Git et GitLab

Introduction à la gestion de versions et au travail collaboratif

Nicolas Delestre
INSA Rouen Normandie
Département ISIA

Table des matières

1 Pourquoi gérer les versions ? 2
- 1.1 Livrables et travail collectif . . . 2
  - 1.1.1 Logiciels, bibliothèques et documents . . . 2
  - 1.1.2 Répartition des tâches et traçabilité . . . 3
- 1.2 Fonctions d'un système de gestion de versions . . . 3
  - 1.2.1 Conserver et consulter un historique . . . 3
  - 1.2.2 Travailler en parallèle et réunir les contributions . . . 3
- 1.3 Repères historiques et architectures . . . 4
  - 1.3.1 Quelques repères historiques . . . 4
  - 1.3.2 Systèmes centralisés et décentralisés . . . 4
- 1.4 Git et GitLab : deux rôles complémentaires . . . 5
  - 1.4.1 Git, le logiciel de gestion de versions . . . 5
  - 1.4.2 GitLab, une forge pour le projet collectif . . . 5

2 Comprendre le modèle de Git 6
- 2.1 Dépôt et historique local . . . 6
  - 2.1.1 Un historique disponible sur la machine de l'auteur . . . 6
  - 2.1.2 Des instantanés de l'ensemble du projet . . . 7
- 2.2 Commit et identification d'une version . . . 7
  - 2.2.1 Une étape accompagnée de son contexte . . . 7
  - 2.2.2 Identifiants et intégrité des données . . . 8
- 2.3 Les trois espaces de travail . . . 8
  - 2.3.1 Le répertoire de travail . . . 8
  - 2.3.2 L'index, préparation du prochain instantané . . . 8
  - 2.3.3 Le dépôt local et la circulation des contenus . . . 9
- 2.4 États des fichiers et transitions . . . 9
  - 2.4.1 Fichiers non suivis et fichiers suivis . . . 9
  - 2.4.2 Contenus non modifiés, modifiés et indexés . . . 10
  - 2.4.3 Un fichier peut être indexé puis modifié à nouveau . . . 10

3 Enregistrer et examiner un travail local 12
- 3.1 Environnement et aide . . . 12
  - 3.1.1 Une commande, des actions et des options . . . 12
  - 3.1.2 L'identité associée aux commits . . . 13
- 3.2 Initialisation et observation du dépôt . . . 13
  - 3.2.1 Créer le dépôt avec `git init` . . . 13
  - 3.2.2 Lire l'état du projet avec `git status` . . . 13
- 3.3 Sélection et enregistrement des contenus . . . 14
  - 3.3.1 Préparer le contenu avec `git add` . . . 14
  - 3.3.2 Valider une étape avec `git commit` . . . 14
- 3.4 Sources et fichiers générés . . . 15

3.4.1 Choisir ce qui appartient au projet versionné . . . 15
3.4.2 Décrire les exclusions dans `.gitignore` . . . 15
3.5 Suppression et arrêt du suivi . . . 16
3.5.1 Préparer une suppression avec `git rm` . . . 16
3.5.2 Conserver le fichier avec `git rm --cached` . . . 16
3.6 Lecture des différences et de l'historique . . . 17
3.6.1 Comparer le travail courant et l'index . . . 17
3.6.2 Vérifier ce qui sera enregistré . . . 17
3.6.3 Retrouver les étapes avec `git log` . . . 18

4 Échanger avec un dépôt distant **19**
4.1 Dépôts locaux et dépôt partagé . . . 19
4.1.1 Le rôle du serveur Git . . . 19
4.1.2 Un nom pour le distant et une branche suivie . . . 19
4.2 Clonage et accès au dépôt . . . 20
4.2.1 Obtenir une copie avec `git clone` . . . 20
4.2.2 SSH et HTTPS . . . 20
4.3 Recevoir les changements avec `git pull` . . . 21
4.3.1 Récupérer puis intégrer les commits . . . 21
4.3.2 Une mise à jour sans divergence . . . 21
4.4 Publier les commits avec `git push` . . . 22
4.4.1 Transmettre les versions enregistrées . . . 22
4.4.2 Comprendre un refus de publication . . . 22
4.5 Une collaboration sans conflit . . . 22
4.5.1 Deux copies du même rapport . . . 22
4.5.2 Une première contribution publiée par Camille . . . 23
4.5.3 Réception puis contribution de Samir . . . 23

5 Comprendre et résoudre un conflit **25**
5.1 Contributions concurrentes et divergence . . . 25
5.1.1 Deux commits issus d'une même version . . . 25
5.1.2 Une fusion automatique ou un conflit . . . 26
5.2 Résoudre un conflit de fusion . . . 26
5.2.1 Choisir la fusion lors d'une divergence . . . 26
5.2.2 Repérer l'interruption de l'intégration . . . 27
5.2.3 Lire les marqueurs de conflit . . . 27
5.2.4 Construire le texte final . . . 28
5.2.5 Vérifier le document . . . 28
5.2.6 Indexer la résolution et finaliser la fusion . . . 28
5.2.7 Publier puis recevoir la version résolue . . . 29
5.3 Coordonner les contributions . . . 29

6 GitLab et la gestion de projet **31**
6.1 Les services d'une forge . . . 31
6.1.1 Hébergement, utilisateurs et permissions . . . 31
6.1.2 Documentation partagée et wiki . . . 31
6.1.3 Intégration et déploiement continus . . . 32
6.2 Décrire et suivre une tâche avec un ticket . . . 32
6.2.1 Le travail attendu et son résultat . . . 32
6.2.2 Attribution, échanges et clôture . . . 33
6.3 Relier les tickets et les commits . . . 33
6.3.1 Une référence dans le message de commit . . . 33

6.3.2 Une convention pour la traçabilité collective 33
6.4 Du ticket au document partagé 34
6.4.1 Réalisation et vérification de la correction 34
6.4.2 Les responsabilités de chaque outil 34

Conclusion **36**

Avant-propos

Ce polycopié a été élaboré à partir de mon support de cours.

La rédaction de ce document a été réalisée avec l'aide de *Codex (GPT-6.1 Sol Moyen)*, un modèle d'intelligence artificielle générative développé par OpenAI. Il a été utilisé comme outil de rédaction initiale à partir de mes supports de cours. Les textes produits ont été relus, discutés et modifiés au cours de la rédaction. Le contenu pédagogique, les choix de notions abordées, leur organisation générale et les supports ayant servi de sources restent issus de mes enseignements.

Cette mention a pour objectif de rendre explicite l'utilisation d'un système d'intelligence artificielle générative dans la production du document et de distinguer cet usage des sources pédagogiques et scientifiques sur lesquelles le contenu s'appuie.

Chapitre 1

Pourquoi gérer les versions ?

Un livrable évolue au cours de sa réalisation : un programme reçoit une correction, une bibliothèque acquiert une nouvelle fonction, un document est relu puis complété. Lorsque plusieurs personnes participent au même projet, ces évolutions doivent pouvoir être conservées, comprises et réunies. La gestion de versions répond à ce besoin en organisant l'historique des modifications et leur partage.

Ce chapitre présente les problèmes auxquels répondent les outils de gestion de versions, puis situe Git et GitLab dans le travail d'une équipe.

> Le fil conducteur du cours est la rédaction collective d'un petit rapport en LaTeX. Plusieurs auteurs en rédigent les différentes parties, relisent le texte et corrigent les erreurs avant de produire une version commune. La rédaction de ce rapport permettra d'illustrer les besoins de gestion de versions à chaque étape : conserver les états du rapport, réunir les contributions et suivre les corrections. Bien que le livrable soit ici un document, sa réalisation soulève les mêmes questions de collaboration et de traçabilité que celle d'un logiciel.
> Dans tout le polycopié, les passages présentés sur un fond gris clair et accompagnés d'un trait vertical se rapportent à ce fil conducteur.

1.1 Livrables et travail collectif

1.1.1 Logiciels, bibliothèques et documents

Dans les spécialités d'ingénieurs du département ISIA, le travail de conception et de réalisation conduit à produire des livrables variés. Un livrable est un résultat identifiable du projet : il peut s'agir d'un logiciel, d'une bibliothèque réutilisable ou d'un document décrivant une étude. Sa production comporte généralement plusieurs étapes, depuis une première ébauche jusqu'à une version répondant aux objectifs fixés.

> Pour notre rapport en LaTeX, le texte est décrit dans un ou plusieurs fichiers sources, auxquels peuvent s'ajouter des illustrations. La compilation de ces sources produit notamment un fichier PDF destiné à la lecture. Une correction du texte source conduit donc à un nouveau résultat compilé. Comprendre l'évolution du document suppose de pouvoir retrouver les sources correspondant aux différentes étapes de sa rédaction.
> Pour conserver les étapes de ce rapport, les auteurs pourraient multiplier les copies portant des noms tels que `rapport-v2.tex` ou `rapport-final.tex`. Cette pratique offre un premier repérage, mais devient vite ambiguë. Le nom d'une copie ne décrit pas précisément les changements qu'elle contient ni les raisons de ces changements. Lorsque les copies circulent entre plusieurs auteurs, leur ordre et leur contenu deviennent également difficiles à établir.

1.1.2 Répartition des tâches et traçabilité

Une équipe répartit les tâches pour permettre à ses membres de contribuer au même livrable.

> Pour notre rapport, une personne peut rédiger l'introduction pendant qu'une autre présente les résultats. Cette répartition n'empêche pas des interventions sur des parties communes : la conclusion peut être enrichie par plusieurs auteurs, et une relecture peut concerner l'ensemble du texte.

La collaboration demande alors de distinguer deux questions. La première porte sur le contenu : quelles modifications ont été apportées, par qui et pour quelle raison ? La seconde porte sur l'organisation : quelles tâches restent à réaliser, à qui sont-elles confiées et dans quel délai ? La *traçabilité* désigne ici la possibilité de retrouver les contributions enregistrées et de comprendre leur place dans l'évolution du livrable. Elle repose à la fois sur un historique des modifications et sur des explications permettant de les interpréter.

> Lors de la relecture de notre rapport, une phrase peut présenter une valeur erronée. L'équipe doit pouvoir identifier la modification qui a introduit cette valeur, comprendre son contexte et enregistrer sa correction. Le suivi des tâches permet, de son côté, de savoir que cette correction est attendue puis qu'elle a été réalisée.

Ces besoins sont complémentaires : ils motivent l'utilisation conjointe d'un logiciel de gestion de versions et d'une forge.

1.2 Fonctions d'un système de gestion de versions

1.2.1 Conserver et consulter un historique

Un *système de gestion de versions* est un logiciel qui enregistre des états successifs d'un ensemble de fichiers et permet de consulter leur évolution. L'ensemble des données gérées pour conserver cet historique est appelé un *dépôt*. L'historique constitue une suite d'étapes identifiables, auxquelles sont associées des informations telles que l'auteur déclaré et un message expliquant la modification. L'enregistrement d'une étape est couramment appelé une *validation*, ou *commit* en anglais.

L'enregistrement est explicite : chaque sauvegarde effectuée dans l'éditeur ne devient pas automatiquement une nouvelle étape de l'historique. Les auteurs choisissent les changements qui constituent une étape cohérente et lui associent une explication.

> Pour notre rapport, la correction d'une valeur peut ainsi former une étape distincte de l'ajout d'une nouvelle section. Cette séparation facilite la compréhension des changements et de leurs conséquences.

L'historique permet de comparer des versions, de retrouver un contenu antérieur et de comprendre à quel moment une modification est apparue.

> Appliqué à notre rapport, il permet notamment de retrouver un paragraphe utile qui aurait disparu pendant une révision : son contenu reste consultable dans les versions où il était enregistré. L'équipe dispose ainsi d'un moyen de raisonner sur l'évolution du document, au-delà de son seul état courant.

1.2.2 Travailler en parallèle et réunir les contributions

Le *travail concurrent* désigne la possibilité, pour plusieurs personnes, de modifier leurs copies du projet pendant une même période. Chaque contributeur peut avancer sur sa tâche sans attendre que tous les autres aient terminé. Les modifications produites séparément doivent

ensuite être réunies dans une version commune. Cette opération d'intégration est appelée une *fusion*, ou *merge* en anglais.

> Lors de la rédaction de notre rapport, l'outil peut souvent combiner automatiquement les contributions de deux auteurs qui modifient des passages distincts. Lorsque leurs modifications portent sur un même passage et ne peuvent pas être combinées automatiquement, il signale un *conflit*. Les auteurs déterminent alors le contenu à retenir. Une fusion automatique ne garantit pas non plus la cohérence du sens. Dans notre rapport, une modification des résultats peut ainsi nécessiter une révision de la conclusion, même si les deux parties occupent des fichiers différents et que l'outil ne signale aucun conflit.

La gestion de versions permet également de conserver des lignes d'évolution parallèles, appelées *branches*. Une équipe peut ainsi préparer une évolution séparément avant de l'intégrer à une autre ligne de travail. La manipulation des branches et les stratégies qui organisent leur utilisation seront approfondies dans la suite de votre formation. Dans ce cours, l'étude de la collaboration porte sur le partage des contributions et sur la compréhension d'un conflit simple.

1.3 Repères historiques et architectures

1.3.1 Quelques repères historiques

Les systèmes de gestion de versions se développent dès les années 1970, notamment avec SCCS, puis avec RCS dans les années 1980. Ils répondent au besoin de conserver les révisions des fichiers et de retrouver leurs états antérieurs. Des outils tels que CVS puis Subversion organisent le travail autour d'un dépôt central partagé par les membres d'une équipe. Git et Mercurial représentent une autre approche, fondée sur des dépôts distribués.

Git est créé en 2005 à l'initiative de Linus Torvalds pour répondre aux besoins du développement du noyau Linux. Ce contexte explique l'importance accordée au travail distribué, à la rapidité des opérations et à la gestion de nombreuses contributions. Une présentation de cette origine est disponible dans *Pro Git*, de Scott Chacon et Ben Straub [1]. Pour comprendre la place de Git, la distinction essentielle porte sur l'emplacement du dépôt et sur les opérations qui dépendent d'un serveur.

1.3.2 Systèmes centralisés et décentralisés

Dans un système *centralisé*, un serveur héberge le dépôt de référence et son historique. Chaque contributeur travaille sur une copie des fichiers et échange avec ce serveur pour consulter les versions qui ne sont pas disponibles localement ou pour enregistrer sa contribution dans le dépôt central. CVS et Subversion relèvent de cette architecture. Une personne peut modifier ses fichiers sans connexion, mais l'enregistrement de ces modifications dans l'historique central nécessite l'accès au serveur.

Dans un système *décentralisé*, également appelé *distribué*, un contributeur dispose d'un dépôt local lui permettant de conserver un historique sur sa propre machine. Avec un clonage complet usuel de Git, l'historique du projet est disponible localement. La consultation de cet historique et la création de nouveaux commits peuvent donc s'effectuer sans connexion à un serveur. Les échanges avec d'autres dépôts interviennent ensuite pour partager les contributions.

Un projet Git peut néanmoins s'organiser autour d'un dépôt partagé hébergé sur un serveur. L'équipe choisit alors ce dépôt comme point de rencontre pour publier et recevoir les contributions. Ce choix d'organisation ne retire pas aux autres dépôts leur capacité à enregistrer un historique local. Le serveur facilite les échanges ; il n'est pas nécessaire à chaque enregistrement de version.

1. Scott Chacon et Ben Straub, *Pro Git*, deuxième édition, « Une rapide histoire de Git », édition en ligne.

> Pour la rédaction de notre rapport en LaTeX, l'utilisation d'un système décentralisé permet à deux auteurs de rédiger et d'enregistrer chacun plusieurs étapes du rapport sur leur machine. Une connexion est ensuite nécessaire pour transmettre ces étapes au serveur ou pour recevoir celles qui y ont été publiées. Cette séparation entre travail local et échanges distants est fondamentale pour comprendre les commandes de Git.

1.4 Git et GitLab : deux rôles complémentaires

1.4.1 Git, le logiciel de gestion de versions

Git prend en charge l'enregistrement des versions, la consultation de l'historique et les échanges entre dépôts. Il fonctionne sur la machine de chaque contributeur et peut être utilisé depuis un terminal. Un projet peut être versionné avec Git sur une seule machine, avant même qu'un serveur soit choisi pour le partager.

> Pour notre rapport, Git permet notamment de conserver une première version du texte, d'enregistrer une correction et de comparer les étapes de rédaction. Lorsqu'un dépôt partagé est disponible, il permet également de transmettre les commits et d'intégrer ceux des autres auteurs. Les opérations locales d'enregistrement et les opérations d'échange remplissent donc des fonctions distinctes.

1.4.2 GitLab, une forge pour le projet collectif

Une *forge* est une plateforme qui rassemble des services destinés au travail collectif sur un projet. GitLab est une forge qui propose notamment l'hébergement de dépôts Git, la gestion des utilisateurs et de leurs permissions, le suivi des tâches par des tickets et la documentation partagée. GitHub, Bitbucket et SourceForge sont d'autres exemples de forges.

Un *ticket* décrit un travail à effectuer ou un problème à traiter.

> Pour notre rapport, un ticket peut signaler la valeur erronée évoquée à la section 1.1 et permettre de suivre sa correction. L'historique Git conserve la modification du texte, tandis que le ticket permet de retrouver la demande et les échanges associés. Une référence au ticket dans le message du commit relie ces deux informations.

Les forges peuvent également proposer des mécanismes d'intégration et de déploiement continus, qui automatisent des vérifications et la mise à disposition de résultats. Ces possibilités seront abordées dans la suite de votre formation. Leur configuration n'est pas nécessaire pour comprendre la gestion de versions présentée ici.

Git et GitLab interviennent ainsi à des niveaux différents du même travail. Git gère l'historique et la circulation des contributions entre dépôts ; GitLab fournit un espace partagé pour héberger le dépôt et organiser le projet. La compréhension du dépôt local, des versions enregistrées et des changements préparés constitue la prochaine étape avant l'étude des commandes.

Chapitre 2

Comprendre le modèle de Git

Git organise le travail autour de versions enregistrées localement et d'un choix explicite des changements à conserver. Comprendre ce modèle permet de distinguer trois actions qui peuvent sembler proches : modifier un fichier, préparer son contenu pour un enregistrement et créer une nouvelle étape de l'historique. Ces actions n'ont ni le même rôle ni les mêmes effets.

> Lors de la rédaction de notre rapport en LaTeX, un auteur peut corriger une valeur du rapport tout en préparant une nouvelle section. Il peut souhaiter enregistrer la correction séparément, sans inclure le texte encore inachevé. Les notions d'instantané, de commit et d'index expliquent comment Git rend ce choix possible.

Les commandes seront étudiées dans le chapitre consacré à l'utilisation locale ; il s'agit ici de comprendre les mécanismes auxquels elles correspondent.

Les figures de ce chapitre sont reproduites sans modification à partir de *Pro Git*, de Scott Chacon et Ben Straub [1].

2.1 Dépôt et historique local

2.1.1 Un historique disponible sur la machine de l'auteur

Git est un système décentralisé, conçu initialement pour le développement du noyau Linux, comme indiqué à la section 1.3.1. Chaque contributeur peut disposer d'un dépôt local et y enregistrer ses propres étapes de travail. Dans une organisation usuelle, le répertoire du projet contient un répertoire caché nommé `.git`, qui conserve notamment les données nécessaires à l'historique. Les fichiers visibles du projet et cet historique local jouent donc des rôles distincts.

> Pour notre rapport, l'auteur travaille sur le fichier `main.tex`. Une sauvegarde dans l'éditeur met à jour ce fichier, mais n'ajoute pas à elle seule une version à l'historique Git. Une validation explicite est nécessaire pour conserver une nouvelle étape. Cette étape est enregistrée dans le dépôt local, indépendamment de toute publication sur un serveur.

La consultation de l'historique local et la création de commits ne nécessitent pas de connexion à un serveur Git. Un dépôt distant intervient pour échanger les versions entre contributeurs. Il ne faut donc pas confondre une modification conservée sur la machine de son auteur et une modification déjà accessible à l'équipe.

1. Scott Chacon et Ben Straub, *Pro Git*, deuxième édition, « Rudiments de Git ». Illustrations sous licence Creative Commons Attribution – Pas d'utilisation commerciale – Partage dans les mêmes conditions 3.0 (CC BY-NC-SA 3.0).

2.1.2 Des instantanés de l'ensemble du projet

Un *instantané*, ou *snapshot*, décrit l'état de l'ensemble des fichiers versionnés à une étape donnée. Git permet de retrouver cet état sans demander au lecteur de reconstituer lui-même une succession de corrections. Le modèle porte sur le projet : un commit peut enregistrer une modification touchant plusieurs fichiers, tout en conservant les autres fichiers dans leur état précédent.

La figure 2.1 représente cinq versions successives d'un projet contenant trois fichiers, A, B et C. Dans la deuxième version, les contenus de A et C ont changé, tandis que celui de B est conservé. Dans la troisième, seul C change à nouveau. Les éléments en pointillé signalent la réutilisation d'un contenu inchangé : l'état du projet reste complet, même lorsque certains fichiers n'ont pas été modifiés.

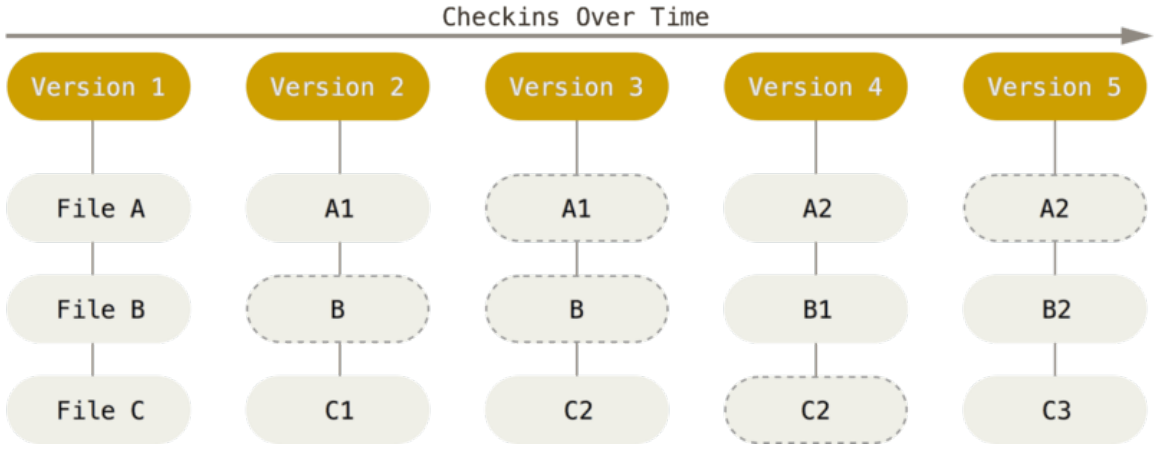


FIGURE 2.1 – Instantanés successifs d'un projet et réutilisation des contenus inchangés. Source : *Pro Git*.

> Une version de notre rapport peut réunir `main.tex` et une illustration. Si seul le texte est corrigé, l'instantané suivant décrit le texte corrigé et la même illustration. L'absence de modification de l'image ne la fait pas disparaître de la version du projet. Inversement, les fichiers présents dans le répertoire mais qui ne sont pas pris en charge par Git ne font pas automatiquement partie de cet instantané.

Un instantané représente une version complète du projet, sans que Git recopie nécessairement tous les fichiers à chaque étape.

2.2 Commit et identification d'une version

2.2.1 Une étape accompagnée de son contexte

Un *commit* est un enregistrement qui associe un instantané du projet à des informations permettant de le situer dans l'historique. Il comporte notamment une identité d'auteur, des informations de date, un message et, sauf pour le premier commit, un lien vers un ou plusieurs commits parents. Dans une succession simple de validations, le parent correspond à l'étape précédente. Le premier commit constitue le point de départ de cet historique ; un commit de fusion peut réunir plusieurs lignes d'évolution.

Le message décrit l'intention de l'étape enregistrée.

> Pour notre rapport, « Correction de la valeur présentée dans les résultats » explique mieux le changement que « Modification du fichier ». Le lien avec la version précédente permet ensuite de retrouver le texte effectivement modifié. Le contenu du commit et son message se

complètent ainsi pour rendre la contribution compréhensible.

Deux commits peuvent décrire le même contenu de fichiers tout en constituant des enregistrements différents, par exemple si leur message ou leur place dans l'historique diffère. Un commit ne se réduit donc ni au texte de son message ni à un simple numéro de version du projet. Il désigne une étape précise, avec son contenu et son contexte.

2.2.2 Identifiants et intégrité des données

Git attribue à ses objets, dont les commits, des identifiants calculés à partir de leur contenu par une fonction de hachage. Un identifiant de commit apparaît sous la forme d'une suite de caractères hexadécimaux ; les affichages courants peuvent en utiliser une forme abrégée suffisamment longue pour distinguer le commit dans le dépôt. Cet identifiant sert à désigner un enregistrement sans dépendre d'un nom de fichier ou d'un numéro choisi manuellement.

Le calcul des identifiants participe au contrôle de l'intégrité des données conservées. Modifier le contenu d'un commit conduit à un autre objet et à un autre identifiant. Cela ne constitue toutefois pas une preuve de l'exactitude scientifique du rapport ni une authentification de l'identité d'auteur déclarée. Le rôle de Git est de conserver et de relier les versions enregistrées ; la qualité de leur contenu relève du travail de l'équipe.

La stabilité d'un commit ne signifie pas que toute présentation de l'historique serait définitivement immuable. Certaines opérations peuvent construire d'autres commits et modifier les étapes auxquelles une ligne de travail fait référence. Ces opérations ne sont pas étudiées ici. Dans l'usage présenté dans ce cours, de nouvelles validations prolongent l'historique et conservent les étapes précédentes.

2.3 Les trois espaces de travail

2.3.1 Le répertoire de travail

Le *répertoire de travail*, ou *working directory*, contient les fichiers sur lesquels l'auteur intervient avec ses outils habituels. L'éditeur y lit et y écrit le texte ; le compilateur y trouve les sources nécessaires à la production du document. Son contenu peut différer de la dernière version enregistrée dans le dépôt.

> Pour notre rapport, la correction d'une phrase dans `main.tex` modifie d'abord le répertoire de travail. La compilation peut également y produire `main.pdf`, `main.aux` et `main.log`. Leur présence sur le disque ne suffit pas à les inclure dans l'historique. Le choix des sources à versionner et l'exclusion des fichiers générés seront étudiés avec les commandes locales.

2.3.2 L'index, préparation du prochain instantané

L'*index*, appelé *staging area* en anglais, décrit le contenu préparé pour le prochain commit. L'indexation consiste à y sélectionner une version du contenu d'un fichier. Elle ne consiste pas à désigner un fichier dont Git prendrait automatiquement toutes les modifications futures. Si ce fichier est de nouveau modifié après son indexation, l'index et le répertoire de travail peuvent donc contenir deux versions différentes de son texte.

> Lors de la rédaction de notre rapport, l'auteur peut préparer la correction d'une valeur de `main.tex`, puis poursuivre la rédaction d'un autre passage du même fichier. Tant qu'aucune nouvelle indexation ne remplace le contenu préparé, le prochain commit conserve la correction sélectionnée, sans le passage ajouté ensuite. L'index permet ainsi de choisir une étape cohérente sans imposer que tout le travail en cours soit terminé.

Les contenus déjà suivis qui ne sont pas modifiés restent représentés dans l'index. Le prochain instantané peut donc décrire l'ensemble du projet, même si l'auteur n'indexe qu'un fichier modifié. L'index est une préparation de version, et non une simple liste de fichiers récemment touchés.

2.3.3 Le dépôt local et la circulation des contenus

Le *dépôt local* conserve les commits et les données auxquelles ils donnent accès. La validation enregistre un instantané correspondant au contenu préparé dans l'index et l'associe à un nouveau commit. Elle ne publie pas ce commit sur un serveur et ne sélectionne pas automatiquement les changements supplémentaires du répertoire de travail.

La figure 2.2 représente ces trois espaces. La flèche *Stage Fixes* correspond à la préparation dans l'index, et la flèche *Commit* à l'enregistrement dans le dépôt. La flèche *Checkout the project* représente la mise à disposition, dans le répertoire de travail, des fichiers d'une version conservée dans le dépôt. Ces flèches décrivent des opérations locales ; aucun échange avec GitLab n'est représenté.

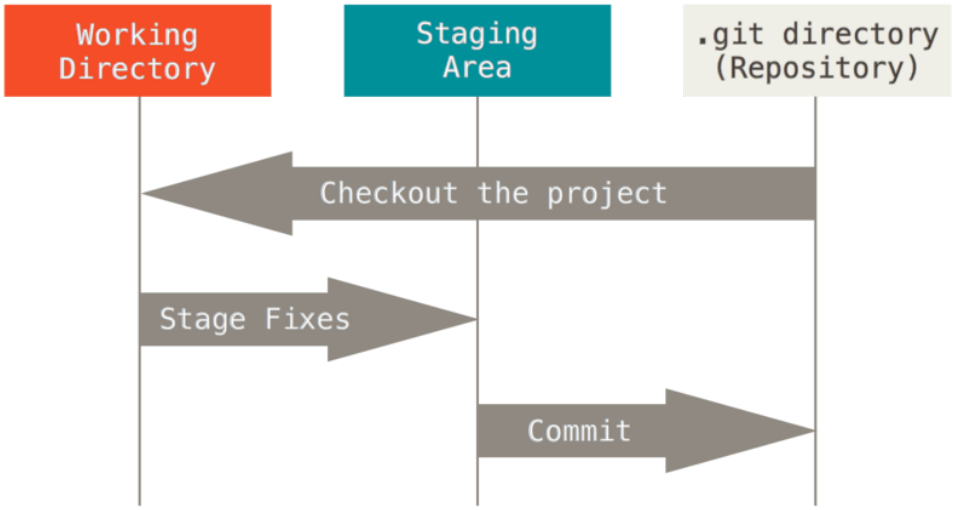


FIGURE 2.2 – Répertoire de travail, index et dépôt local. Source : *Pro Git*.

> Pour notre rapport, ces espaces répondent à trois questions différentes : quel texte l'auteur est-il en train de modifier, quel texte a-t-il préparé pour la prochaine validation et quelles versions a-t-il déjà enregistrées ? Les distinguer évite de confondre une sauvegarde dans l'éditeur, une indexation et un commit.

2.4 États des fichiers et transitions

2.4.1 Fichiers non suivis et fichiers suivis

Un fichier *non suivi*, ou *untracked*, est présent dans le répertoire de travail sans être représenté dans l'index ni dans le dernier commit de la ligne de travail courante. Git peut signaler sa présence, mais ne l'inclut pas automatiquement dans une validation. L'indexation d'un nouveau fichier prépare son introduction dans le prochain instantané. Il est alors pris en charge par Git avant même que le premier commit qui le contient ait été créé.

> Pour notre rapport, un nouveau fichier contenant une annexe est d'abord non suivi. Son indexation prépare son ajout au rapport versionné, puis un commit enregistre cet ajout. Les fichiers auxiliaires de compilation peuvent, de leur côté, rester hors du suivi. Le fait qu'un

> fichier soit utile à un outil ou présent sur le disque ne détermine donc pas son inclusion dans le dépôt.

2.4.2 Contenus non modifiés, modifiés et indexés

Pour les fichiers suivis, l'état dépend des comparaisons entre le dernier commit, l'index et le répertoire de travail. Dans le cas simple où ces trois contenus coïncident, le fichier est *non modifié*, ou *unmodified*. Après une édition, si l'index correspond encore au dernier commit, le répertoire de travail contient une modification non indexée : le fichier est *modifié*, ou *modified*. L'indexation de cette nouvelle version prépare alors une modification pour le prochain commit : elle est *indexée*, ou *staged*.

La figure 2.3 résume les transitions les plus courantes. *Add the file* représente l'introduction d'un fichier non suivi dans l'index. *Edit the file* correspond à l'édition, puis *Stage the file* à l'indexation de la modification. Après la validation représentée par *Commit*, le fichier redevient non modifié si son contenu de travail n'a pas changé depuis l'indexation.

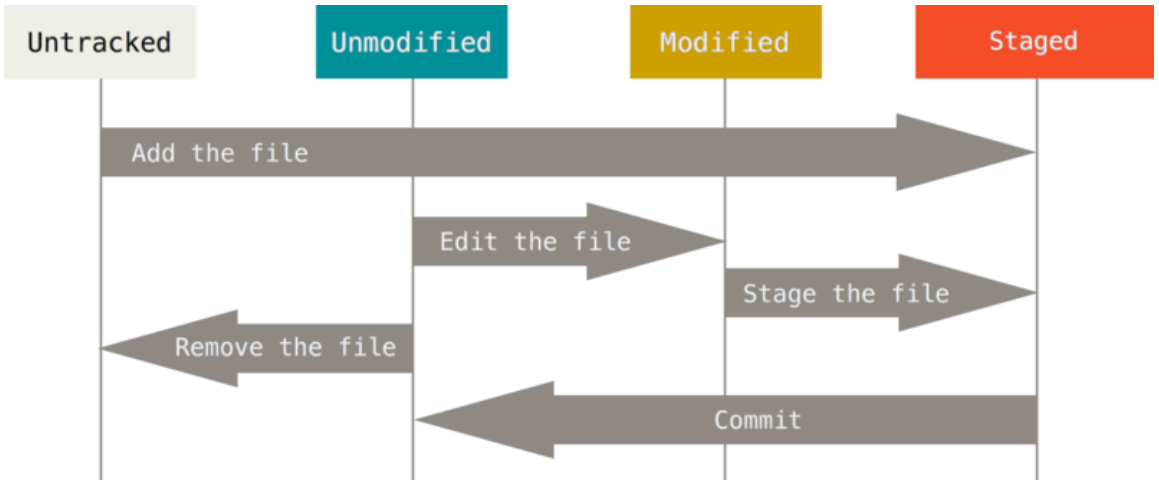


FIGURE 2.3 – Cycle simplifié des états d'un fichier. Source : *Pro Git*.

Ce schéma est une vue d'ensemble, pas une description exhaustive des situations possibles. En particulier, sa flèche *Remove the file* résume un arrêt du suivi sans détailler la suppression préparée dans l'index et son enregistrement ultérieur. Une suppression ne fait pas disparaître le fichier des commits antérieurs. La différence entre suppression du fichier et arrêt de son suivi sera précisée lors de l'étude des commandes correspondantes.

2.4.3 Un fichier peut être indexé puis modifié à nouveau

Les états précédents ne constituent pas quatre cases toujours exclusives. Un même fichier peut présenter un changement déjà indexé par rapport au dernier commit et un autre changement non indexé par rapport à l'index. La comparaison des trois espaces permet de décrire cette situation sans ambiguïté.

> Notre rapport comporte d'abord une valeur erronée. L'auteur corrige cette valeur et indexe le fichier, puis ajoute un paragraphe sans renouveler l'indexation. Le tableau 2.1 décrit les contenus de `main.tex` à chaque étape. Les lettres A, B et C désignent ici des états du texte, et non des identifiants de commits. A est le texte initial, B le texte avec la valeur corrigée et C le texte corrigé enrichi du nouveau paragraphe.

Situation	Dernier commit	Index	Travail
État initial enregistré	A	A	A
Correction de la valeur	A	A	B
Indexation de la correction	A	B	B
Ajout du paragraphe	A	B	C
Validation de la correction	B	B	C

TABLE 2.1 – Évolution des trois contenus de `main.tex` au cours de la rédaction de notre rapport.
Juste avant la validation, le fichier comporte donc une correction indexée et un ajout non indexé. Le commit enregistre B, qui correspond au contenu préparé, tandis que le répertoire de travail conserve C. Le nouveau paragraphe n'est ni perdu ni enregistré dans ce commit : il reste une modification locale en attente. Une nouvelle indexation serait nécessaire pour le préparer à une validation ultérieure.

Ce raisonnement relie les notions du chapitre : un commit conserve un instantané du projet, cet instantané est préparé dans l'index et le répertoire de travail peut continuer à évoluer séparément. Les commandes locales permettront d'observer ces différences, de choisir les contenus à indexer et d'enregistrer des étapes compréhensibles.

Chapitre 3

Enregistrer et examiner un travail local

Le modèle présenté au chapitre 2 prend une forme concrète avec les commandes locales de Git. L'auteur observe d'abord l'état du projet, sélectionne les contenus à conserver dans l'index, puis enregistre une étape cohérente par un commit. La lecture des différences et de l'historique permet de vérifier ce qui est préparé et ce qui a été conservé.

> Le rapport en LaTeX qui sert de fil conducteur comporte un premier fichier `main.tex` contenant une valeur à corriger. Les séquences de commandes décrivent sa première validation, puis l'enregistrement de la correction et la poursuite de la rédaction. Les commandes de suppression sont présentées dans des situations distinctes, afin de conserver le rapport utilisé comme fil conducteur.

Toutes les validations de ce chapitre restent locales : elles ne publient rien sur un serveur Git.

3.1 Environnement et aide

3.1.1 Une commande, des actions et des options

Depuis un terminal, le programme `git` reçoit le nom d'une action, puis les options et les arguments nécessaires à cette action. Par exemple, `git status` demande un état du projet, tandis que `git add` sélectionne le contenu des fichiers désignés. Les commandes qui agissent sur un dépôt supposent que le répertoire courant se trouve dans ce dépôt. Les séquences de commandes utilisent le répertoire du projet comme répertoire courant. Une commande exécutée au mauvais endroit peut signaler que le répertoire n'appartient pas à un dépôt Git.

Les commandes suivantes permettent de connaître la version installée et d'accéder à une aide sur une action particulière. Le symbole `$` représente l'invite du terminal et ne fait pas partie de la commande.

```
$ git --version
$ git help add
$ git add -h
```

`git help add` ouvre la documentation détaillée de l'indexation lorsque cette documentation est installée. L'option `-h` fournit une aide courte dans le terminal. Ces consultations ne modifient ni les fichiers du projet ni son historique. Les sorties de Git dépendent notamment de sa version et de la langue configurée ; les séquences ci-dessous retiennent des affichages courts pour mettre en évidence les informations utiles.

3.1.2 L'identité associée aux commits

La création d'un commit nécessite une identité d'auteur, composée notamment d'un nom et d'une adresse électronique.

Pour notre rapport, la configuration suivante définit l'identité de Camille pour les dépôts utilisés par son compte sur la machine. Le nom et l'adresse sont fictifs.

```
$ git config --global user.name "Camille Exemple"
$ git config --global user.email "camille@example.invalid"
```

L'option `--global` donne à cette configuration une portée commune aux dépôts de l'utilisateur. Sans cette option, la même configuration, effectuée dans un dépôt existant, s'applique à ce seul dépôt et permet de remplacer les valeurs générales. Ces réglages déterminent les informations associées aux futurs commits ; ils ne modifient pas ceux qui existent déjà. Ils ne constituent pas des identifiants de connexion à un serveur Git. Si l'identité nécessaire n'est pas disponible, Git peut refuser la validation et signaler les informations manquantes.

3.2 Initialisation et observation du dépôt

3.2.1 Créer le dépôt avec `git init`

Les sources de notre rapport sont regroupées dans un répertoire nommé `rapport`. La séquence suivante décrit sa création, le déplacement du terminal dans ce répertoire, puis l'initialisation du dépôt Git.

```
$ mkdir rapport
$ cd rapport
$ git init
```

`mkdir` et `cd` sont des commandes du terminal ; `git init` crée les données du dépôt, notamment le répertoire `.git`. La commande peut aussi être exécutée dans un répertoire qui contient déjà des fichiers : elle ne les efface pas et ne les enregistre pas automatiquement. À ce stade, aucun commit n'existe encore. Le nom de la branche initiale peut apparaître dans les messages de Git et dépend de sa configuration ; les séquences de commandes ne supposent pas qu'il s'agisse de `main` ou de `master`.

Le fichier initial du rapport est donné au listing 3.1. Il s'agit d'un document complet, dont la compilation produit un PDF d'une page. La valeur 10 servira ensuite à illustrer une correction.

Listing 3.1 – Fichier `main.tex` dans la version initiale de notre rapport.

```
\documentclass[a4paper]{article}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\usepackage[french]{babel}

\begin{document}
\section{Résultats}
La valeur mesurée est 10.
\end{document}
```

3.2.2 Lire l'état du projet avec `git status`

La commande `git status` décrit l'état des fichiers par rapport à l'index et à la dernière version enregistrée. Elle distingue notamment les modifications préparées pour un commit, celles

qui restent dans le répertoire de travail et les fichiers non suivis. Elle sert à observer le dépôt, sans enregistrer de version ni modifier les fichiers de travail.

> Pour notre rapport, une compilation du fichier produit `main.aux`, `main.log` et `main.pdf` à côté du source. Avant toute indexation ou exclusion, l'affichage court peut être le suivant.
>
> ```
> $ pdflatex main.tex
> $ git status --short
> ?? main.aux
> ?? main.log
> ?? main.pdf
> ?? main.tex
> ```
>
> Les deux points d'interrogation indiquent des fichiers non suivis. L'option `--short` fournit une présentation compacte ; sans cette option, Git détaille les catégories et les opérations possibles. La présence de `main.tex` dans cette liste confirme qu'une initialisation ne suffit pas à commencer son suivi. Les fichiers générés par la compilation seront exclus à la section 3.4.

3.3 Sélection et enregistrement des contenus

3.3.1 Préparer le contenu avec `git add`

`git add` met à jour l'index à partir du contenu des fichiers désignés. Pour l'ajout présenté ici, le fichier doit exister au chemin indiqué et le terminal doit se trouver dans le dépôt.

> Pour notre rapport, l'indexation porte seulement sur le fichier source, sans inclure les produits de compilation.
>
> ```
> $ git add main.tex
> $ git status --short
> A  main.tex
> ?? main.aux
> ?? main.log
> ?? main.pdf
> ```
>
> La lettre `A`, pour *added*, indique que l'ajout de `main.tex` est préparé dans l'index. La commande ne crée pas encore de commit et ne modifie pas le fichier de travail.

Une modification effectuée après cette indexation demanderait une nouvelle sélection pour être incluse dans le prochain commit. Un nom de fichier incorrect entraîne une erreur : Git ne peut pas indexer un contenu situé à un chemin qui ne correspond pas à un fichier connu ou présent.

Une même commande peut désigner plusieurs fichiers lorsque leurs changements forment un tout.

> Pour notre rapport, la commande `git add main.tex conclusion.tex` préparerait ainsi les contenus de ces deux fichiers.

Le choix des fichiers est donc lié au sens de l'étape à enregistrer. Une sélection explicite facilite ce contrôle, en particulier lorsque le répertoire contient aussi des fichiers générés.

3.3.2 Valider une étape avec `git commit`

`git commit` crée un commit à partir de l'instantané préparé dans l'index. Dans l'usage présenté ici, l'identité d'auteur est configurée et l'index contient un changement à enregistrer. L'option `-m` fournit le message directement dans la commande.

> Pour notre rapport, la première validation est créée par la commande suivante.

```
$ git commit -m "Version initiale du rapport"
```

Git affiche notamment un identifiant abrégé du commit, son message et un bilan des fichiers modifiés. Les valeurs exactes de cet affichage varient selon le contenu et les informations du commit. Après cette première validation, `main.tex` est enregistré et ne présente plus de modification si son texte n'a pas changé depuis l'indexation. Les fichiers auxiliaires restent non suivis.

Une nouvelle exécution de `git commit` sans changement préparé ne crée pas une nouvelle version dans ce parcours : Git indique qu'il n'y a rien à valider. La seule modification d'un fichier déjà suivi ne suffit pas non plus ; elle doit d'abord être sélectionnée dans l'index. Cette distinction évite de croire qu'une validation enregistre automatiquement tout le répertoire de travail.

Un commit cohérent correspond à une intention identifiable : un premier document, une correction de valeur ou l'ajout d'une section complète. Un message précis permet de retrouver cette intention dans l'historique. La validation conserve l'étape sur la machine de l'auteur ; sa publication vers un dépôt distant relève d'une autre opération.

3.4 Sources et fichiers générés

3.4.1 Choisir ce qui appartient au projet versionné

Le dépôt conserve les fichiers sources nécessaires à la réalisation du projet. Tout fichier résultant d'une compilation doit rester hors du dépôt : il ne doit être ni indexé ni ajouté à l'historique. Cette règle concerne aussi bien les fichiers auxiliaires que le document PDF compilé, puisqu'ils peuvent être régénérés à partir des sources. Leur évolution peut en outre dépendre de l'environnement de compilation plutôt que d'un changement de contenu.

Pour notre rapport, les sources sont le texte et les illustrations utilisées. Le fichier `main.tex` est versionné, tandis que `main.aux`, `main.log` et le PDF compilé restent hors du suivi. Une illustration au format PDF peut en revanche être une ressource du document et devoir être conservée. L'exclusion du PDF produit par la compilation sera donc ciblée sur son nom, sans exclure toutes les illustrations portant cette extension.

3.4.2 Décrire les exclusions dans `.gitignore`

Un fichier nommé `.gitignore` contient des motifs désignant les fichiers non suivis que Git ne doit pas proposer habituellement à l'ajout. Il ne supprime aucun fichier du disque et ne retire pas du suivi ceux qui ont déjà été enregistrés.

Pour notre rapport, le fichier présenté au listing 3.2 est placé à la racine du dépôt.

Listing 3.2 – Exclusions adaptées à notre rapport.

```
*.aux
*.log
*.out
*.synctex.gz
/main.pdf
```

Le motif `*.aux` désigne les noms qui se terminent par `.aux` ; les autres motifs commençant par une étoile suivent le même principe. Le motif `/main.pdf` cible le PDF du rapport à la racine du dépôt. Une illustration telle que `images/schema.pdf` reste ainsi disponible pour le suivi. Ce fichier d'exclusion est volontairement limité aux besoins de notre rapport ; un document plus complexe peut produire d'autres fichiers auxiliaires.

`.gitignore` appartient lui-même aux sources du projet : son enregistrement permet aux contributeurs de partager les mêmes exclusions.

Pour notre rapport, la séquence suivante prépare puis valide le fichier d'exclusion.

```
$ git add .gitignore
$ git commit -m "Exclusion des fichiers de compilation"
$ git status --short
```

En l'absence d'autres modifications, l'affichage court est désormais vide. Les fichiers compilés existent toujours sur le disque, mais ne figurent plus dans la liste habituelle des fichiers non suivis.

Une tentative d'ajout explicite d'un fichier ignoré peut être refusée avec un message expliquant son exclusion. Lorsqu'un fichier est déjà suivi, l'ajout d'un motif dans `.gitignore` ne suffit pas : l'arrêt du suivi nécessite l'opération décrite à la section 3.5.2.

3.5 Suppression et arrêt du suivi

3.5.1 Préparer une suppression avec `git rm`

`git rm` retire un fichier suivi du répertoire de travail et prépare son absence dans l'index. Le prochain commit peut alors enregistrer cette suppression. Le fichier demeure consultable dans les commits antérieurs qui le contenaient. La commande ne correspond donc pas à un effacement de toutes ses versions.

Dans un premier cas d'utilisation, une annexe provisoire nommée `annexe.tex` a déjà été enregistrée, mais n'est plus utile au rapport. Le fichier n'a pas de modification en attente et la séquence suivante décrit son retrait.

```
$ git rm annexe.tex
rm 'annexe.tex'
$ git status --short
D  annexe.tex
$ git commit -m "Retrait de l'annexe provisoire"
```

La lettre `D`, pour *deleted*, indique la suppression préparée dans l'index. Avant le commit, le dernier état enregistré contient encore l'annexe.

Si un fichier comporte des changements qui risqueraient d'être perdus, Git peut refuser sa suppression : ce refus invite à examiner son état plutôt qu'à forcer l'opération. Le retrait d'un fichier encore utilisé par le document nécessite également une modification cohérente du texte qui le référence.

3.5.2 Conserver le fichier avec `git rm --cached`

L'option `--cached` retire le fichier de l'index tout en conservant sa copie dans le répertoire de travail. Le prochain commit enregistre son retrait du projet versionné. Cette opération sert notamment à cesser de suivre un produit de compilation précédemment enregistré par erreur. Elle ne retire pas le fichier des versions antérieures de l'historique.

Dans un second cas d'utilisation, `main.pdf` était déjà suivi avant l'adoption des exclusions. Le PDF de travail correspond au contenu enregistré, et `.gitignore` contient désormais le motif `/main.pdf`. La séquence suivante cesse de le suivre sans supprimer le PDF présent sur la machine.

```
$ git rm --cached main.pdf
rm 'main.pdf'
$ git commit -m "Arrêt du suivi du PDF compilé"
```

Malgré le message `rm` affiché, le fichier local reste présent grâce à l'option `--cached`. Le motif d'exclusion évite qu'il soit ensuite proposé comme fichier non suivi. Sans ce motif, la copie

conservée pourrait apparaître dans cette catégorie et être ajoutée à nouveau. Cette situation se distingue du simple retrait d'une modification de l'index : ici, le fichier cesse d'appartenir au prochain instantané.

3.6 Lecture des différences et de l'historique

3.6.1 Comparer le travail courant et l'index

`git status` indique quels fichiers ont changé, mais ne décrit pas le texte de ces changements. `git diff` complète cette observation en comparant, par défaut, le répertoire de travail à l'index pour les fichiers suivis. La commande ne modifie aucun de ces contenus. Un fichier non suivi n'apparaît pas dans cette comparaison habituelle.

La rédaction de notre rapport se poursuit à partir de l'état obtenu après la validation des exclusions : `main.tex` et `.gitignore` sont enregistrés, et aucun changement n'est en attente. La valeur 10 est alors remplacée par 12 dans le texte. La commande `git diff` permet d'observer cette correction avant son indexation. L'extrait suivant ne conserve que les lignes de contenu qui changent.

```
$ git diff
-La valeur mesurée est 10.
+La valeur mesurée est 12.
```

Le signe `-` désigne une ligne de l'ancien contenu et le signe `+` une ligne du nouveau contenu. L'affichage complet comporte également des en-têtes identifiant le fichier et la zone comparée. Ces signes font partie de la présentation des différences, pas du texte du rapport.
Après `git add main.tex`, le répertoire de travail et l'index contiennent la même correction. `git diff` n'affiche donc plus de différence, même si cette correction n'a pas encore été validée. L'absence de sortie ne signifie pas à elle seule qu'aucun changement n'attend un commit.

3.6.2 Vérifier ce qui sera enregistré

`git diff --cached` compare le contenu préparé dans l'index à celui du dernier commit. Elle permet ainsi de relire les changements destinés à la prochaine validation. Avant le premier commit, cette commande présente les contenus préparés comme des ajouts. Comme `git diff`, elle observe les contenus sans les modifier.

Pour notre rapport, la correction est indexée, puis l'auteur ajoute un paragraphe sans indexer ce dernier ajout. Les deux commandes suivantes portent alors sur des changements différents.

```
$ git diff --cached
$ git diff
```

La première présente la correction de 10 en 12, déjà préparée dans l'index. La seconde présente le paragraphe supplémentaire, présent seulement dans le répertoire de travail. `git status --short` affiche alors `MM main.tex` : le fichier comporte à la fois une modification indexée et une modification non indexée. Dans cet affichage, la première colonne décrit l'état de l'index par rapport au dernier commit, et la seconde l'état du travail par rapport à l'index. Les deux lettres `M` signifient ici *modified*.
La validation suivante conserve seulement la correction préparée.

```
$ git commit -m "Correction de la valeur mesurée"
```

Le paragraphe ajouté reste dans le fichier de travail après le commit. La situation correspond à celle du tableau 2.1 : une nouvelle validation ne prendrait ce paragraphe en compte qu'après une nouvelle indexation. Cette séquence montre pourquoi la vérification de l'index est utile même lorsque l'auteur connaît le texte actuellement affiché dans son éditeur.

3.6.3 Retrouver les étapes avec `git log`

`git log` présente les commits accessibles depuis la version courante. L'affichage habituel donne notamment leur identifiant, leur auteur, leur date et leur message. L'option `--oneline` fournit un aperçu plus compact avec un identifiant abrégé et la première ligne du message. Ces consultations ne changent ni les fichiers ni l'historique.

```
$ git log
$ git log --oneline
```

> L'historique de notre rapport contient alors trois étapes : la version initiale du rapport, l'exclusion des fichiers de compilation et la correction de la valeur mesurée. Les validations se succèdent sans divergence et sont présentées du plus récent au plus ancien. Le paragraphe encore non indexé n'y figure pas. Le nom d'auteur affiché provient de l'identité configurée au moment de chaque validation, et les identifiants dépendent des commits réellement créés.

Si le dépôt ne contient encore aucun commit, Git ne peut pas présenter un tel historique et le signale.

Le travail local s'organise ainsi autour d'observations et de choix explicites. `status` et `diff` décrivent les changements, `add` prépare ceux qui forment une étape cohérente, `commit` l'enregistre et `log` permet de la retrouver. Le partage de ces étapes avec d'autres auteurs nécessitera les échanges avec un dépôt distant.

Chapitre 4

Échanger avec un dépôt distant

Les commits créés localement conservent les étapes du travail de leur auteur. Pour qu'une équipe dispose de ces contributions, elles doivent circuler entre ses dépôts. Un dépôt partagé hébergé sur un serveur Git sert de point de rencontre : chacun peut y publier ses commits et recevoir ceux des autres.

> La rédaction de notre rapport en LaTeX se poursuit désormais à deux. Camille et Samir disposent chacun de leur machine et rejoignent un dépôt partagé contenant une version enregistrée du document. Le chapitre décrit d'abord le clonage de ce dépôt, puis la réception et la publication de nouvelles versions. Une chronologie de contributions successives montre comment ces opérations s'articulent sans conflit.

4.1 Dépôts locaux et dépôt partagé

4.1.1 Le rôle du serveur Git

Un *dépôt distant* est un dépôt avec lequel le dépôt local échange des données. Dans l'organisation étudiée ici, il est hébergé sur un serveur accessible par le réseau. Les contributeurs l'utilisent comme dépôt partagé, mais continuent à enregistrer leurs propres versions dans leurs dépôts locaux. L'hébergement commun ne transforme donc pas Git en système centralisé.

> Pour notre rapport, le dépôt partagé contient le fichier `main.tex` avec la valeur corrigée à 12 et le fichier `.gitignore` défini au chapitre 3. Sa mise à disposition sur le serveur est supposée réalisée. Le paragraphe qui restait non indexé à la fin du travail local n'appartient pas à cette version enregistrée. Il ne sera pas transmis par les échanges de commits décrits dans ce chapitre.

La présence d'une contribution sur le serveur ne met pas automatiquement à jour les fichiers de tous les contributeurs. Chaque dépôt conserve son propre état jusqu'à un nouvel échange.

> Pour notre rapport, Samir peut ainsi travailler sur une version antérieure alors que Camille a déjà publié une correction. Cette différence est normale : les dépôts évoluent séparément entre deux synchronisations.

4.1.2 Un nom pour le distant et une branche suivie

Git associe un nom et une adresse à chaque dépôt distant avec lequel le dépôt local est configuré pour échanger. Après un clonage usuel, le dépôt d'origine porte le nom local `origin`. Ce nom est une convention créée par Git ; il ne désigne ni un logiciel de forge ni un serveur unique pour tous les projets.

Le clonage prépare également une association entre la branche de travail initiale et la branche correspondante du dépôt d'origine. On parle de *branche suivie*. Cette association permet à Git

de déterminer la source habituelle des changements à recevoir. Dans la configuration usuelle retenue ici, elle permet aussi de publier sur la branche correspondante avec une commande sans arguments supplémentaires. La manipulation des branches sera approfondie dans la suite de votre formation.

> Pour notre rapport, les deux auteurs utiliseront chacun le nom `origin` pour désigner le même dépôt partagé. Ils conserveront l'association établie au clonage. Il n'est donc pas nécessaire de choisir un nom de branche dans les séquences de commandes présentées.

4.2 Clonage et accès au dépôt

4.2.1 Obtenir une copie avec `git clone`

`git clone` crée un nouveau dépôt local à partir d'un dépôt existant. Dans un clonage complet usuel, elle récupère l'historique, configure le distant `origin` et met à disposition les fichiers de la branche initiale dans un répertoire de travail. L'accès au dépôt source doit être autorisé et son adresse doit être correcte. Le répertoire de destination doit être absent ou vide. Un dépôt déjà initialisé avec `git init` n'est pas nécessaire pour cette opération.

> Pour notre rapport, Camille rejoint le dépôt partagé en créant un nouveau répertoire `rapport-camille`. La séquence suivante utilise une adresse fictive, destinée à montrer la forme de la commande. Dans ce cours, `git.exemple.com` désigne un serveur fictif.
>
> ```
> $ git clone https://git.exemple.com/equipe/rapport.git rapport-camille
> $ cd rapport-camille
> ```

Le dernier argument choisit le nom du répertoire local. Sans cet argument, Git déduit habituellement ce nom de celui du dépôt source. `cd` place ensuite le terminal dans le nouveau répertoire afin que les commandes suivantes concernent ce dépôt. Une adresse inconnue, un accès refusé ou une destination déjà occupée par des fichiers peuvent empêcher le clonage.

Les fichiers non suivis du répertoire de travail d'un autre auteur ne sont pas copiés avec son historique. La configuration d'identité de chaque utilisateur n'est pas non plus reprise comme une propriété du projet.

> Pour notre rapport, le clonage récupère les sources enregistrées et `.gitignore`, mais pas les produits de compilation exclus ni le paragraphe resté hors de l'historique. Camille et Samir utilisent chacun leur propre identité pour leurs futurs commits.

4.2.2 SSH et HTTPS

Un serveur Git peut proposer plusieurs modes d'accès, notamment HTTPS et SSH. Le protocole détermine la manière d'établir la connexion et de s'authentifier ; il ne change pas le rôle des commits ni celui de l'index. Un accès en lecture peut suffire au clonage, tandis que la publication nécessite une autorisation d'écriture.

En HTTPS, l'adresse commence par `https://`. Selon la configuration du service, l'accès à un dépôt privé ou l'écriture peuvent nécessiter une authentification, par exemple avec un jeton d'accès. En SSH, l'authentification repose habituellement sur une clé dont la partie publique est associée au compte sur le serveur.

> Pour notre rapport, une adresse SSH du dépôt partagé pourrait prendre la forme suivante, également fictive.
>
> `git@git.exemple.com:equipe/rapport.git`
>
> Dans cette notation, `git` est le nom du compte technique utilisé pour la connexion SSH, et non le nom d'auteur enregistré dans les commits.

Les modalités d'authentification et les restrictions réseau dépendent du service utilisé. Le choix entre SSH et HTTPS repose donc sur les accès réellement proposés par le serveur et sur la configuration de la machine.

La commande `git remote -v` affiche les adresses des distants configurés, sans établir de nouvelle connexion ni modifier l'historique.

> Pour le clone de Camille, cet affichage peut être le suivant.
>
> ```
> $ git remote -v
> origin  https://git.exemple.com/equipe/rapport.git (fetch)
> origin  https://git.exemple.com/equipe/rapport.git (push)
> ```
>
> Les mentions `fetch` et `push` identifient les adresses utilisées respectivement pour recevoir et pour publier. Elles sont identiques dans ce cas. L'affichage permet de vérifier à quel dépôt les échanges suivants seront adressés.

4.3 Recevoir les changements avec `git pull`

4.3.1 Récupérer puis intégrer les commits

`git pull` effectue deux opérations : elle récupère des données du dépôt distant, puis intègre les commits de la branche suivie dans la branche de travail courante. La récupération correspond à une opération appelée *fetch*. L'intégration met à jour la version locale en tenant compte de son historique. Il ne s'agit donc pas d'un simple remplacement des fichiers locaux par ceux du serveur.

Dans ce chapitre, les contributions se succèdent sans divergence et la commande est utilisée sans option supplémentaire. Le choix du comportement à adopter lorsque les historiques divergent sera présenté au chapitre consacré aux conflits.

```
$ git pull
```

La commande suppose un dépôt local, un distant accessible et une branche suivie correctement configurée. Dans le parcours présenté, les modifications de travail sont enregistrées et le répertoire est propre avant l'échange. Si des modifications non validées (c'est-à-dire non enregistrées dans un commit) risquent d'être écrasées par l'intégration, Git refuse de poursuivre et les signale. Cette situation se distingue d'un conflit entre deux contributions déjà enregistrées.

4.3.2 Une mise à jour sans divergence

Lorsque les nouveaux commits distants prolongent directement l'historique local, Git peut mettre à jour la branche locale jusqu'à la version reçue. Cette mise à jour est appelée une *avance rapide*, ou *fast-forward*. Elle n'a pas besoin de créer un commit de fusion supplémentaire : les nouveaux commits prolongent simplement l'historique connu. Si aucun nouveau commit n'est à intégrer, la commande signale que la version est déjà à jour.

> Pour notre rapport, Samir peut avoir conservé la version commune pendant que Camille publiait une correction. La réception de cette correction met alors à jour son historique local et son fichier `main.tex`. Il peut relire le texte reçu et compiler sa copie avant de poursuivre la rédaction.

Si les deux auteurs ont créé séparément des commits depuis leur dernière version commune, leurs historiques ont divergé. Une intégration peut alors nécessiter une fusion et éventuellement une résolution de conflit. Selon les réglages de Git, une divergence peut aussi provoquer un refus de la mise à jour automatique. Ces situations seront examinées dans le chapitre consacré aux conflits.

4.4 Publier les commits avec `git push`

4.4.1 Transmettre les versions enregistrées

`git push` transmet les commits et les données nécessaires à la mise à jour de la branche destinataire sur le dépôt distant. Dans la configuration usuelle issue du clonage et conservée ici, la commande suivante publie la branche courante vers la branche correspondante sur `origin`. Elle suppose un accès réseau, une authentification valide si nécessaire et une autorisation d'écriture sur cette branche.

```
$ git push
```

La publication porte sur les commits, pas sur l'ensemble du répertoire de travail. Un texte modifié mais non validé n'est pas envoyé ; un contenu seulement indexé ne l'est pas davantage. Plusieurs commits locaux peuvent être transmis lors d'un même échange. Si aucun nouveau commit n'est à publier, Git signale que le distant est déjà à jour pour cette publication.

> Pour notre rapport, une correction de `main.tex` doit donc être indexée puis validée avant de pouvoir être partagée. Le PDF résultant de la compilation reste hors du dépôt, conformément à la règle de la section 3.4.1. Samir recevra la correction lorsqu'il effectuera à son tour une récupération et une intégration dans son dépôt.

4.4.2 Comprendre un refus de publication

Un dépôt distant peut refuser une publication (`git push`) lorsque sa branche contient des commits que la proposition de mise à jour ne conserve pas. Dans le travail collectif usuel, ce refus se produit notamment lorsqu'une autre personne a publié (`git push`) depuis la dernière synchronisation. Git évite ainsi qu'une contribution déjà partagée soit écartée par une mise à jour qui n'en tient pas compte. Le message peut comporter la mention `rejected` ou `non-fast-forward`.

Une telle situation nécessite de recevoir et d'intégrer les contributions manquantes (`git pull`), puis de vérifier l'état du dépôt (`git status`) et le document obtenu avant une nouvelle publication (`git push`). Le refus d'une publication (`git push`) ne signifie pas à lui seul qu'il existe un conflit de contenu. L'intégration (`git pull`) peut être automatique ou demander une décision sur les passages modifiés.

Recevoir et intégrer les changements (`git pull`) avant de publier (`git push`) permet de travailler à partir d'un état plus récent du projet, mais ne garantit pas l'absence de difficultés. Une autre personne peut publier (`git push`) entre ces deux opérations, et les contributions reçues peuvent elles-mêmes être en conflit. Un refus lié aux permissions ou à l'authentification a, de son côté, une autre cause et ne se résout pas en récupérant et en intégrant de nouveaux commits (`git pull`). Le message de Git permet de distinguer ces situations.

4.5 Une collaboration sans conflit

4.5.1 Deux copies du même rapport

Le clonage donne à chaque auteur son propre dépôt et son propre répertoire de travail.

> Pour notre rapport, Camille et Samir rejoignent tous deux la version partagée contenant la valeur 12. Camille dispose déjà du clone `rapport-camille` ; Samir obtient sa copie avant que Camille ne publie sa prochaine contribution.
>
> ```
> $ git clone https://git.exemple.com/equipe/rapport.git rapport-samir
> $ cd rapport-samir
> ```
>
> Les deux répertoires contiennent alors les mêmes sources enregistrées, mais leurs futurs changements resteront indépendants jusqu'aux échanges avec le serveur. L'identité d'auteur

est configurée sur chaque machine. Le fichier `.gitignore` reçu au clonage écarte les fichiers de compilation dans les deux copies.

4.5.2 Une première contribution publiée par Camille

Avant sa contribution, Camille vérifie l'état du dépôt et reçoit les éventuelles mises à jour. La séquence suivante est exécutée depuis `rapport-camille`, avec un répertoire de travail propre.

```
$ git status --short
$ git pull
```

Camille remplace ensuite la valeur 12 par 13 dans le rapport. La lecture des différences permet de contrôler cette seule correction avant son indexation et sa validation. La publication rend le commit accessible sur le dépôt partagé.

```
$ git diff
$ git add main.tex
$ git commit -m "Actualisation de la valeur mesurée"
$ git push
```

À l'issue de cette séquence, le dépôt de Camille et le dépôt partagé contiennent le nouveau commit. La copie de Samir contient encore la valeur 12, puisqu'il n'a pas encore reçu cette contribution. Le `push` de Camille ne modifie donc pas à distance les fichiers ouverts dans l'éditeur de Samir.

4.5.3 Réception puis contribution de Samir

Samir n'a pas encore modifié sa copie et la met à jour depuis le répertoire `rapport-samir`. La réception se fait ici par avance rapide, car il n'a créé aucun commit supplémentaire.

```
$ git status --short
$ git pull
```

La valeur 13 figure désormais dans son fichier `main.tex`. Samir ajoute ensuite le paragraphe « La mesure a été répétée trois fois. » à la suite des résultats. Il peut compiler le document pour vérifier le résultat de sa rédaction, puis préparer et publier cette nouvelle contribution.

```
$ pdflatex main.tex
$ git diff
$ git add main.tex
$ git commit -m "Précision sur le nombre de mesures"
$ git push
```

Le dépôt partagé contient maintenant la correction de Camille et le paragraphe de Samir. Une nouvelle exécution de `git pull` depuis `rapport-camille`, resté sans modification, met à son tour la copie de Camille à jour. Les deux auteurs disposent alors du même texte enregistré. Le tableau 4.1 résume cette circulation. Les noms V0, V1 et V2 désignent les étapes du rapport et ne sont pas des identifiants Git. V0 contient la valeur 12, V1 la valeur 13 et V2 ajoute le paragraphe sur les trois mesures.

Situation	Commande Git	Camille	Serveur	Samir
Après les deux clonages	`git clone`	V0	V0	V0
Camille valide sa correction	`git commit`	V1	V0	V0
Camille publie	`git push`	V1	V1	V0
Samir reçoit la correction	`git pull`	V1	V1	V1
Samir valide son paragraphe	`git commit`	V1	V1	V2
Samir publie	`git push`	V1	V2	V2
Camille reçoit le paragraphe	`git pull`	V2	V2	V2

TABLE 4.1 – Version enregistrée du rapport disponible dans chacun des trois dépôts. La chronologie distingue les validations locales des publications et des réceptions. Le travail se déroule ici sans divergence parce que Samir reçoit la contribution de Camille avant d'enregistrer la sienne, puis Camille reçoit celle de Samir sans avoir créé d'autre commit.

Lorsque les auteurs enregistrent leurs contributions en parallèle, cette succession simple n'est plus assurée. La compréhension des divergences et des conflits complète alors celle des échanges entre dépôts.

Chapitre 5

Comprendre et résoudre un conflit

Le travail collectif ne suit pas toujours une succession où chaque auteur attend la publication du précédent. Deux personnes peuvent enregistrer des contributions différentes à partir d'une même version du projet. Git doit alors réunir ces contributions lors de leur intégration. Lorsque certains changements ne peuvent pas être combinés automatiquement, il signale un conflit et attend une résolution explicite.

> Pour notre rapport, Camille et Samir partent de la version commune obtenue au chapitre 4, qui contient la valeur 13 et la phrase « La mesure a été répétée trois fois. ». Ils corrigent séparément la même valeur, puis valident leurs textes. Le chapitre suit le conflit rencontré par Samir, depuis sa détection jusqu'à la publication d'une version vérifiée.

5.1 Contributions concurrentes et divergence

5.1.1 Deux commits issus d'une même version

Deux historiques divergent lorsque chaque auteur crée des commits qui prolongent leur dernière version commune de manière différente. Une validation locale (`git commit`) ne tient pas compte des validations effectuées au même moment dans un autre dépôt. La divergence apparaît donc même si les auteurs travaillent chacun dans un répertoire propre et enregistrent correctement leurs changements.

> Pour notre rapport, Camille remplace la valeur 13 par 14 dans `main.tex`. La séquence suivante décrit la sélection de la correction, sa validation et sa publication depuis le dépôt de Camille.
>
> ```
> $ git add main.tex
> $ git commit -m "Correction de la valeur mesurée à 14"
> $ git push
> ```
>
> Sans avoir reçu cette contribution, Samir remplace de son côté la valeur 13 par 15. Depuis `rapport-samir`, il sélectionne puis valide son propre changement.
>
> ```
> $ git add main.tex
> $ git commit -m "Correction de la valeur mesurée à 15"
> ```
>
> Les deux commits ont le même point de départ, mais aucun n'inclut la correction de l'autre auteur. Le dépôt partagé contient déjà la contribution de Camille, tandis que le dépôt de Samir comporte un commit supplémentaire qui n'y figure pas. Une tentative de publication par Samir (`git push`) est donc refusée dans cette situation. Ce refus protège la contribution déjà publiée ; il ne résout pas la différence entre les deux textes.

5.1.2 Une fusion automatique ou un conflit

Pour intégrer des contributions divergentes, Git compare les changements aux contenus de leur version commune. Des changements sur des passages distincts peuvent souvent être combinés automatiquement, y compris s'ils concernent le même fichier. Une divergence n'implique donc pas nécessairement un conflit. Lorsque Git ne peut pas construire automatiquement un résultat pour certains passages, il les signale pour qu'une personne décide du texte à retenir.

Dans notre rapport, les corrections de Camille et de Samir remplacent la même ligne par deux textes différents. La valeur correcte dépend du contenu du rapport, pas de l'ordre dans lequel les commits ont été publiés. Git peut conserver les deux propositions dans les historiques, mais ne peut pas déterminer à la place des auteurs si le texte final doit indiquer 14 ou 15.
La figure 5.1 représente la circulation des contributions entre deux auteurs et un serveur. Les lignes horizontales correspondent à l'évolution des dépôts, de gauche à droite. Les flèches de clonage, de réception et de publication représentent les échanges entre ces dépôts. La zone marquée « Conflit » situe l'arrêt de l'intégration ; la flèche suivante décrit la résolution et la validation locales avant une nouvelle publication.

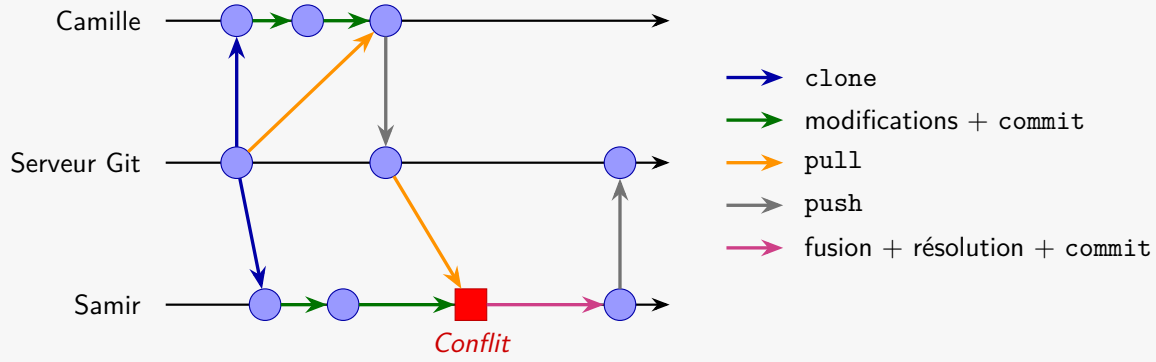


FIGURE 5.1 – Chronologie d'une collaboration comportant une résolution de conflit.
Pour notre rapport, la contribution de Camille est publiée en premier. Samir reçoit cette contribution alors qu'il a déjà validé la sienne. La résolution s'effectue dans le dépôt et les fichiers de Samir, avant de partager son résultat avec l'équipe.

5.2 Résoudre un conflit de fusion

5.2.1 Choisir la fusion lors d'une divergence

Selon sa version et sa configuration, Git peut refuser de poursuivre une réception (`git pull`) lorsque les historiques divergent et qu'aucun comportement d'intégration adapté n'a été choisi. Il affiche alors un message indiquant les réglages possibles. Ce choix n'est pas nécessairement demandé au tout premier `git pull` : les précédentes réceptions peuvent avoir fonctionné sans divergence.

Pour notre rapport, Samir a enregistré sa correction et ne possède plus de modification non validée (c'est-à-dire non enregistrée dans un commit). Sa tentative de réception rencontre la contribution divergente publiée par Camille.

```
$ git pull
```

Dans l'état actuel de nos connaissances et pour l'utilisation de Git présentée dans ce cours, le comportement retenu est la fusion. Ce choix conserve les contributions enregistrées et permet de les réunir dans un commit de fusion. Le réglage suivant définit ce comportement par défaut dans le dépôt courant.

```
$ git config pull.rebase false
```

Ce réglage sélectionne la fusion comme mode d'intégration. Cette commande de configuration ne modifie ni les fichiers ni les commits existants. Sans option `--global`, sa portée est limitée au dépôt courant et le choix est conservé pour les réceptions suivantes.

Pour notre rapport, Samir effectue cette configuration dans son dépôt, puis relance la réception. Si la fusion était déjà configurée, Git pouvait directement tenter l'intégration dès la première commande.

```
$ git pull
```

5.2.2 Repérer l'interruption de l'intégration

La réception des données distantes peut réussir alors que leur intégration reste inachevée. Le dépôt local conserve alors les commits reçus, mais aucun commit de fusion n'enregistre encore le résultat combiné. Git indique les fichiers concernés et laisse l'opération de fusion en cours.

Pour notre rapport, l'intégration signale un conflit de contenu dans `main.tex`. La commande `git status` permet de consulter l'état de la fusion ; l'affichage court identifie le fichier non fusionné.

```
$ git status --short
UU main.tex
```

Les deux lettres `U` indiquent ici un fichier en conflit que les deux contributions ont modifié. Elles ne signifient pas que les commits de Camille ou de Samir ont été perdus. À ce stade, une nouvelle réception (`git pull`) ou une publication (`git push`) ne remplace pas le travail de résolution. L'opération en cours doit d'abord être traitée.

5.2.3 Lire les marqueurs de conflit

Pour un conflit de contenu dans un fichier texte, Git insère habituellement des marqueurs qui délimitent les deux propositions.

Le listing 5.1 présente le passage concerné dans la copie de Samir. Le libellé `version-recue` remplace ici l'identifiant que Git affiche après le dernier marqueur ; cet identifiant dépend des commits réellement créés.

Listing 5.1 – Passage en conflit dans le fichier `main.tex` de Samir.

```
<<<<<<< HEAD
La valeur mesurée est 15.
=======
La valeur mesurée est 14.
>>>>>>> version-recue
```

La ligne `<<<<<<< HEAD` ouvre la proposition provenant de la version locale en cours d'intégration. La ligne `=======`, composée de sept signes égaux, sépare les deux propositions. La ligne commençant par `>>>>>>>` termine la proposition reçue. Le libellé `HEAD` désigne ici le commit local à partir duquel la fusion est effectuée ; il ne fait pas partie du contenu à conserver.

Dans notre rapport, la première partie contient donc la valeur 15 enregistrée par Samir, et la seconde la valeur 14 reçue avec la contribution de Camille. Cette lecture dépend du dépôt dans lequel la fusion est effectuée. Les mentions « local » et « reçu » n'indiquent ni quelle proposition est correcte ni quel auteur doit avoir le dernier mot. Le reste du fichier, notamment le paragraphe sur les trois mesures, reste présent en dehors du passage en conflit.

5.2.4 Construire le texte final

Résoudre un conflit consiste à remplacer le passage délimité par les marqueurs par le contenu voulu. Le résultat peut reprendre l'une des propositions, combiner des éléments des deux ou formuler un nouveau texte. La décision repose sur le sens du document et sur les échanges entre les auteurs. Elle ne se limite pas à supprimer les lignes de marqueurs en laissant deux phrases contradictoires.

Pour notre rapport, Camille et Samir vérifient ensemble la valeur à retenir. Dans le scénario retenu, cette vérification les conduit à conserver 14. Samir remplace donc tout le passage en conflit par la ligne suivante dans son éditeur.

```
La valeur mesurée est 14.
```

Le fichier ne contient plus les marqueurs `<<<<<<<`, `=======` et `>>>>>>>`. Le paragraphe « La mesure a été répétée trois fois. » reste présent. Cette édition modifie le répertoire de travail de Samir ; elle ne termine pas à elle seule la fusion et ne crée pas de commit.

5.2.5 Vérifier le document

La vérification porte sur le contenu retenu et sur sa cohérence avec le reste du projet. Une compilation réussie ne prouve pas qu'une résolution est correcte : un document peut compiler tout en contenant une valeur erronée ou même certains marqueurs oubliés. La relecture du passage et du contexte reste donc nécessaire.

Pour notre rapport, Samir vérifie la valeur 14, la conservation du paragraphe et l'absence de marqueurs dans le fichier. La compilation permet ensuite de contrôler que le document LaTeX reste exploitable.

```
$ pdflatex main.tex
```

Le résultat attendu est un rapport comportant les résultats convenus par les auteurs et une mise en page correcte. Les fichiers produits par la compilation restent hors du dépôt, conformément à la section 3.4.1. La sélection suivante doit porter sur le fichier source résolu, sans ajouter le PDF ni les fichiers auxiliaires.

5.2.6 Indexer la résolution et finaliser la fusion

L'indexation d'un fichier résolu (`git add`) indique à Git le contenu retenu pour ce fichier. Elle fait disparaître son état non fusionné, mais ne vérifie pas à la place de l'auteur le sens du texte ni l'absence de marqueurs. Un ajout effectué trop tôt pourrait donc préparer un résultat incorrect. La vérification précède l'indexation, puis un contrôle de l'état du dépôt permet de savoir si des conflits restent à traiter.

Pour notre rapport, Samir indexe la version vérifiée de `main.tex` et consulte l'état du dépôt.

```
$ git add main.tex
$ git status
```

Git indique que les conflits sont résolus mais que la fusion est encore en cours. Dans un projet comportant plusieurs fichiers en conflit, chacun doit recevoir une résolution et être indexé avant la finalisation. La création du commit de fusion (`git commit`) enregistre ensuite l'instantané préparé et termine l'opération.

```
$ git commit -m "Fusion des corrections de la valeur mesurée"
$ git status --short
```

Le commit créé a deux parents : le commit local de Samir et celui de Camille qui vient d'être intégré. Il conserve ainsi le lien avec les deux contributions, même si le texte final reprend la valeur de Camille. L'affichage court est vide si aucune autre modification n'est en attente. La

fusion est terminée localement, mais son résultat n'est pas encore publié.

5.2.7 Publier puis recevoir la version résolue

La publication du commit de fusion (`git push`) rend la résolution accessible dans le dépôt partagé. Elle suppose que les droits d'écriture sont disponibles et que le dépôt distant n'a pas reçu entre-temps une contribution qui empêcherait cette mise à jour. Si une nouvelle contribution a été publiée, elle doit être intégrée et vérifiée avant une nouvelle tentative.

Pour notre rapport, aucune autre publication n'intervient pendant la résolution. Samir peut donc publier son commit de fusion depuis `rapport-samir`.

```
$ git push
```

Camille n'a pas modifié sa copie depuis sa publication. La réception de la version résolue se fait donc par avance rapide depuis `rapport-camille`.

```
$ git pull
```

Les deux auteurs disposent alors du même commit de fusion et du même texte enregistré. La correction locale, son indexation, sa validation et sa publication constituent bien des étapes différentes. Le tableau 5.1 récapitule leurs effets dans le dépôt de Samir et sur le serveur.

Étape	Outil ou commande	Effet
Choix de la fusion, si nécessaire	`git config pull.rebase false`	Comportement des réceptions défini pour le dépôt.
Réception et tentative d'intégration	`git pull`	Conflit signalé ; fusion en cours.
Correction et relecture	Éditeur de texte	Contenu résolu dans le répertoire de travail.
Vérification de compilation	`pdflatex main.tex`	Vérification technique du document.
Indexation de la résolution	`git add main.tex`	Contenu retenu préparé dans l'index.
Validation de la fusion	`git commit`	Commit de fusion créé localement.
Publication du résultat	`git push`	Commit de fusion disponible sur le serveur.

TABLE 5.1 – Étapes du traitement du conflit rencontré par Samir sur `main.tex`.

5.3 Coordonner les contributions

Des contributions courtes et cohérentes facilitent la compréhension d'une divergence. Leur enregistrement séparé (`git add`, puis `git commit`) aide à identifier l'intention de chaque auteur. La réception régulière des contributions publiées (`git pull`) et leur publication fréquente une fois validées (`git push`) limitent le temps pendant lequel les copies évoluent sans échange. Ces pratiques ne garantissent pas l'absence de conflit, mais réduisent la quantité de changements à examiner ensemble.

La répartition des tâches et la communication complètent ces opérations techniques.

Pour notre rapport, les auteurs peuvent convenir qu'une personne révise les résultats pendant que l'autre complète la méthode. Lorsque deux corrections doivent porter sur la même valeur, une discussion préalable permet de déterminer l'intention commune avant de modifier le texte.

Une fusion automatique ne dispense pas de vérifier la cohérence du document obtenu.

> Pour notre rapport, un changement de valeur dans les résultats peut nécessiter une correction de la conclusion, même si les modifications concernent des passages différents et ne déclenchent aucun conflit. La compilation et la relecture restent donc utiles après une intégration réussie.

Le conflit signale une décision de contenu encore nécessaire ; il n'est pas, à lui seul, la preuve qu'une contribution a été perdue. Git conserve les versions enregistrées et permet de relier leur intégration aux commits dont elles proviennent. Le suivi des tâches et des décisions de l'équipe peut, en complément, être organisé dans une forge au moyen de tickets.

Chapitre 6

GitLab et la gestion de projet

Git conserve les versions enregistrées et permet de réunir les contributions de plusieurs auteurs. L'équipe doit également savoir quelles tâches restent à réaliser, qui les prend en charge et comment vérifier leur résultat. La forge complète la gestion de versions en donnant un espace commun à cette organisation.

> Pour notre rapport en LaTeX, Camille et Samir ont appris à enregistrer, publier et intégrer leurs contributions. Ils disposent désormais d'un moyen de partager le texte, mais les commandes Git ne décrivent pas à elles seules le travail attendu. Ce chapitre présente le rôle de GitLab dans le suivi de ce travail, en reliant une tâche, un ticket et les commits qui la réalisent.

6.1 Les services d'une forge

6.1.1 Hébergement, utilisateurs et permissions

GitLab est une forge qui rassemble l'hébergement de dépôts Git et des services de collaboration autour de projets. Le *projet GitLab* comprend notamment le dépôt, les tickets et les réglages d'accès. Ces éléments ne sont pas tous des fichiers du dépôt principal. Un clonage (`git clone`) récupère le dépôt Git ; il ne copie pas les tickets et les discussions de la forge dans le répertoire de travail.

L'accès aux services dépend des autorisations attribuées à l'utilisateur. Les rôles définissent les actions permises dans le projet, comme la consultation du contenu, la participation au suivi des tâches ou la publication de commits. Disposer d'un compte sur une instance GitLab ne donne pas automatiquement accès à tous ses projets. De même, pouvoir cloner un dépôt ne garantit pas le droit d'y publier. Des règles propres à une branche peuvent également limiter les publications autorisées.

> Pour notre rapport, Camille et Samir doivent pouvoir consulter le projet, participer aux tickets et publier leurs contributions sur la branche utilisée par l'équipe. Ces droits sont distincts de l'identité enregistrée par Git dans leurs commits. Le réglage `user.name` indique un nom d'auteur, mais ne donne aucune permission sur la forge. Les modalités d'accès effectives dépendent de l'instance et des règles du projet.

6.1.2 Documentation partagée et wiki

Un *wiki* est un espace de pages destiné à organiser une documentation partagée. Il peut décrire des conventions, des choix communs ou des informations utiles à la prise en main du projet. Il complète les fichiers versionnés, sans les remplacer. Les sources nécessaires à la production du livrable doivent rester disponibles dans son dépôt.

Pour notre rapport, le wiki peut présenter la répartition des rôles et les conventions de relecture adoptées par Camille et Samir. Le texte LaTeX et les illustrations restent dans le dépôt du rapport, afin qu'une version du document puisse être retrouvée à partir de ses sources. Le wiki décrit ainsi l'organisation du travail, tandis que le dépôt conserve les états du livrable.

6.1.3 Intégration et déploiement continus

L'*intégration continue*, ou *CI* pour *continuous integration*, permet d'exécuter automatiquement des opérations de construction et de vérification lors de l'évolution d'un projet. Le *déploiement continu* prolonge cette automatisation jusqu'à la mise à disposition d'un résultat dans un environnement prévu à cet effet. GitLab peut organiser ces opérations en enchaînements de tâches appelés *pipelines*. Ces services supposent une configuration et des moyens d'exécution ; la seule présence d'un dépôt ne les active pas automatiquement.

Pour notre rapport, une automatisation pourrait compiler les sources après une publication (`git push`) et signaler une erreur de compilation à l'équipe. Le PDF produit pourrait être rendu disponible comme résultat de cette exécution, sans être ajouté au dépôt Git. Cette possibilité respecte la règle d'exclusion des fichiers compilés énoncée à la section 3.4.1.

La configuration de ces automatisations sera abordée dans la suite de votre formation.

6.2 Décrire et suivre une tâche avec un ticket

6.2.1 Le travail attendu et son résultat

Un *ticket*, appelé *issue* dans la documentation de GitLab, décrit un travail à effectuer ou un problème à traiter. Il peut concerner une nouvelle fonctionnalité, une correction ou une amélioration de la documentation. Il conserve la demande et les échanges qui aident à la comprendre. Le ticket se distingue donc du commit : il décrit le travail attendu, alors que le commit enregistre une étape de sa réalisation.

Un titre précis identifie le sujet du ticket. La description situe le problème, indique le comportement ou le contenu attendu et donne les informations nécessaires à sa vérification. Une demande comme « Corriger le document » reste trop générale pour permettre à un autre membre de l'équipe de savoir ce qui est attendu. La désignation du passage concerné et d'un résultat vérifiable rend la tâche plus facile à prendre en charge.

Pour poursuivre notre rapport, une courte conclusion a été ajoutée après la résolution du conflit. Elle indique encore la valeur 13, tandis que les résultats donnent désormais 14. Cette incohérence motive le ticket présenté au tableau 6.1. Le numéro 12 est choisi pour illustrer les références entre un ticket et les commits du même projet.

Élément	Contenu du ticket `#12`
Titre	Valeur incohérente dans la conclusion.
Problème	La conclusion de `main.tex` indique 13, alors que les résultats validés indiquent 14.
Résultat attendu	La conclusion reprend la valeur retenue dans les résultats, sans modifier le paragraphe sur les trois mesures.
Personne assignée	Samir.
Vérification prévue	Relecture de la conclusion et compilation du rapport.

Table 6.1 – Ticket de correction associé à notre rapport.

La description explique ici l'écart à traiter sans imposer une longue discussion dans le message du futur commit. Les précisions éventuelles peuvent être ajoutées aux échanges du ticket. L'équipe conserve ainsi le contexte de la correction en un endroit identifiable.

6.2.2 Attribution, échanges et clôture

L'assignation d'un ticket désigne la personne chargée de le prendre en charge. Elle permet de répartir les tâches, mais ne réalise pas les modifications et ne donne pas de permissions supplémentaires sur le dépôt. Les commentaires servent à demander une précision, à discuter d'un choix ou à rendre compte d'une vérification. Le suivi peut être complété par des étiquettes ou une échéance lorsque le projet en a besoin.

Un ticket reste ouvert tant que le travail attendu n'est pas considéré comme terminé. Sa clôture traduit une décision de suivi : le résultat répond à la demande et les vérifications prévues ont été effectuées. Elle ne remplace ni l'enregistrement d'un changement (`git commit`) ni sa publication (`git push`). Le détail des écrans et des actions disponibles dépend de la version de GitLab et des permissions attribuées.

> Pour notre rapport, Samir prend en charge le ticket `#12` et Camille participe à la relecture. Une éventuelle discussion sur la valeur à conserver peut être consignée dans les commentaires du ticket. La tâche est considérée comme terminée lorsque la conclusion corrigée est partagée et que les deux auteurs ont pu en vérifier la cohérence.

6.3 Relier les tickets et les commits

6.3.1 Une référence dans le message de commit

Dans un même projet GitLab, la notation `#` suivie du numéro d'un ticket permet d'y faire référence dans un message de commit. Lorsque le commit est publié dans le projet, GitLab peut relier le changement au ticket mentionné [1]. Le numéro est celui du ticket dans ce projet, et non un identifiant de commit.

> Pour notre rapport, la correction de la conclusion peut être enregistrée avec le message suivant. La séquence suppose que Samir a déjà corrigé et vérifié le fichier, et que les seules modifications indexées sont celles de cette tâche.
>
> ```
> $ git add main.tex
> $ git commit -m "Correction de la conclusion (#12)"
> $ git push
> ```
>
> La référence au ticket complète le message sans le remplacer. « Correction de la conclusion » indique l'intention de la modification, tandis que `#12` donne accès à la demande et à son contexte. Un message réduit au seul numéro de ticket apporterait moins d'information à la lecture de l'historique local (`git log`). Le message reste enregistré avec le commit, même lorsque la forge n'est pas accessible.

La mention `#12` établit un lien ; elle ne constitue pas, à elle seule, une instruction de clôture automatique du ticket. Dans le parcours retenu ici, la clôture est effectuée dans GitLab après vérification du résultat. Les mécanismes de clôture automatique ne sont pas nécessaires à cette première utilisation.

6.3.2 Une convention pour la traçabilité collective

Pour le travail collectif présenté dans ce cours, les contributions sont rattachées aux tickets qui décrivent les tâches correspondantes. Les messages des commits liés à une tâche mentionnent donc son ticket. Cette convention facilite la compréhension des choix et le suivi de leur réalisation ; Git n'impose pas lui-même l'existence d'un ticket pour créer un commit.

Une tâche peut nécessiter plusieurs commits cohérents. Le ticket conserve alors une demande commune, tandis que les commits rendent visibles les étapes de sa réalisation. Réciproquement,

1. Documentation GitLab, « Crosslinking issues », section « From commit messages ».

regrouper plusieurs tâches sans rapport dans un même commit rend plus difficile la lecture de leurs effets respectifs. La sélection des modifications dans l'index reste donc importante, même lorsqu'un ticket est associé au travail.

> Pour notre rapport, le ticket `#12` permet de retrouver pourquoi la conclusion devait changer. Le commit permet de retrouver la ligne effectivement corrigée. Les commentaires peuvent conserver la vérification faite par Camille, et la clôture indique que la demande est satisfaite. La traçabilité résulte de l'articulation de ces informations, pas de leur répétition dans tous les outils.

6.4 Du ticket au document partagé

6.4.1 Réalisation et vérification de la correction

Le traitement d'une tâche associe l'organisation dans la forge et les opérations sur les fichiers.

> Pour notre rapport, Samir reçoit d'abord la version partagée (`git pull`) dans un répertoire de travail sans modification en attente. Il consulte le ticket `#12`, puis vérifie que les résultats indiquent 14 et que la conclusion indique encore 13. Le passage à corriger est présenté au listing 6.1.
>
> Listing 6.1 – Conclusion de notre rapport avant la correction du ticket `#12`.
>
> ```
> \section{Conclusion}
> La valeur retenue est 13.
> ```
>
> Samir remplace la valeur 13 par 14 dans la conclusion, sans toucher au reste du texte. La lecture des différences (`git diff`) permet de vérifier que seule cette ligne a changé. La compilation (`pdflatex main.tex`) et la relecture du PDF permettent de contrôler le résultat obtenu. L'indexation (`git add main.tex`), la validation (`git commit`) avec une référence à `#12`, puis la publication (`git push`) partagent la correction sous la forme d'un commit identifiable. Camille peut ensuite recevoir cette contribution (`git pull`) et vérifier la cohérence entre les résultats et la conclusion. Le ticket peut conserver un commentaire indiquant cette vérification, puis être clôturé. Le PDF compilé reste un résultat de fabrication du rapport, sans être ajouté au dépôt.

6.4.2 Les responsabilités de chaque outil

Le tableau 6.2 distingue les informations utiles au suivi du travail. Il permet de comprendre pourquoi une tâche peut être décrite dans GitLab, enregistrée localement dans Git, puis vérifiée à partir du document produit.

Question	Information correspondante
Quel travail est attendu ?	Titre et description du ticket dans GitLab.
Qui le prend en charge ?	Assignation et échanges du ticket.
Quel contenu est préparé ?	Index du dépôt local, observable avec `git diff --cached`.
Quel changement est enregistré ?	Commit et message associé, consultables dans l'historique Git.
Le changement est-il partagé ?	Publication du commit (`git push`) sur le dépôt partagé.
La demande est-elle satisfaite ?	Vérification du résultat, commentaire et clôture du ticket.

Table 6.2 – Articulation entre suivi de tâche, gestion de versions et vérification du livrable.

Une fermeture de ticket n'envoie aucun fichier aux autres auteurs, et un commit local n'informe pas automatiquement toute l'équipe de l'avancement. Le partage des versions et le suivi des tâches doivent donc rester cohérents.

> Pour notre rapport, cela signifie que la correction annoncée comme terminée est aussi publiée, accessible à Camille et vérifiée dans le document final.

Conclusion

Ce polycopié donne aux membres d'une équipe projet les bases nécessaires pour développer leur premier projet intégratif (PI) de manière collaborative et rédiger collégialement le rapport de ce même projet. La gestion de versions s'applique aussi bien aux sources du logiciel qu'à celles du document qui en présente les objectifs, la réalisation et les résultats. Elle permet de conserver des étapes identifiables, de partager les contributions et de réunir les modifications des différents membres de l'équipe.

Le travail collectif repose sur l'articulation entre les outils et l'organisation du projet. Git conserve l'historique et permet les échanges entre dépôts, tandis que GitLab accompagne la répartition des tâches et la traçabilité des décisions au moyen des tickets. Des commits cohérents, des messages compréhensibles, une synchronisation régulière et la vérification des résultats contribuent à rendre le travail de chacun accessible à l'ensemble de l'équipe.

Ce document reste une introduction à Git. Les opérations présentées permettent de commencer à travailler ensemble, mais ne couvrent pas toutes les possibilités du logiciel. Des fonctionnalités plus avancées, notamment la gestion des branches et l'organisation de développements parallèles, seront enseignées dans la suite de votre formation.

Les étudiants qui souhaitent progresser d'ici là peuvent poursuivre leur apprentissage avec le livre *Pro Git*, de Scott Chacon et Ben Straub. Cet ouvrage collaboratif est disponible gratuitement en ligne, ses sources sont publiques et il est traduit dans de nombreuses langues. Il est publié sous licence Creative Commons BY-NC-SA 3.0. La version française est accessible à l'adresse `https://git-scm.com/book/fr/v2`.