1. Git 개요와 설치

버전 관리의 필요성과 종류 (로컬 vs 분산)

✅ 1. 버전 관리란?

> **버전 관리 시스템(Version Control System, VCS)** 이란
> 소스 코드 또는 문서의 **변경 이력**을 **시간의 흐름에 따라 기록**하고,
> 필요할 경우 **과거 버전으로 되돌릴 수 있게 해주는 시스템**이다.

✅ 2. 왜 버전 관리가 필요한가?

필요성	설명
🔁 **변경 이력 추적**	언제, 누가, 어떤 내용을 바꿨는지 기록
🪃 **되돌리기 (Rollback)**	실수나 오류 발생 시 안전하게 이전 상태로 복원
👥 **협업 지원**	여러 개발자가 동시에 작업하고 병합 가능
🧪 **실험적 브랜치 관리**	새로운 기능을 테스트한 뒤 통합 여부 결정 가능
📦 **릴리즈 및 배포 이력 관리**	버전별 태그 관리 및 배포 추적
🔒 **감사 및 보안**	보안 이슈 추적, 책임 소명 가능
☁️ **원격 백업**	서버와 클라우드 연동으로 소스 유실 방지

✅ 3. 버전 관리의 방식 분류

버전 관리 시스템은 다음 3가지 방식으로 나뉘어:

3.1 🔸 로컬 버전 관리(Local VCS)

- **개발자의 컴퓨터**에만 변경 이력을 저장하는 방식
- 모든 작업이 **로컬 파일 시스템**에서만 일어남

```
# 예시 도구: RCS (Revision Control System)
```

📉 단점

- 백업이 없음 → 하드디스크 고장 시 데이터 손실
- 협업 불가 (로컬만 있음)
- 히스토리 공유 어려움

3.2 🔸 중앙 집중형 버전 관리(Centralized VCS)

- 하나의 **중앙 서버**가 전체 변경 이력을 저장
- 클라이언트들은 중앙 서버에 접속하여 `checkout`/`commit`함

```
# 대표 시스템: CVS, Subversion(SVN), Perforce 등
```

📈 장점

- 이력 관리 중앙화
- 프로젝트 진행 상황을 서버에서 파악 가능
- 권한 제어가 비교적 간단

📉 단점

- 서버 장애 시 **모든 작업 중단**
- 커밋/로그 확인 등도 **항상 네트워크 연결 필요**
- 오프라인 작업 불가능

3.3 🔸 분산 버전 관리(Distributed VCS)

- 모든 클라이언트가 **전체 히스토리와 코드의 사본**을 가짐
- 중앙 서버가 있더라도, **필수는 아님**
- 각자의 로컬에서도 `commit`, `log`, `branch` 등 사용 가능

```
# 대표 시스템: Git, Mercurial(Hg), Bazaar
```

📈 장점

- **오프라인에서도 모든 작업 가능**
- 서버가 없어도 **복구 가능** (로컬에 전체 이력이 있음)
- **분기(branch)** 와 병합(merge)에 매우 유리
- **속도 빠름** (로컬에서 대부분 수행)

📉 단점

- 처음에는 **개념이 복잡**
- 각자 로컬에 commit을 하므로 **동기화 타이밍 조절 필요**

✅ 4. Git은 왜 분산형을 채택했을까?

Git은 다음의 이유로 **분산형 구조**를 채택했어:

- 리누스 토발즈가 만든 리눅스 커널은 **전 세계 수천명이 협업**해야 했고,
- 각자 로컬에서 기능 개발 → 나중에 통합이 필요했기 때문.

- 커밋, 브랜치, 히스토리 확인 등 거의 모든 작업을 **로컬에서 빠르게 처리**할 수 있게 설계되었음.
- Git은 **파일이 아닌 스냅샷**을 저장하며, **SHA-1 해시** 기반으로 무결성을 보장함.

✅ 5. 요약 비교표

항목	로컬 VCS	중앙 집중형 VCS	분산형 VCS
저장 위치	사용자 PC만	중앙 서버	사용자 + 서버
협업 가능	❌	✅	✅
서버 장애 대응	❌	작업 중단	로컬에서 계속 가능
성능 (속도)	빠름	느림 (서버 의존)	매우 빠름
오프라인 사용	✅	❌	✅
복구 가능성	낮음	낮음	높음
대표 도구	RCS	SVN	Git

✅ 6. 결론

- 개인 프로젝트: **Git + GitHub** 형태가 가장 유연함
- 팀 협업: 분산형이지만 **원격 협업을 위한 중앙 서버 구조(GitHub, GitLab 등)** 도 함께 사용
- 현대 개발 환경에서는 **Git 기반의 분산 버전 관리 시스템이 거의 표준**

Git이란? Git의 철학 (작은 단위, 분산 구조)

✅ 1. Git의 정의

> **Git은** 분산형 버전 관리 시스템(DVCS: *Distributed Version Control System*)으로,
> 텍스트 중심의 소스코드 변경 이력을 **효율적이고 안전하게 관리**하기 위해 만들어진 도구이다.

Git은 2005년, 리누스 토발즈가 리눅스 커널 개발을 위해 처음 만들었고,
지금은 오픈소스 개발의 **사실상 표준 도구**로 자리잡았다.

✅ 2. Git의 탄생 배경

- 2005년 이전까지 리눅스 커널은 **BitKeeper**라는 상용 VCS를 사용했음
- BitKeeper가 라이선스 문제로 **오픈소스 커뮤니티에 제공 중단**
- 커뮤니티 전체가 혼란에 빠지자, 리누스 토발즈가 직접 Git을 개발함
- 당시 설계 목표는 다음과 같았음:

설계 목표	설명
속도	매우 빠른 로컬 작업

설계 목표	설명
단순함	복잡한 구조 없이 CLI 기반
완전한 분산	각 개발자 PC에 전체 이력 보관
무결성	데이터 위조 불가능 (SHA-1 해시 기반)
브랜칭과 병합	가볍고 빠르고 안전하게 지원

✅ 3. Git의 철학

Git은 단순한 툴이 아니라, **아주 명확한 철학과 원칙**에 따라 설계되었어. 아래는 Git 철학의 핵심 요소야:

🔹 3.1 스냅샷(Snapshot) 기반 저장

- Git은 파일의 변경 내용을 저장하는 것이 아니라,
  프로젝트 전체의 스냅샷(전체 상태)을 저장한다.
- 매번 `commit` 할 때마다 **전체 트리의 정지 이미지(snapshot)** 를 저장함.
- 실제로는 **변경되지 않은 파일은 이전 버전의 블롭(blob)을 재사용**함 → 매우 효율적.

> 📌 기존 VCS: 변경 사항(diff) 저장
> 📌 Git: 전체 스냅샷 + 중복 제거 방식

🔹 3.2 로컬 중심 설계

- Git의 모든 명령은 **로컬에서 수행**됨
  → commit, branch, log, diff 등은 서버 없이 작동
- 중앙 서버 없이도 **개별 개발자가 독립적**으로 작업 가능
- 이는 오픈소스, 오프라인 개발, 다양한 브랜치 실험 등에 매우 유리함

🔹 3.3 분산 구조

- Git은 **모든 사용자가 전체 히스토리와 데이터의 복제본을 보유**
- 서버가 죽어도 복구 가능 (분산 백업 구조)
- 서버는 **협업과 공유의 수단**일 뿐, 필수가 아님

🔹 3.4 변경 불가한 이력 (Immutable History)

- 커밋은 SHA-1 해시로 식별됨 (내용 기반 해시)
- 누군가가 과거 커밋을 **변조하거나 위조하는 것이 매우 어렵다**
- 히스토리의 **신뢰성과 추적성**을 확보

🔹 3.5 브랜치와 병합의 자유

- 브랜치 생성이 **매우 가볍고 빠름**
  - 그냥 커밋 ID에 포인터 하나 추가하는 수준
- 병합(merge)이나 리베이스(rebase)도 효율적
- 실험, 롤백, 기능 개발 등에 **브랜치를 적극 활용**하는 것을 권장

> 기존: 브랜치는 무겁고 관리 어려움
> Git: 브랜치는 기본 작업 단위로 장려됨

🔹 3.6 커밋 중심의 사고

- Git은 **파일 중심이 아니라 커밋 중심**
- 커밋이 프로젝트의 상태를 결정함
- 커밋을 기반으로 cherry-pick, rebase, revert, bisect 등 강력한 기능을 수행

🔹 3.7 명시적, 고의적, 논리적 작업 흐름

- Git은 **자동이 아니라 명시적인 작업**을 선호함
  - `add` → `commit` → `push`
- 변경 이력을 **의도적으로 구조화**하고 관리
- 히스토리를 **아름답고 논리적**으로 유지하려는 철학이 반영됨

✅ 4. Git의 철학이 실무에 주는 교훈

Git 철학	실무 적용 전략
분산 구조	로컬에서 실험하고, 원격은 안정적 브랜치만 공유
브랜치 자유	`feature/`, `bugfix/`, `hotfix/` 등 브랜치 전략 사용
변경 불가	신중한 커밋, 커밋 메시지 규칙 유지
로컬 중심	오프라인에서도 문제없이 작업 가능
히스토리 강조	커밋 단위를 작게 유지 (`작은 단위, 자주 커밋`)

✅ 5. 한 문장으로 요약

> **Git은 "빠르고, 안전하고, 분산된 시스템"을 목표로 만들어졌으며, 모든 작업의 중심을 커밋과 스냅샷에 두고 있다.**

Git vs SVN

> **Git = 분산 버전 관리(DVCS)**
> **SVN = 중앙 집중형 버전 관리(CVCS)**

✅ 1. 기본 구조 비교

항목	Git	SVN
구조	**분산형 (Distributed)**	**중앙 집중형 (Centralized)**
저장소 위치	로컬 + 서버 (모두 완전한 저장소)	**중앙 서버에 저장**, 로컬에는 복사본만
독립 작업 가능	**완전 가능** (오프라인 커밋 등)	**불가능**, 서버 접속 필요
속도	빠름 (로컬에서 처리됨)	느림 (서버 왕복 필수)
복구 가능성	로컬 저장소로 복구 가능	서버가 날아가면 전체 손실

✅ 2. 브랜치 관리

항목	Git	SVN
브랜치 생성	**가볍고 빠름** (`git branch`)	**디렉토리 복사 수준** (`branches/` 디렉토리 구조)
브랜치 병합	매우 유연하고 빠름	복잡하고 충돌 위험 높음
브랜치 사용 권장	적극 사용 (기능 개발 단위)	잘 안 씀 (무거움)

✅ 3. 이력 및 커밋

항목	Git	SVN
커밋 단위	**스냅샷 기반** (전체 프로젝트 상태)	**변경점 기반** (diff 중심)
커밋 로그 조작	가능 (`rebase`, `reset`, `reflog`)	불가 (되돌리기 불편)
히스토리 관리	매우 강력함 (분기, 병합, 복구)	단순한 시간 순서 로그

✅ 4. 협업과 병합

항목	Git	SVN
협업 방식	로컬에서 개발 → 원격에 푸시	서버에 직접 커밋
충돌 관리	유연한 병합 도구 지원	충돌이 자주 발생
Pull Request	GitHub, GitLab 등에서 지원	없음 (수동으로 관리)

✅ 5. 오프라인 작업

항목	Git	SVN
커밋, 로그 보기	**완전 가능 (로컬 히스토리 존재)**	**불가 (서버 필요)**
브랜치 이동	로컬에서 자유롭게 가능	서버 연결 필요

✅ 6. 사용 도구와 생태계

항목	Git	SVN
주요 도구	Git CLI, GitHub, GitLab, Bitbucket	TortoiseSVN, SVN CLI
대중성	**업계 표준** (오픈소스, 스타트업, 기업)	유지보수된 구형 시스템
확장성	Git Hooks, GitHub Actions 등 풍부함	제한적

✅ 7. 요약 정리표

비교 항목	Git	SVN
버전 관리 방식	**분산형(DVCS)**	**중앙 집중형(CVCS)**
속도	빠름 (로컬 작업)	느림 (서버 의존)
브랜치	가볍고 자유로움	무겁고 복잡
협업	로컬에서 작업 후 공유	서버 중심 개발
신뢰성	로컬 복구 가능	서버 장애 시 전체 영향
오프라인 작업	가능	불가
학습 곡선	다소 높음	낮음 (GUI 도구 사용 용이)
대표 사용처	오픈소스, 웹/앱 개발, 스타트업, DevOps	오래된 기업 시스템, 문서 저장소 등

✅ 8. Git이 SVN보다 압도적인 이유?

- 오프라인 작업 가능 (여행 중에도 개발 가능)
- 브랜치 기반 워크플로우 장려 (기능 개발 단위 관리)
- 병합과 되돌리기, 실험적 작업에 매우 강함
- GitHub/GitLab 기반의 **Pull Request + 코드리뷰 체계** 확립 가능
- **CI/CD 및 DevOps 파이프라인과의 자연스러운 연동**

> 💬 Git은 단순한 도구가 아니라, **개발 문화(브랜치 전략, 협업 방식)를 바꾸는 시스템**이야.

Git 설치 및 설정 (`git config`)

> Git은 설치 후 반드시 사용자 정보를 설정해야 한다.
> 이 정보는 **커밋의 서명**이 되며, 협업에서 누가 작업했는지 추적할 수 있게 해준다.

✅ 1. Git 설치

🔹 1.1 운영체제별 설치

▪ Windows

- 공식 홈페이지: https://git-scm.com
- 설치 시 Git Bash 포함됨 (권장 사용)
- Git GUI, VSCode 연동 등도 포함 가능

▪ macOS

- Homebrew 사용:

```
brew install git
```

▪ Linux

- Debian/Ubuntu:

```
sudo apt update
sudo apt install git
```

- RHEL/CentOS:

```
sudo yum install git
```

✅ 2. 사용자 정보 설정 (`git config`)

Git은 커밋마다 누가 작성했는지를 남기기 위해
사용자 이름과 **이메일 주소**를 반드시 설정해야 해.

```
git config --global user.name "오정석"
git config --global user.email "oh@example.com"
```

🔹 옵션 설명

- `--global` : **전체 사용자 계정에 적용**
- `--local` : **현재 Git 프로젝트에만 적용**
- `--system` : 시스템 전체에 적용 (잘 안 씀)

```
git config --local user.name "프로젝트별 이름"
```

✅ 3. 설정 확인

```
git config --list
```

결과 예시:

```
user.name=오정석
user.email=oh@example.com
core.editor=vim
color.ui=auto
```

✅ 4. 주요 설정 항목들

항목	명령어	설명
사용자 이름	`user.name`	커밋 작성자 이름
이메일 주소	`user.email`	커밋 작성자 이메일
기본 에디터	`core.editor`	커밋 메시지 입력용 (`vim`, `nano`, `code`)
자동 색상	`color.ui`	색상 출력 여부 (`auto`, `true`, `false`)
줄바꿈	`core.autocrlf`	윈도우/유닉스 줄바꿈 문제 해결
병합 도구	`merge.tool`	병합 시 사용하는 GUI 도구 설정
기본 브랜치	`init.defaultBranch`	`master` 대신 `main` 등 원하는 이름 지정

```
git config --global core.editor "code --wait"
git config --global color.ui auto
git config --global init.defaultBranch main
```

✅ 5. Git 설정 파일 구조

우선순위	위치	파일
1	프로젝트 설정	`.git/config`
2	사용자 설정	`~/.gitconfig`
3	시스템 설정	`/etc/gitconfig`

```
cat ~/.gitconfig
cat .git/config
```

✅ 6. VSCode와 Git 연동 팁

- `Git`은 VSCode에 기본 내장되어 있음
- `"user.name"`과 `"user.email"` 설정만 제대로 해두면
  VSCode 내에서 커밋/푸시 등 모든 기능 사용 가능

✅ 7. 실수 복구: 설정 제거

```
# 전역 설정 제거
git config --global --unset user.name
git config --global --unset user.email
```

✅ 8. 초기 설정 체크리스트 ✅

- `user.name`, `user.email` 설정
- `core.editor` 설정 (CLI 환경일 경우)
- `init.defaultBranch` 설정 (`main`, `develop` 등)
- `.gitignore` 준비
- GUI 연동(VSCode, SourceTree 등)

✅ 마무리 요약

명령어	설명
`git config --global user.name "이름"`	사용자 이름 설정
`git config --global user.email "이메일"`	사용자 이메일 설정
`git config --list`	전체 설정 확인
`git config --edit --global`	설정 파일 직접 수정

• 사용자 정보 설정 (`user.name`, `user.email`)

✅ 1. 사용자 정보는 왜 필요한가?

- Git에서 커밋을 할 때마다 **작성자(author)** 정보가 커밋에 **영구히 기록**된다.
- 이 정보는 **협업 시 추적**에 사용되며, 누가 무엇을, 언제 바꿨는지를 확인할 수 있는 **디지털 서명**이다.
- GitHub, GitLab 등 원격 저장소에서 커밋한 사람을 **자동으로 연결**하기 위해 이메일 정보가 필수다.

예시 커밋 로그:

```
commit f8d9031d21a24cb2d7c...
Author: Oh Jeongseok <oh@example.com>
Date:   Thu May 30 19:41:21 2025 +0900

    README 파일 추가
```

✅ 2. 사용자 정보 설정 방법

🔹 전역 설정 (권장)

```
git config --global user.name "Oh Jeongseok"
git config --global user.email "oh@example.com"
```

- 이 설정은 사용자 컴퓨터의 모든 Git 프로젝트에 적용됨
- 설정 값은 `~/.gitconfig` 파일에 저장됨

🔹 로컬 설정 (특정 프로젝트 전용)

```
git config --local user.name "TeamA Bot"
git config --local user.email "bot@teama.io"
```

- 프로젝트 디렉토리의 `.git/config`에 저장됨
- 오픈소스 기여나 팀별 bot 사용자명 지정할 때 사용

✅ 3. 설정 값 확인

```
git config user.name
git config user.email
```

또는 전체 설정 보기:

```
git config --list
```

✅ 4. 설정 제거 및 변경

🔹 변경

```
git config --global user.name "새 이름"
git config --global user.email "새 이메일"
```

🔹 **제거**

```
git config --global --unset user.name
git config --global --unset user.email
```

✅ 5. 설정 파일 직접 수정

전역 설정:

```
vim ~/.gitconfig
```

```
[user]
  name = Oh Jeongseok
  email = oh@example.com
```

로컬 설정:

```
vim .git/config
```

✅ 6. GitHub Verified 뱃지를 위한 팁

- GitHub에 등록된 이메일 주소와 **일치하는 이메일**로 설정해야
  → 커밋에 **"Verified" 배지**가 붙는다.
- 보안을 위해 GitHub는 **no-reply 이메일**도 제공함:

```
git config --global user.email "12345678+username@users.noreply.github.com"
```

- 이건 GitHub > Settings > Emails 탭에서 확인 가능

✅ 7. 실무 팁: 여러 계정 사용할 때

회사용 vs 개인용 계정 분리

```
# 개인
git config --global user.name "Jeongseok"
git config --global user.email "me@gmail.com"

# 회사 프로젝트 디렉토리 내부에서:
git config --local user.name "Jeongseok Corp"
git config --local user.email "jeongseok@corp.com"
```

> 📌 `git config --local`이 `--global`보다 우선 적용된다.

✅ 8. 마무리 요약

항목	명령어	설명
사용자 이름 설정	`git config --global user.name "이름"`	커밋 작성자 이름
사용자 이메일 설정	`git config --global user.email "이메일"`	커밋 작성자 이메일
설정 확인	`git config --list`	전체 설정 보기
설정 제거	`--unset` 옵션 사용	개별 항목 제거
프로젝트별 설정	`--local` 사용	특정 프로젝트에만 적용됨

이제 커밋하면 항상 다음처럼 기록된다:

```
Author: Oh Jeongseok <oh@example.com>
```

• 에디터, 라인 끝 처리 등 기타 설정

> `git config`를 활용한 **환경 설정 최적화**

✅ 1. 기본 에디터 설정 (`core.editor`)

커밋 메시지 입력이나 `rebase` 편집 등에서 사용되는 기본 텍스트 에디터

🔹 CLI 환경 (터미널 작업 위주):

에디터	설정 예시
Vim	`git config --global core.editor "vim"`
Nano	`git config --global core.editor "nano"`

```
# Vim으로 설정
git config --global core.editor "vim"
```

🔹 GUI/IDE 연동:

에디터	설정 예시
VS Code	`git config --global core.editor "code --wait"`
Sublime Text	`git config --global core.editor "subl -n -w"`

```
# VSCode가 열리고 닫힐 때까지 기다림
git config --global core.editor "code --wait"
```

> `--wait` 옵션이 없으면 Git이 에디터가 닫히기 전에 종료돼 버림

✅ 2. 줄바꿈(라인 끝) 처리 설정 (`core.autocrlf`)

운영체제에 따라 **줄바꿈 방식(Line Ending)** 이 달라서 생기는 문제를 방지

플랫폼	줄바꿈 방식	설명
Windows	CRLF (`\r\n`)	캐리지 리턴 + 라인 피드
Linux/macOS	LF (`\n`)	라인 피드만 사용

🔹 설정 옵션

설정값	의미
`true`	체크아웃할 때 LF → CRLF 변환 (윈도우), 커밋 시 CRLF → LF
`input`	커밋할 때만 CRLF → LF, 체크아웃은 그대로 (리눅스/Mac 권장)
`false`	아무 것도 하지 않음 (전적으로 사용자가 책임짐)

🔹 예시

```
# Windows 권장 설정
git config --global core.autocrlf true

# Linux / macOS 권장 설정
git config --global core.autocrlf input
```

🔹 설정 확인

```
git config --get core.autocrlf
```

✅ 3. 출력 색상 설정 (`color.ui`)

Git 명령어 출력 결과에 색상 적용 (시각적으로 보기 좋음)

```
git config --global color.ui auto
```

설정값	설명
`auto`	터미널이면 색상 사용 (기본 추천)
`true`	항상 색상 사용
`false`	색상 끄기

✅ 4. 기본 브랜치명 설정 (`init.defaultBranch`)

Git 2.28 이상부터는 `git init` 시 기본 브랜치명을 직접 지정 가능
(기존에는 무조건 `master`, 요즘은 `main`으로 많이 바꿈)

```
git config --global init.defaultBranch main
```

이후 `git init` 할 때 자동으로 `main` 브랜치 생성됨

✅ 5. diff 도구 설정 (`diff.tool`, `merge.tool`)

GUI 도구나 IDE에서 병합/차이 비교를 할 때 어떤 도구를 쓸지 지정할 수 있어

```
# diff 도구로 meld 사용
git config --global diff.tool meld
git config --global merge.tool meld
```

> 기타 도구: kdiff3, Beyond Compare, vimdiff 등

✅ 6. 전역 `.gitignore` 파일 설정

개인 환경에 따라 무조건 무시하고 싶은 파일들 존재
(예: macOS의 `.DS_Store`, Windows의 `Thumbs.db`, IDE 설정 파일 등)

```
# 전역 gitignore 경로 지정
git config --global core.excludesfile ~/.gitignore_global
```

```
# ~/.gitignore_global 예시
.DS_Store
node_modules/
*.log
.idea/
```

✅ 7. 기타 유용한 설정 모음

설정	설명
`alias`	자주 쓰는 명령어 별칭 (예: `git co` = `git checkout`)
`push.default current`	현재 브랜치만 푸시하도록 설정
`pull.rebase true`	pull 시 자동으로 rebase 수행 (히스토리 정리 목적)
`credential.helper`	GitHub 로그인 정보 저장 (캐시/파일 방식)

```
git config --global alias.co checkout
git config --global push.default current
git config --global pull.rebase true
git config --global credential.helper store
```

✅ 마무리 요약

항목	설정 예시	목적
에디터	`core.editor`	커밋 메시지 입력 도구
줄바꿈	`core.autocrlf`	OS 간 줄바꿈 충돌 방지
색상	`color.ui`	터미널 가독성 향상
기본 브랜치	`init.defaultBranch`	`main` 사용 환경 대응
전역 무시	`core.excludesfile`	모든 프로젝트에서 무시할 파일 지정

Git GUI 도구 (SourceTree, GitKraken 등)

> Git을 시각적으로 다룰 수 있게 도와주는 GUI 클라이언트 도구들
> CLI가 익숙하지 않거나 협업 시 시각적 히스토리 관리가 필요한 개발자에게 유용

✅ 1. GUI 도구를 쓰는 이유

이유	설명
🔍 **브랜치 트리 시각화**	머지, 리베이스 등 복잡한 브랜치 구조 확인 용이
🧩 **충돌 관리 간소화**	병합 충돌 시 GUI 기반으로 선택 가능
💉 **Stage, Commit, Reset 직관화**	단계별 커밋이 GUI 버튼으로 처리됨
👥 **협업 브랜치 추적 용이**	원격 브랜치와 로컬 브랜치 비교 쉬움
📂 **파일 상태 보기**	어떤 파일이 변경됐는지 쉽게 확인 가능

✅ 2. 대표적인 Git GUI 도구

🔷 1. SourceTree (by Atlassian)

항목	설명
특징	무료, Windows/Mac 지원, Atlassian(GitHub 유사 서비스)에서 제공
기능	Git & Mercurial 지원, 복잡한 브랜치 시각화, Stash 관리, Submodule 지원
장점	시각적으로 가장 친숙한 Git 툴 중 하나, 브랜치/로그 관리 탁월

항목	설명
단점	무겁고 느릴 수 있음, 설치 시 Atlassian 계정 필요
홈페이지	https://www.sourcetreeapp.com/

🔷 2. GitKraken

항목	설명
특징	크로스 플랫폼(GitHub, GitLab, Bitbucket 등 통합 가능), 무료/유료 버전 있음
기능	시각적 브랜치 그래프, 내장 충돌 편집기, GitHub 연동, 팀 협업 기능
장점	디자인 깔끔, 직관적 UI, 히스토리 분석 탁월
단점	로그인 필요, 일부 기능은 유료
홈페이지	https://www.gitkraken.com/

🔷 3. GitHub Desktop

항목	설명
특징	GitHub 전용 GUI 클라이언트
기능	로컬 → GitHub 업로드, 커밋/푸시/풀, 풀 리퀘스트 작성
장점	깔끔한 UI, GitHub와 완벽 연동
단점	GitHub 이외의 플랫폼과는 연동이 제한됨
홈페이지	https://desktop.github.com/

🔷 4. SmartGit

항목	설명
특징	고급 기능 + GitFlow 통합 지원, 상용 툴(무료 사용 가능)
기능	서브모듈, 리베이스, 병합 도구, SVN 연동
장점	전문 개발자/팀에 적합, 커맨드라인과 GUI 동시 지원
단점	인터페이스 복잡, 학습 필요
홈페이지	https://www.syntevo.com/smartgit/

🔷 5. VS Code Git 패널

항목	설명
특징	VS Code에 내장된 Git 기능
기능	커밋, 브랜치 관리, 충돌 표시, GitHub 연동
장점	별도 설치 없이 사용 가능, GitLens 확장 플러그인으로 강화 가능
단점	고급 기능은 CLI 필요, 브랜치 트리 시각화는 약함
확장 예시	GitLens, Git Graph, GitHub Pull Requests and Issues

✅ 3. GUI 도구 선택 가이드

사용자 유형	추천 도구
Git 입문자	**GitHub Desktop**, **SourceTree**
시각적 브랜치 관리 중시	**GitKraken**, **SmartGit**
GitHub 연동 + VSCode 사용자	**GitLens + GitHub Pull Requests**
Git + Mercurial 동시에 관리	**SourceTree**, **SmartGit**
오픈소스 팀 협업	**GitKraken (Teams)** 또는 CLI 기반 + GitHub PR 관리

✅ 4. GUI 도구 vs CLI

항목	GUI 도구	CLI
학습 난이도	낮음 (드래그/드롭/클릭)	높음 (명령어 숙지 필요)
유연성	제한적 (기능 UI에 한정)	무한 (스크립트, hook 등 가능)
속도	느릴 수 있음	빠름 (로딩 없음)
디버깅	히스토리 시각화 우수	내부 구조 파악 우수

> 실무에서는 **GUI + CLI 혼용**을 권장해.
> 예를 들어 커밋은 GUI, 병합/리베이스는 CLI로 처리하는 식으로 말이야.

✅ 마무리 요약

도구	특징	추천 대상
SourceTree	직관적, 무료	Git 입문자, 팀 프로젝트

도구	특징	추천 대상
GitKraken	시각화 최고, 협업 기능 탁월	시니어 개발자, UI 중시
GitHub Desktop	GitHub 연동 최적화	GitHub 전용 사용 시
SmartGit	고급 기능 + 상용 툴	고급 사용자
VSCode Git	통합 환경	VS Code 사용자