoto

## Fork

Clonatura di un intero repo mantenendo un collegamento logico

## Upstream

Repository l"padre"

## Pull/merge request

Richiesta di aggiornamento proveniente dal repository forkato.



6.

**Ottenere riconoscimento**





# Produttività

Tutti dovrebbero essere interessati alla propria produttività. In aggiunta, in molti posti di lavoro la produttività documentata potrebbe essere un elemento per la valutazione del personale.

Per questo è utile tenere traccia di quello che si fa.



# Tracciamento del lavoro con git

Git è un efficiente memorizzatore del lavoro effettuato. Alcuni semplici comandi permettono di avere un'idea della produttività di un utente.

Un comando molto semplice è "git shortlog" che indica il numero di commit

```
git shortlog -sn --all
```

16 Prof. Missiroli (home)

3 Scott Tolinski

2 Scott

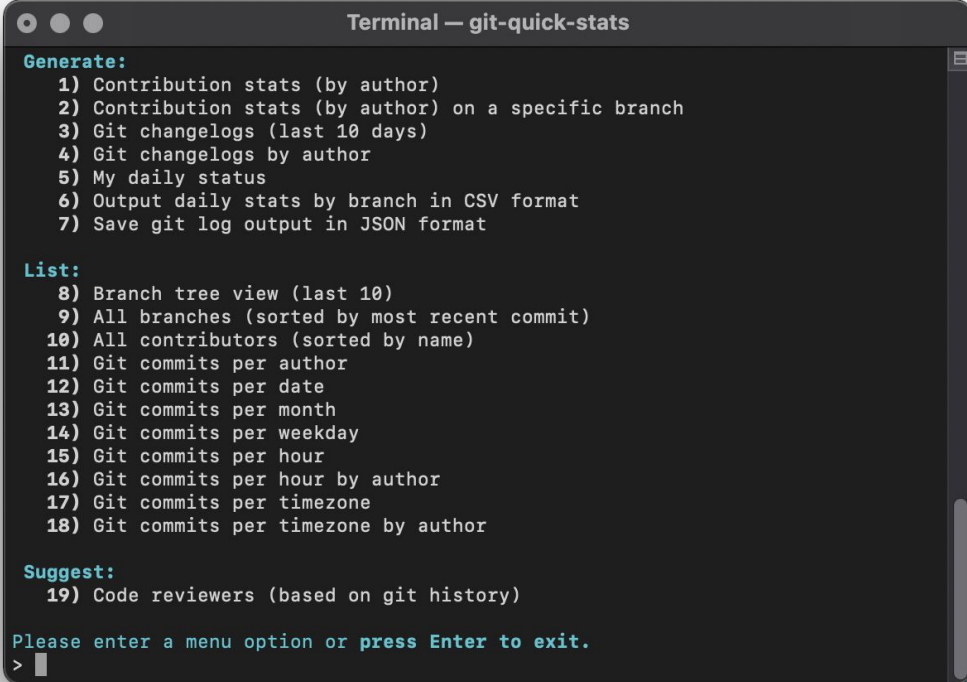
2 prof@blackbird

1 Jared Malcolm

1 Oleg Lytvyn

# Quick-stats

Il numero di commit non è mai una metrica troppo significativa. **Git quick-stats** è un programma facilmente installabile (reperibile qui: <https://github.com/arzzen/git-quick-stats>) che offre molte opzioni interessanti.

A terminal window titled "Terminal — git-quick-stats" with a dark background and light blue text. It displays a menu of 19 options categorized into "Generate:", "List:", and "Suggest:". The cursor is at the bottom, ready for input.

```
Terminal — git-quick-stats

Generate:
 1) Contribution stats (by author)
 2) Contribution stats (by author) on a specific branch
 3) Git changelogs (last 10 days)
 4) Git changelogs by author
 5) My daily status
 6) Output daily stats by branch in CSV format
 7) Save git log output in JSON format

List:
 8) Branch tree view (last 10)
 9) All branches (sorted by most recent commit)
10) All contributors (sorted by name)
11) Git commits per author
12) Git commits per date
13) Git commits per month
14) Git commits per weekday
15) Git commits per hour
16) Git commits per hour by author
17) Git commits per timezone
18) Git commits per timezone by author



Suggest:
19) Code reviewers (based on git history)

Please enter a menu option or press Enter to exit.
> |
```

# Gitinspector

Gitinspector è un programma per Linux che genera un ottimo grafico riassuntivo.



Le seguenti informazioni storiche sui commit, per autore, sono state trovate nel repository.

Autore v	Commit	Inserimenti	Rimozioni	% dei cambiamenti
 S1mone	11	163	12	53.19
 samuliga	5	149	5	46.81



 S1mone  
 samuliga





Di seguito il numero di righe da ogni autore che sono sopravvissute a sono ancora intatte nella versione corrente.

Autore v	Righe	Stabilità	Età	% in commenti
 S1mone	175	107.4	0.1	16.00
 samuliga	120	80.5	0.0	12.50



 S1mone  
 samuliga

La seguente timeline storica è stata ricavata dal repository.

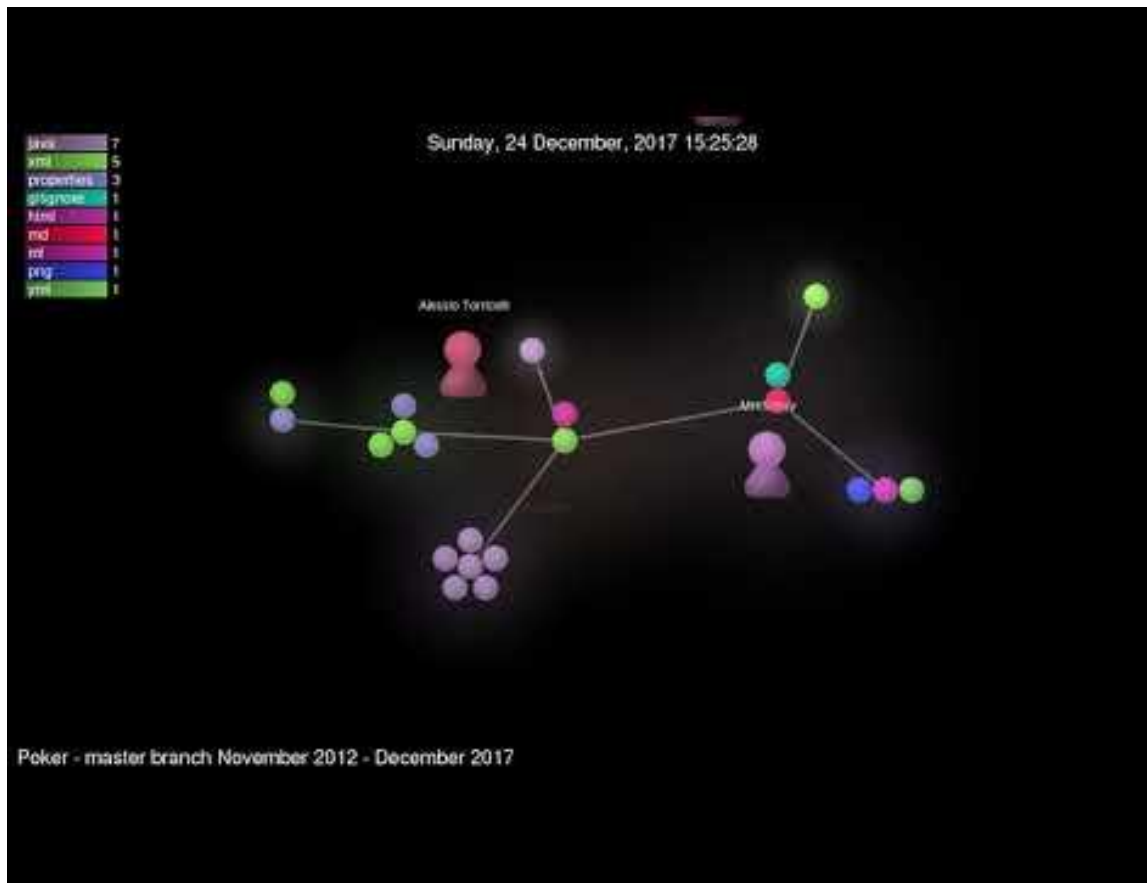
Autore	2021W12
 S1mone	
 samuliga	
<b>Righe modificate:</b>	<b>329</b>

Non sono state rilevate violazioni delle metriche nel repository.

# Gourcevid

Gitinspector è un programma per Linux che genera un'animazione dei commit. Really cool.

Attenti alle dimensioni, può raggiungere diversi Gb. Usare programmi come Handbrake per ridurli.



There are three kinds of lies:  
lies, damned lies, and  
statistics

-- Disraeli (1804-1881)





## “Git mente!”

E' ampiamente possibile che questi metodi non restituiscano una fotografia fedele del lavoro compiuto.

Spesso accade perché gli autori del lavoro non compiono il commit/push in prima persona, ma lo passano ad una unica persona che opera il commit.

Oppure se si inserisce una libreria corposa e/o codice trovato su internet.

Oppure se si lavora in pair/mob programming

## Soluzione 1 (manuale)

In generale, ogni autore dovrebbe commitare solo il codice che ha davvero redatto.

Per evitare malintesi, potete modificare temporaneamente il vostro username/password con git config e poi resettare i vostri dati

## Soluzione 2 (co-author)

Durante un commit, nel messaggio LASCIARE ALMENO UNA RIGA VUOTA nel commento poi aggiungere i tag necessari Rilevata da Github e Gitlab, ma non da gitinspector e git log (si può ottenere con un po' di modifiche).

```
$ git commit -m "Esempio  
di commit.
```

```
>
```

```
>
```

**Linee vuote**

```
Co-authored-by: piffy  
<piffy@gmail.com>
```

```
Co-authored-by:  
prof.missiroli  
<prof.missiroli@corni.it>  
"
```



## Soluzione 3 (git -pair)

Installare git pair,  
configurare gli utenti e  
usarlo in questo modo

```
git users jane@example.com  
bob@example.com  
gitp commit -m "Did the code"
```



# **GRAZIE!**

**...e abbiamo quasi  
concluso....**



# CREDITS

Ringraziamenti a :

- » Presentation template by SlidesCarnival
- » Photographs by Unsplash
- » Anil Gupta ([www.guptaanil.com](http://www.guptaanil.com))
- » Pete Nicholls ([github.com/Aupajo](https://github.com/Aupajo))
- » Armando Fox

***Questo documento è distribuito con  
licenza CreativeCommon BY-SA 3.0***