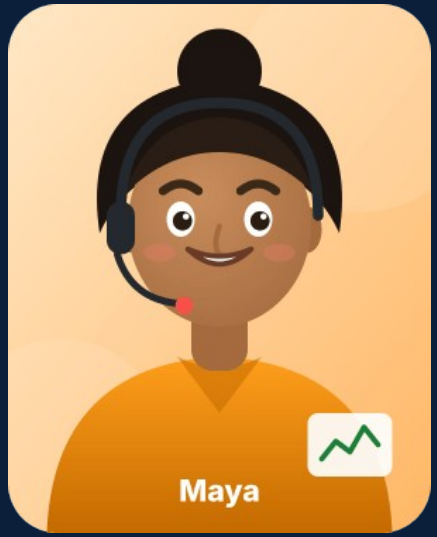
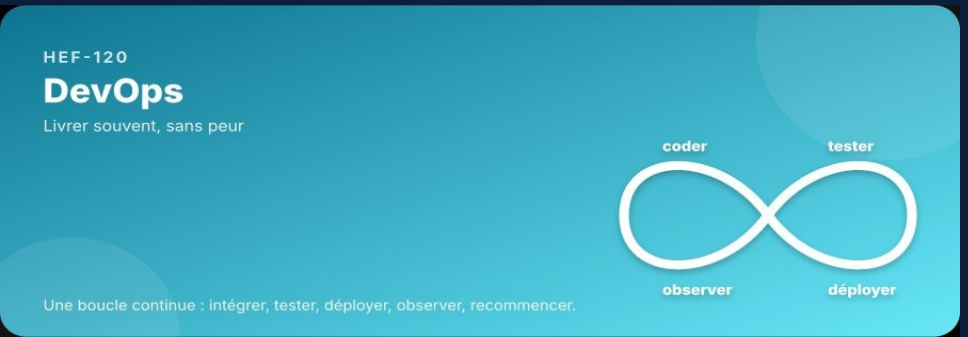
HEF-120 · MISSION 02 / 10

Git et stratégie de collaboration

Git n'est pas qu'un bouton « sauvegarder » : c'est un graphe de l'histoire du projet, et une manière de travailler à plusieurs sans se marcher dessus.



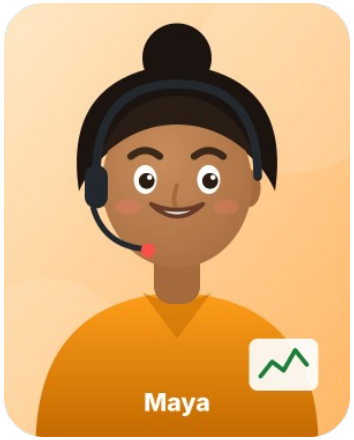


Badge : Maître de l'historique

100 min · intermédiaire

L'histoire

Deux formateurs de HashBox modifient le même fichier de tarifs, chacun de son côté. Le premier remplace le second sans le savoir ; trois jours plus tard, les anciens prix réapparaissent. Un autre jour, un bug est apparu « quelque part dans les 200 derniers commits » et personne ne sait lequel l'a introduit.



> *« Git résout les deux problèmes, dit Sam. Chaque version est un point d'un graphe ; on fusionne des branches avec détection des conflits ; et on peut chercher automatiquement le commit coupable. Il faut juste comprendre le modèle. »*

Tu apprends le modèle de Git (commits, branches, fusions), les stratégies de collaboration (trunk-based, GitFlow), les conventions de messages, et tu retrouves un bug automatiquement avec `git bisect`.

Ta mission

À la fin, je serai capable de…

1. expliquer le modèle de Git : commits, graphe orienté acyclique, branches comme pointeurs
2. fusionner et résoudre un conflit
3. comparer fusion (merge) et réécriture (rebase)
4. choisir une stratégie de branches (trunk-based, GitFlow) et protéger la branche principale
5. retrouver un commit fautif avec `git bisect`

Défi d'échauffement

Défi d'échauffement

Cinq personnes éditent le même rapport en s'envoyant des versions par e-mail : `rapport_v3_final_FINAL.docx`. Cite trois problèmes. Comment un système qui retient **chaque modification** avec son auteur, sa date et son explication résoudrait-il chacun d'eux ?

Le concept

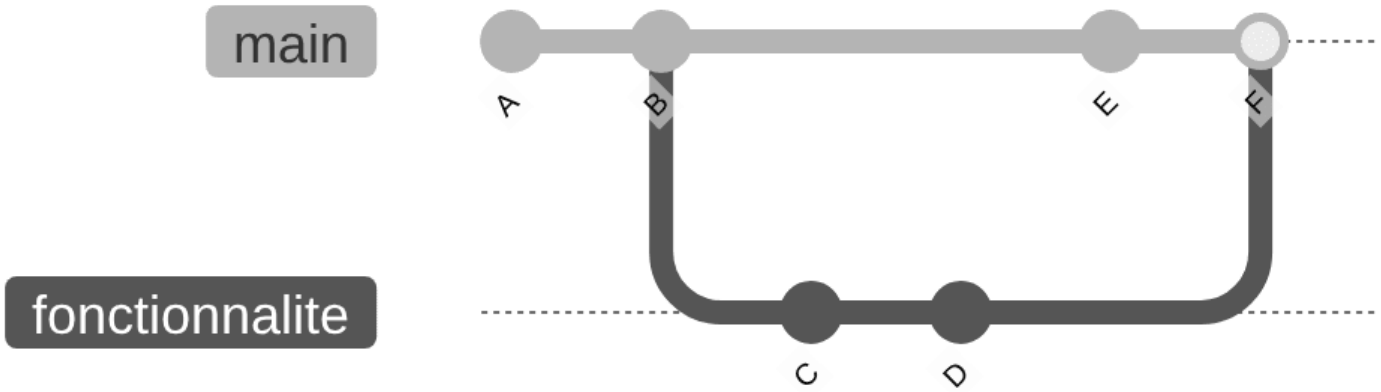
Le modèle de Git

Git conserve l'histoire sous forme de **commits** : des instantanés complets du projet, chacun avec :

- un **identifiant** (haché SHA du contenu et du parent) ;
- un **ou plusieurs parents** ;
- un auteur, une date, un message.

Les commits forment un **graphe orienté acyclique** (DAG). Une **branche** n'est qu'un **pointeur** mobile vers un commit ; HEAD indique où l'on se trouve.

Le modèle de Git (suite)



Description du schéma : la branche principale avance de A à B ; une branche « fonctionnalite » part de B et reçoit C et D ; pendant ce temps la branche principale reçoit E ; puis la fusion F réunit les deux histoires.

Le concept

Les commandes de tous les jours

Besoin	Commande
Voir l'état	`git status`, `git log --oneline --graph`
Enregistrer	`git add fichier` puis `git commit -m "message"`
Créer / changer de branche	`git switch -c nom` / `git switch nom`
Voir les différences	`git diff`
Récupérer / publier	`git fetch`, `git pull`, `git push`
Fusionner	`git merge nom`
Annuler **sans réécrire** l'histoire	`git revert <commit>`
Chercher un bug	`git bisect`

Le concept

Fusion (merge) et rebase

	Merge	Rebase
Idée	crée un commit de fusion à deux parents	rejoue tes commits au-dessus de la branche cible
Histoire	fidèle, avec bifurcations	linéaire
Risque	graphe plus touffu	**réécrit** l'histoire : jamais sur une branche déjà partagée

Règle d'or du rebase : **ne réécris jamais l'histoire publique** que d'autres ont déjà récupérée.

Les conflits

Un **conflit** survient quand deux branches modifient les **mêmes lignes** différemment. Git marque les zones :

Le concept

Les conflits (suite)

```
<<<<<<< HEAD
prix = 40
=======
prix = 45
>>>>>>> fonctionnalite
```

On **choisit** (ou combine), on supprime les marqueurs, on enregistre. Les conflits sont moins douloureux avec de **petites branches** de **courte durée**.

Stratégies de collaboration

Stratégie	Principe	Quand
Trunk-based	tout le monde intègre souvent (au moins chaque jour) dans la branche principale ; branches très courtes ; fonctionnalités cachées derrière des *drapeaux*	équipes qui livrent en continu (recommandé par DORA)

Le concept

Stratégies de collaboration (suite)

Stratégie	Principe	Quand
GitFlow	branches `develop`, `release`, `hotfix`, `feature`	livraisons planifiées, plusieurs versions maintenues
GitHub Flow	une branche principale toujours déployable + branches courtes + demande de fusion	applications web

Demandes de fusion (*pull requests*)

Une **PR** propose de fusionner une branche. C'est le lieu de la **revue** et de l'**automatisation** (CI). Bonnes pratiques : **petites** PR (moins de 400 lignes), description claire, tests verts, revue rapide (quelques heures).

Le concept

Branche principale protégée

Règles côté dépôt : fusion **uniquement via PR**, **CI verte**, au moins **une revue**, pas de *push* direct, pas de réécriture d'historique. C'est l'une des **portes de qualité** de la mission 1 du module.

Messages de commit et versionnement

Les **commits conventionnels** structurent l'historique : `type(portée): résumé` avec `feat`, `fix`, `docs`, `refactor`, `test`, `chore`, et `!` pour une rupture.

Le **versionnement sémantique** (`MAJEUR.MINEUR.CORRECTIF`) en découle :

Changement	Incrémente
`fix`	CORRECTIF

Le concept

Messages de commit et versionnement (suite)

Changement	Incrémente
`feat`	MINEUR
rupture de compatibilité (`!`)	MAJEUR

`git bisect` : la recherche dichotomique d'un bug

Tu indiques un commit **mauvais** et un commit **bon** ; Git teste le milieu, puis réduit l'intervalle de moitié à chaque étape : 200 commits en environ 8 essais. Avec `git bisect run <script>`, tout est **automatique**.

> 🧠 ***Astuce** : commits **petits, atomiques**, avec un message qui explique le **pourquoi**. git bisect et git revert deviennent alors des outils magiques.*

Exemple pas à pas

Problème : utiliser **réellement** Git dans un dépôt temporaire : historique, branches, conflit résolu, puis `git bisect run` pour retrouver automatiquement le commit qui a cassé une fonction.

```
import os, subprocess, tempfile
from pathlib import Path

ENV = dict(os.environ, GIT_AUTHOR_NAME="Kodi", GIT_AUTHOR_EMAIL="kodi@exemple.fr",
           GIT_COMMITTER_NAME="Kodi", GIT_COMMITTER_EMAIL="kodi@exemple.fr", GIT_C
   ONFIG_GLOBAL="/dev/null", GIT_CONFIG_SYSTEM="/dev/null")

def git(depot, *args, ok=True):
    r = subprocess.run(["git", "-c", "init.defaultBranch=main", *args], cwd=depot,
    env=ENV, capture_output=True, text=True)
    if ok and r.returncode != 0:
        raise RuntimeError(f"git {' '.join(args)} : {r.stderr.strip()}")
    return r

def commit(depot, nom, contenu, message):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    Path(depot, nom).write_text(contenu)
    git(depot, "add", nom)
    git(depot, "commit", "-q", "-m", message)
    return git(depot, "rev-parse", "HEAD").stdout.strip()

with tempfile.TemporaryDirectory() as d:
    git(d, "init", "-q")

    # ---- 1. un historique linéaire avec un bug glissé au 6e commit ----
    commits = []
    for i in range(1, 11):
        # le « prix » vaut 40 jusqu'au commit 5 ; le commit 6 le casse (45), le re
   ste ne le corrige pas
        prix = 40 if i < 6 else 45
        commits.append(commit(d, "tarifs.py", f"PRIX = {prix}\nVERSION = {i}\n", f
   "feat: étape {i}"))
    print("historique :", len(commits), "commits")

    # ---- 2. git bisect run : on cherche automatiquement le premier mauvais commi
```

Exemple pas à pas (suite)

```
t ----
 test = Path(d, "..", "verifier.py").resolve()
 Path(test).write_text("import sys, re\nsrc = open('tarifs.py').read()\nsys.exi
t(0 if 'PRIX = 40' in src else 1)\n")
 git(d, "bisect", "start")
 git(d, "bisect", "bad", commits[-1])
 git(d, "bisect", "good", commits[0])
 resultat = git(d, "bisect", "run", "python3", str(test))
 fautif = next(l for l in resultat.stdout.splitlines() if "is the first bad com
mit" in l).split()[0]
 git(d, "bisect", "reset")
 print("commit fautif trouvé :", fautif[:8], "=", git(d, "log", "-1", "--format
=%s", fautif).stdout.strip())
 assert fautif == commits[5]                                    # le 6e co
mmit
 nb_etapes = resultat.stdout.count("running")
 print("étapes de recherche :", nb_etapes, "pour", len(commits), "commits")
 assert nb_etapes <= 4
```

Exemple pas à pas (suite)

```
    # ---- 3. une branche, une fusion, et un conflit ----
    git(d, "switch", "-q", "-c", "remise")
    commit(d, "tarifs.py", "PRIX = 42\nVERSION = 10\nREMISE = 0.1\n", "feat: prix
promotionnel et remise")
    git(d, "switch", "-q", "main")
    commit(d, "tarifs.py", "PRIX = 50\nVERSION = 10\n", "fix: nouveau prix de base
")
    fusion = git(d, "merge", "remise", ok=False)
    assert fusion.returncode != 0 and "CONFLICT" in fusion.stdout
    print("conflit détecté :", [l for l in fusion.stdout.splitlines() if "CONFLICT
" in l][0])
    contenu = Path(d, "tarifs.py").read_text()
    assert "<<<<<<<" in contenu and "=======" in contenu and ">>>>>>>" in contenu

    # résolution : on garde le nouveau prix ET la remise
    Path(d, "tarifs.py").write_text("PRIX = 50\nVERSION = 10\nREMISE = 0.1\n")
    git(d, "add", "tarifs.py")
    git(d, "commit", "-q", "-m", "merge: intègre la remise avec le nouveau prix")
    parents = git(d, "rev-list", "--parents", "-n", "1", "HEAD").stdout.split()
```

Exemple pas à pas (suite)

```
    print("commit de fusion à", len(parents) - 1, "parents")
    assert len(parents) == 3

    # ---- 4. revert : annuler sans réécrire l'histoire ----
    avant = git(d, "rev-list", "--count", "HEAD").stdout.strip()
    git(d, "revert", "--no-edit", "HEAD", "-m", "1")
    apres = git(d, "rev-list", "--count", "HEAD").stdout.strip()
    print(f"revert : l'historique passe de {avant} à {apres} commits (rien n'est e
ffacé)")
    assert int(apres) == int(avant) + 1
```

Lecture :

1. **Historique** : 10 commits simples (`git commit` enregistre un instantané complet).
2. **`git bisect run`** : en ~3-4 essais (recherche dichotomique, comme dans la mission HEF-010), Git retrouve le 6e commit, **sans intervention humaine**, grâce à un script de test dont le code de sortie dit « bon » (0) ou « mauvais » (1).

Exemple pas à pas (suite)

3. **Conflit** : deux branches modifient la même ligne ; Git s'arrête, marque les zones, **ne devine pas**. On résout, on `add`, on `commit` : le commit de fusion a **deux parents**.
4. **`revert`** : crée un **nouveau** commit qui annule l'effet de l'ancien. L'histoire publique reste intacte — c'est la bonne façon d'annuler un changement déjà partagé.

À toi, avec aide

Écris `analyser_commits(messages)` qui, à partir d'une liste de messages de commits conventionnels, calcule la **prochaine version** (à partir de `1.4.2`) et un **journal des changements** regroupé par type :

- `feat: …` → mineur ; `fix: …` → correctif ; `feat!: …` ou `BREAKING CHANGE` dans le message → majeur ;
- les autres types (`docs`, `test`, `chore`, `refactor`) n'incrémentent rien ;
- le plus fort incrément l'emporte (majeur > mineur > correctif).

Teste avec trois listes de commits.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Utilise une expression régulière `^(\w+)(\([^)]*\))?(!)?: (.+)$`.

Indice 2

Initialise le niveau à `None` et garde le maximum selon un ordre `correctif < mineur < majeur`.

Indice 3

Si aucun commit n'incrémente, la version reste inchangée.

À toi, avec aide · réponse

Réponse

```
import re
from collections import defaultdict

MOTIF = re.compile(r"^(\w+)(\([^)]*\))?(!)?: (.+)$")
NIVEAU = {"fix": 1, "feat": 2}

def analyser_commits(messages, version="1.4.2"):
    journal, niveau = defaultdict(list), 0
    for m in messages:
        ligne = m.splitlines()[0]
        trouve = MOTIF.match(ligne)
        if not trouve:
            continue
        type_, _, rupture, resume = trouve.groups()
        journal[type_].append(resume)
        courant = NIVEAU.get(type_, 0)
        if rupture or "BREAKING CHANGE" in m:
            courant = 3
        niveau = max(niveau, courant)
```

Réponse (suite)

```
    maj, mi, co = map(int, version.split("."))
    if niveau == 3: maj, mi, co = maj + 1, 0, 0
    elif niveau == 2: mi, co = mi + 1, 0
    elif niveau == 1: co += 1
    return f"{maj}.{mi}.{co}", dict(journal)

assert analyser_commits(["fix: corrige l'arrondi", "docs: précise le README"])[0]
   == "1.4.3"
assert analyser_commits(["fix: x", "feat(api): ajoute la recherche"])[0] == "1.5.0
   "
assert analyser_commits(["feat!: change le format des dates", "fix: y"])[0] == "2.
   0.0"
assert analyser_commits(["chore: bump", "test: ajoute des cas"])[0] == "1.4.2"
assert analyser_commits(["feat: z\n\nBREAKING CHANGE: retire l'ancienne route"])[0
   ] == "2.0.0"
print(analyser_commits(["feat(api): ajoute la recherche", "fix: corrige l'arrondi"
   ]))
```

Réponse (suite)

Automatiser ce calcul dans la CI fournit le numéro de version et les notes de publication sans intervention manuelle.

À toi, en solo

Recréer un petit Git. Écris `MiniDepot` qui conserve un graphe de commits :

1. `commit(message, parents)` crée un identifiant par hachage SHA-1 du contenu (message + parents + compteur) et renvoie l'identifiant ;
2. les branches sont de simples pointeurs `{nom: id}` ;
3. `anciens(id)` renvoie l'ensemble de tous les ancêtres ;
4. `base_commune(a, b)` renvoie l'ancêtre commun **le plus récent** (base de fusion) ;
5. `est_ancetre(a, b)` : est-ce qu'une fusion `b` dans `a` serait un simple « fast-forward » ?
6. compare ton résultat à `git merge-base` sur un vrai dépôt temporaire construit avec le même graphe.

À toi, en solo · réponse

Pistes de correction

```
import hashlib, os, subprocess, tempfile
from pathlib import Path

class MiniDepot:
    def __init__(self):
        self.commits, self.branches, self.n = {}, {}, 0
    def commit(self, message, parents=()):
        self.n += 1
        ident = hashlib.sha1(f"{message}|{','.join(parents)}|{self.n}".encode()).h
   exdigest()[:10]
        self.commits[ident] = {"message": message, "parents": tuple(parents)}
        return ident
    def anciens(self, ident):
        vus, pile = set(), [ident]
        while pile:
            c = pile.pop()
            if c in vus:
                continue
            vus.add(c)
```

Pistes de correction (suite)

```
            pile.extend(self.commits[c]["parents"])
        return vus
    def base_commune(self, a, b):
        communs = self.anciens(a) & self.anciens(b)
        # le plus récent = un commun qui n'est ancêtre d'aucun autre commun
        feuilles = [c for c in communs if not any(c != o and c in self.anciens(o)
   for o in communs)]
        return feuilles[0] if len(feuilles) == 1 else sorted(feuilles)
    def est_ancetre(self, a, b):
        return a in self.anciens(b)

dep = MiniDepot()
A = dep.commit("A"); B = dep.commit("B", (A,))
C = dep.commit("C", (B,)); D = dep.commit("D", (C,))      # branche 1
E = dep.commit("E", (B,))                                 # branche 2
F = dep.commit("F", (D, E))                               # fusion
assert dep.base_commune(D, E) == B and dep.base_commune(D, C) == C
assert dep.est_ancetre(B, F) and not dep.est_ancetre(D, E)
assert dep.base_commune(F, E) == E
```

À toi, en solo · réponse

Pistes de correction (suite)

```
# comparaison avec le vrai git
ENV = dict(os.environ, GIT_AUTHOR_NAME="K", GIT_AUTHOR_EMAIL="k@x.fr", GIT_COMMITT
   ER_NAME="K", GIT_COMMITTER_EMAIL="k@x.fr", GIT_CONFIG_GLOBAL="/dev/null", GIT_C
   ONFIG_SYSTEM="/dev/null")
def git(d, *a):
    return subprocess.run(["git", "-c", "init.defaultBranch=main", *a], cwd=d, env
   =ENV, capture_output=True, text=True, check=True).stdout.strip()
with tempfile.TemporaryDirectory() as d:
    git(d, "init", "-q")
    for nom in "AB":
        Path(d, "f").write_text(nom); git(d, "add", "f"); git(d, "commit", "-q", "
   -m", nom)
    b = git(d, "rev-parse", "HEAD")
    git(d, "switch", "-q", "-c", "b1")
    for nom in "CD":
        Path(d, "f").write_text(nom); git(d, "add", "f"); git(d, "commit", "-q", "
   -m", nom)
    git(d, "switch", "-q", "-c", "b2", b)
```

Pistes de correction (suite)

```
    Path(d, "g").write_text("E"); git(d, "add", "g"); git(d, "commit", "-q", "-m",
    "E")
    assert git(d, "merge-base", "b1", "b2") == b
print("MiniDepot et git merge-base s'accordent")
```

Un commit est un nœud avec des parents ; une fusion en a deux ; la **base commune** est l'ancêtre commun le plus récent, point de départ de toute fusion à trois voies.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Gros commits fourre-tout	Revue difficile, bisect inutile	Commits atomiques avec message explicatif
Réécrire l'histoire déjà partagée	Casse le travail des collègues	Revert pour annuler ; rebase seulement en local
Branches qui vivent des semaines	Conflits énormes, intégration douloureuse	Branches courtes, intégration fréquente
Pousser directement sur la branche principale	Pas de revue ni de CI	Branche protégée, PR obligatoire

Mini-boss

Atelier Git complet, scripté. Écris un script qui, dans un dépôt temporaire, met en scène le travail de trois personnes :

1. historique initial de 8 commits conventionnels ;
2. deux branches en parallèle qui modifient des fichiers différents (fusion sans conflit) et une troisième qui modifie la même ligne qu'une autre (conflit résolu par script avec une règle claire) ;
3. un commit fautif introduit dans une fonction ; retrouve-le avec `git bisect run` et un test automatique ;
4. annule-le avec `git revert` (et non `reset`) ;
5. produis la liste des versions (`v1.0.0`, `v1.1.0`, …) avec des tags à partir des types de commits ;
6. vérifie par code : l'historique de la branche principale ne contient aucun commit réécrit (les SHA publiés avant le revert sont tous encore présents).

Mini-boss · réponse

Pistes de correction

Un bon travail : un `git()` utilitaire comme dans l'exemple, des commits générés à partir d'une liste de messages conventionnels, la résolution du conflit par réécriture du fichier puis `add` et `commit`, un script `bisect` basé sur le code de sortie d'un test, `git revert --no-edit <sha>` (ou `-m 1` pour une fusion), `git tag -a vX.Y.Z -m …` calculé avec `analyser_commits`, et une vérification `git merge-base --is-ancestor <ancien_sha> HEAD` pour chaque SHA publié avant le revert.

Quiz éclair

Quiz éclair

1. Une branche Git est :

A. une copie complète du projet
B. un fichier de configuration
C. un pointeur mobile vers un commit

2. Pour annuler un changement déjà partagé, on utilise :

A. git revert, qui crée un commit d'annulation
B. git rebase sur la branche publique
C. supprimer le dépôt

3. Le trunk-based development repose sur :

A. des branches de longue durée

Quiz éclair

Quiz éclair (suite)

B. des intégrations très fréquentes dans la branche principale
C. un seul développeur

4. 🔁 *Révision mission 01* : DevOps cherche surtout à :

A. remplacer les développeurs par des administrateurs
B. supprimer les tests
C. livrer vite et de façon fiable grâce à une responsabilité partagée

5. 🔁 *Révision mission 01* : Pour réduire le délai moyen à débit constant, on :

A. réduit le travail en cours et la taille des lots
B. augmente la taille des lots
C. retarde les déploiements

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

1. Quelle est la durée de vie typique de tes branches ? Que se passerait-il si elles duraient deux fois moins longtemps ?
2. Quel commit de ton historique récent expliquerait mal le pourquoi, six mois plus tard ?

À retenir

À retenir

1. Git = graphe de commits ; une branche est un pointeur.
2. Merge fidèle, rebase linéaire (jamais sur l'histoire publique).
3. Conflit = mêmes lignes modifiées ; on résout, on enregistre.
4. Trunk-based : petites branches, intégration quotidienne ; branche principale protégée.
5. `git bisect run` retrouve un bug par dichotomie ; `git revert` annule sans réécrire.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Maître de l'historique

Prochaine mission : Build reproductible

