실무 가이드 · GITHUB COLLABORATION

GitHub 협업 실무 가이드

AI 코딩 시대, 협업의 기준점

· 공동 레포 + 브랜치 + PR 기본 모델
· AI의 역할 — 작업 보조·diff 설명·PR 초안
· 사람의 역할 — 범위 결정·검토·승인·병합

KLIC 지음

KLIC BOOKS

차례

CHAPTER

01

GitHub협업 가이드

바이브코더가 Claude Code, Codex, Cursor 같은 AI 코딩 도구를 쓰더라도 협업의 기준점은 GitHub입니다. 이 문서는 초보자도 따라올 수 있도록 GitHub의 말을 일상어로 바꾸고, AI에게 그대로 붙여넣을 수 있는 프롬프트 중심으로 협업 절차를 정리합니다.

기본 모델 공동 레포 + 브랜치 + PR

AI의 역할 작업 보조, diff 설명, PR 초안

사람의 역할 범위 결정, 검토, 승인, 병합

협업 원칙 작게 나누고, 자주 공유하기

CHAPTER

02

1장 GitHub 말을 일상어로 번역하기

GitHub는 어렵다기보다 낯선 말이 많습니다. 아래 단어만 알아도 AI에게 일을 맡기고 팀원과 이야기하는 데 큰 문제가 없습니다.

repository, 레포

프로젝트 폴더가 GitHub에 올라간 것. 문서, 코드, 이미지, 설정 파일이 함께 들어 있습니다.

main branch

팀이 인정한 최종본 줄기입니다. 직접 고치지 않고 작업용 branch에서 바꾼 뒤 PR로 합칩니다.

branch

최종본을 망가뜨리지 않고 실험하는 작업 복사본입니다. 기능, 수정, 문서 작업마다 따로 만듭니다.

commit

작업 저장 지점입니다. "무엇을 왜 바꿨는지" 짧은 메시지를 붙입니다.

push

내 컴퓨터의 커밋을 GitHub에 올리는 일입니다. 올려야 팀원이 볼 수 있습니다.

pull

GitHub에 올라온 최신 변경을 내 컴퓨터로 내려받는 일입니다. 작업 전 습관처럼 합니다.

issue

해야 할 일, 버그, 질문을 적어두는 작업 카드입니다. "무엇을 할지" 합의하는 장소입니다.

pull request, PR

"이 변경을 main에 합쳐도 될까요?"라고 팀에 보여주는 검토 요청서입니다.

review

팀원이 PR을 보고 질문, 수정 요청, 승인을 남기는 과정입니다.

merge

검토가 끝난 PR을 main에 합치는 일입니다. 보통 담당자나 리뷰어가 누릅니다.

conflict

두 사람이 같은 파일의 같은 부분을 다르게 고쳐서 Git이 자동으로 합치지 못하는 상황입니다.

fork

남의 레포를 내 계정으로 복사하는 방식입니다. 팀 내부보다 외부 프로젝트 참여에 더 자주 씁니다.

CHAPTER

03

2장 팀 프로젝트의 기본 흐름

소규모 팀이나 KLIC 프로젝트에서는 한 레포에 팀원을 초대하고, 각자 branch를 만든 뒤 PR로 합치는 방식이 자연스럽습니다. GitHub 공식 문서도 이 방식을 작은 팀과 조직의 private project에 흔한 모델로 설명합니다.

GitHub 협업 흐름도 Issue로 할 일을 정의하고, Branch를 만들어 AI와 사람이 작게 수정한 뒤 Commit과 Push로 GitHub에 공유합니다. 그 다음 Pull Request로 검토를 요청하고, 리뷰를 거쳐 main에 병합합니다. Issue 할 일 정의 Branch 작업 공간 분리 AI + 사람 작게 수정 Commit + Push GitHub에 공유 Pull Request 검토 요청 Review + Merge 확인 후 합치기

한 줄 원칙 : main은 바로 고치지 않습니다. 작업마다 branch를 만들고, AI가 만든 변경도 PR에서 사람이 확인한 뒤 합칩니다.

CHAPTER

04

3장 처음 한 번만 하는 설정

GitHub 계정, Git, GitHub CLI가 준비되어야 합니다. 이 문서는 CLI 기준이므로 `gh` 명령어를 사용합니다.

GitHub CLI 로그인

이 단계는 직접 실행하세요. 브라우저 인증과 계정 권한이 걸려 있으므로 AI에게 토큰이나 인증 코드를 맡기지 않습니다.

Terminal 복사

```
gh auth login --web
gh auth status
```

Git 사용자 이름과 이메일 확인

Terminal 복사

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

값이 비어 있으면 아래처럼 설정합니다. 이메일은 GitHub 계정 이메일 또는 GitHub의 private noreply 이메일을 사용해도 됩니다.

Terminal 복사

```
git config --global user.name "내 GitHub 이름"
git config --global user.email "내이메일@example.com"
```

SSH vs HTTPS : 이 가이드는 `gh auth login --web`(HTTPS)를 기준으로 합니다. SSH 키를 이미 쓰고 있다면 그대로 SSH 주소로 clone해도 됩니다. 둘 중 하나만 쓰면 충분하므로 처음엔 HTTPS가 더 간단합니다.

AI에게 환경 점검시키기

Claude Code / Codex / Cursor에 붙여넣기 복사

```
이 컴퓨터에서 GitHub 협업을 할 준비가 되었는지 확인해줘.
다음 항목을 검사하고, 부족한 것이 있으면 설치/설정 방법을 초보자도 이해할 수 있게 안내해줘.

확인할 것:
1. git 설치 여부와 버전
2. gh(GitHub CLI) 설치 여부와 버전
3. gh auth status 로그인 여부
4. git user.name / user.email 설정 여부
5. 현재 작업 폴더 위치

주의:
- 내 GitHub 인증 코드, 토큰, 비밀번호는 절대 대신 입력하지 마.
- 실제 변경 명령을 실행하기 전에는 어떤 명령을 실행할지 먼저 보여줘.
```

CHAPTER

05

4장 프로젝트를 내 컴퓨터로 가져오기

팀원이 이미 GitHub 레포를 만들고 여러분을 collaborator로 초대했다면, 초대 수락 후 clone부터 시작합니다.

1. **레포 주소를 받습니다.**

예: `github.com/team-name/project-name` 또는 `OWNER/REPO`

2. **GitHub CLI로 clone합니다.**

내 컴퓨터에 프로젝트 폴더가 생깁니다.

3. **설치와 실행은 프로젝트마다 다릅니다.**

`README.md`, `package.json`, `pyproject.toml` 같은 파일을 AI에게 읽히면 빠릅니다.

Terminal 복사

```
gh repo clone OWNER/REPO
cd REPO
```

AI에게 프로젝트 파악시키기 복사

```
이 저장소를 처음 보는 사람에게 설명하듯이 구조를 파악해줘.

원하는 출력:
1. 이 프로젝트가 무엇을 하는지 한 문단 요약
2. 중요한 폴더와 파일 목록
3. 로컬에서 실행하는 명령어
4. 테스트 또는 빌드 명령어
5. 내가 함부로 건드리면 위험한 파일

아직 파일을 수정하지 말고, 읽고 설명만 해줘.
```

AI에게 설치와 실행 맡기기 복사

```
README와 설정 파일을 보고 이 프로젝트를 로컬에서 실행해줘.

진행 방식:
1. 먼저 실행할 명령어 목록을 보여줘.
2. 내가 승인하면 설치와 실행을 진행해.
3. 개발 서버가 켜지면 접속 URL을 알려줘.
4. 오류가 나면 오류 원인과 해결 후보를 쉬운 말로 설명해줘.
```

CHAPTER

06

5장 할일은 issue로 작게 쪼개기

바이브코딩에서 가장 흔한 실패는 AI에게 너무 큰 일을 한 번에 던지는 것입니다. GitHub issue는 "이번에 딱 무엇을 할지" 정하는 장치입니다.

좋은 issue

* 한 번에 끝낼 수 있는 크기 * 완료 조건이 눈에 보임 * 화면, 파일, 데이터 범위가 분명함

나쁜 issue

* "전체 개선", "예쁘게 만들기"처럼 범위가 큼 * 성공 기준이 없음 * AI가 어디까지 바꿔도 되는지 불명확함

협업에 좋은 단위

* issue 1개 * branch 1개 * PR 1개

CLI로 issue 만들기

Terminal 복사

```
gh issue create \
  --title "로그인 화면의 오류 메시지를 더 이해하기 쉽게 바꾸기" \
  --body "목표: 로그인 실패 시 사용자가 다음 행동을 알 수 있게 문구를 수정한다.

완료 조건:
- 비밀번호 오류와 네트워크 오류 문구가 구분된다.
- 모바일 화면에서 문구가 버튼과 겹치지 않는다.
- 관련 스크린샷을 PR에 첨부한다."
```

AI에게 issue 쪼개기 요청 복사

```
다음 아이디어를 GitHub issue로 쪼개줘.

아이디어:
[여기에 만들고 싶은 기능이나 개선 내용을 붙여넣기]

출력 형식:
1. issue 제목
```

```
2. 배경
3. 작업 범위
4. 완료 조건
5. 건드릴 가능성이 있는 파일
6. 너무 커 보이면 더 작은 issue로 나누기

초보자도 이해할 수 있는 말로 써줘.
```

CHAPTER

07

6장 AI와 branch에서 작업하기

AI에게 코드를 맡길 때도 작업 공간은 branch입니다. branch 이름은 짧고 설명적으로 짓습니다. 예:

`fix-login-error-copy`, `feat-profile-card`, `docs-github-guide`

작업 전 최신 상태로 맞추기

Terminal 복사

```
git switch main
git pull
git switch -c fix-login-error-copy
```

AI에게 일을 시키는 기본 프롬프트

AI 작업 프롬프트 복사

```
현재 branch에서 issue #[번호] 작업을 진행해줘.

작업 목표:
[무엇을 바꿀지 적기]

제약:
- issue 범위를 벗어난 리팩터링은 하지 마.
- 관련 없는 파일 포맷팅은 하지 마.
- 비밀키, 토큰, .env 파일은 읽거나 출력하지 마.
- 큰 변경이 필요하면 먼저 나에게 이유를 설명하고 승인받아.

진행 방식:
1. 먼저 수정할 파일과 작업 계획을 5줄 이내로 말해줘.
2. 내가 승인하면 수정해.
3. 수정 후 git diff 요약과 확인 방법을 알려줘.
```

작업 중간에 AI를 멈추고 확인시키기

AI 점검 프롬프트 복사

```
지금까지 변경한 내용을 멈추고 점검해줘.

출력:
1. 변경된 파일 목록
2. 각 파일에서 바뀐 이유
```

```
3. issue 범위를 벗어난 변경이 있는지
4. 내가 브라우저에서 확인해야 할 화면 또는 동작
5. 다음으로 할 일

아직 commit하지 마.
```

중요 : AI가 "다 고쳤다"고 말해도 바로 믿지 않습니다. `git diff`, 화면 확인, 테스트 명령어 중 최소 하나는 직접 확인합니다.

CHAPTER

08

7장 커밋과 푸시

commit은 팀원에게 보여주는 작업 저장 지점입니다. AI가 많은 파일을 바꿨다면 더더욱 관련 변경만 골라서 commit해야 합니다.

변경 확인

Terminal 복사

```
git status
git diff
```

AI에게 commit 전 검토시키기

AI self-review 프롬프트 복사

```
git status와 git diff를 바탕으로 commit 전에 검토해줘.

출력:
1. 이번 변경 요약
2. 관련 없는 변경이 섞였는지
3. 위험하거나 되돌려야 할 변경 후보
4. 실행해야 할 테스트/빌드 명령
5. 추천 commit 메시지 3개

아직 git add나 commit은 하지 마.
```

commit과 push

Terminal 복사

```
git add 파일경로1 파일경로2
git commit -m "fix: clarify login error message"
git push -u origin fix-login-error-copy
```

AI에게 commit 맡길 때 복사

```
이번 issue와 관련된 변경만 stage하고 commit해줘.

조건:
- 먼저 git status와 git diff를 확인해.
```

```
- 관련 없는 파일이 있으면 stage하지 말고 나에게 물어봐.
- commit 메시지는 "type: 짧은 설명" 형식으로 제안하고, 내가 승인하면 commit해.
- commit 후 push까지 진행해.

추천 type:
feat, fix, docs, style, refactor, test, chore
```

CHAPTER

09

8장 Pull Request 만들기

PR은 결과물 자랑이 아니라 검토 요청서입니다. 팀원이 빠르게 이해하도록 "왜, 무엇을, 어떻게 확인하면 되는지"를 씁니다.

CLI로 PR 만들기

Terminal 복사

```
gh pr create \
  --base main \
  --head fix-login-error-copy \
  --title "fix: clarify login error message" \
  --body "## 요약
- 로그인 실패 메시지를 상황별로 구분했습니다.
- 모바일 화면에서 문구가 버튼과 겹치지 않도록 간격을 조정했습니다.

## 확인 방법
- 잘못된 비밀번호로 로그인 시도
- 네트워크를 끄고 로그인 시도
- 모바일 폭에서 오류 문구 확인

Closes #12"
```

준비 중인 PR은 draft로 올리기

작업이 덜 끝났지만 미리 보여주고 싶으면 **draft PR** 을 씁니다. draft PR은 리뷰·병합이 막혀 있고 "준비 완료" 표시를 해야 검토가 시작됩니다.

Terminal 복사

```
gh pr create --draft --base main --head fix-login-error-copy \
  --title "WIP: clarify login error message" \
  --body "아직 작업 중입니다. 완료되면 ready for review로 바꿉니다."

# 완료되면
gh pr ready NUMBER
```

AI에게 PR 본문 초안 만들게 하기

PR 초안 프롬프트 복사

```
현재 branch의 변경사항으로 GitHub PR 제목과 본문 초안을 만들어줘.

참고할 것:
- git log main..HEAD
- git diff main...HEAD
- 연결된 issue가 있으면 issue 내용

본문 형식:
## 요약
-

## 변경한 이유
-

## 확인 방법
- [ ]

## 스크린샷 또는 결과
- 필요하면 어떤 화면을 캡처해야 하는지 알려줘.

## 리뷰어에게 묻고 싶은 점
-

주의:
- 과장하지 말고 실제 변경한 내용만 써.
- 내가 실행하지 않은 테스트를 실행했다고 쓰지 마.
```

좋은 PR 제목 `fix: clarify login error message`, `feat: add profile preview card`, `docs: add setup guide`

나쁜 PR 제목 `update`, `final`, `AI가 수정함`, `여러가지 고침`

CHAPTER

010

9장 리뷰 받고 수정하기

리뷰는 혼나는 과정이 아니라 main에 들어가기 전에 서로 확인하는 과정입니다. PR의 Conversation, Commits, Checks, Files changed 탭을 차례로 보면 상황을 이해하기 쉽습니다.

Conversation

제목, 본문, 리뷰 댓글, 답글이 오가는 곳. 수정 요청도 여기에 쌓입니다.

Commits

이 PR에 담긴 commit 목록. 한 번에 몰아서 올렸는지, 작게 쪼갰는지 볼 수 있습니다.

Checks (CI)

자동 검사 결과. 빌드·테스트·린트가 설정되어 있으면 여기서 초록 체크·빨간 실패로 표시됩니다. 빨간 실패 표시가 있으면 원인을 확인한 뒤 다시 push합니다.

Files changed

실제 변경된 코드(diff)입니다. 줄별로 코멘트를 남기는 곳이며, 리뷰의 핵심입니다.

작성자가 할 일

* 작은 PR로 올리기 * 확인 방법 쓰기 * 스스로 diff 먼저 보기 * 리뷰 댓글에 답하거나 수정하기

리뷰어가 볼 것

* issue 범위를 지켰는지 * 화면이나 기능이 실제로 동작하는지 * 관련 없는 변경이 섞였는지 * 읽기 어려운 코드나 설명이 있는지

수정 요청이 오면

* 새 branch를 만들지 않습니다. * 같은 branch에서 수정합니다. * commit 후 push하면 기존 PR이 자동 업데이트됩니다.

리뷰 댓글을 AI에게 정리시키기

리뷰 대응 프롬프트 복사

```
PR 리뷰 댓글을 보고 수정 계획을 세워줘.

입력:
[리뷰 댓글을 붙여넣기]

출력:
1. 반드시 수정해야 하는 항목
2. 질문에 답하면 되는 항목
3. 수정할 파일 후보
4. 각 항목의 위험도
5. 내가 리뷰어에게 답글로 남길 문장 초안

아직 코드는 수정하지 마.
```

수정 후 다시 push

Terminal 복사

```
git status
git add 파일경로
git commit -m "fix: address review feedback"
git push
```

main 최신 변경 반영하기

Terminal 복사

```
git switch main
git pull
git switch 내-작업-브랜치
git merge main
```

CHAPTER

011

10장 fork는 언제 쓰나

fork는 팀 내부 협업의 기본값이 아닙니다. 보통 권한이 없는 외부 프로젝트에 기여하거나, 원본과 독립된 실험 공간이 필요할 때 씁니다.

공동 레포 branch 방식

* 팀원이 같은 레포에 초대되어 있음 * KLIC 내부 팀 프로젝트에 적합 * PR은 같은 레포의 branch 끼리 생성

fork 방식

* 원본 레포에 push 권한이 없음 * 오픈소스나 외부 프로젝트 참여에 적합 * 내 계정의 fork에서 작업 후 원본으로 PR

fork 방식 명령 예시

Terminal 복사

```
    gh repo fork OWNER/REPO --clone --remote
    cd REPO
    git switch -c fix-small-typo

    # 작업 후
    git add .
    git commit -m "docs: fix typo"
    git push -u origin fix-small-typo
    gh pr create --repo OWNER/REPO --base main --head 내아이디:fix-small-
typo
```

CHAPTER

012

11장 바이브코더 협업 규칙

AI를 쓰면 속도는 빨라지지만, 변경 범위가 커지고 의도를 놓치기 쉽습니다. 아래 규칙은 팀 프로젝트에서 사고를 줄이는 기본선입니다.

작업 규칙

* 작업 전 `git pull` * 작업마다 새 branch * PR은 작게 * AI에게 먼저 계획을 말하게 하기

확인 규칙

* commit 전 `git diff` * PR 전 직접 실행 또는 화면 확인 * 테스트하지 않았으면 테스트했다고 쓰지 않기 * AI가 만든 설명은 사람이 고쳐 쓰기

금지 규칙

* main에 직접 push하지 않기 * 비밀키와 토큰을 AI에게 붙여넣지 않기 * 의미 모를 대량 변경을 한 PR에 넣지 않기 * `reset --hard`, force push는 혼자 결정하지 않기

레포 보호 설정 (담당자가 한 번만)

아래 설정은 GitHub 웹에서 담당자가 켭니다. 켜 두면 실수로 main이 망가지는 일을 막아 줍니다.

branch protection

main 브랜치 보호. 직접 push 금지, PR과 리뷰 승인을 의무화. `Settings → Branches`.

status checks

CI(빌드·테스트)가 통과해야만 병합 허용. 빨간 실패 표시인 PR은 merge 버튼이 막힙니다.

CODEOWNERS

특정 폴더/파일을 고치면 지정한 리뷰어가 자동으로 필수 승인자가 됩니다. 루트 `.github/CODEOWNERS` 파일로 설정.

merge 전략

보통 **squash merge**(PR 커밋을 하나로 합침) 추천. 작은 팀은 history가 깔끔해집니다. `Settings → General → Pull Requests`.

PR 올리기 전 체크리스트

* 현재 branch가 main이 아니다. * issue 또는 작업 목표가 PR 본문에 연결되어 있다. * 관련 없는 파일 변경이 섞이지 않았다. * 직접 실행, 화면 확인, 빌드, 테스트 중 하나 이상을 했다. * 리뷰어가 볼 수 있게 확인 방법을 적었다. * AI가 만든 변경 중 이해하지 못한 부분을 질문했다.

CHAPTER

013

12장 자주 막히는 상황

막혔을 때는 에러 메시지를 지우지 말고 그대로 복사해서 AI에게 설명을 요청합니다.

`gh: command not found`

GitHub CLI가 설치되어 있지 않습니다.

AI에게 설치 요청 복사

```
    현재 운영체제에 GitHub CLI(gh)를 설치하는 방법을 안내해줘.
    가능하면 공식 설치 방법을 기준으로 하고, 설치 후 gh --version과 gh auth status 확인
까지 도와줘.
```

`fatal: not a git repository`

현재 위치가 Git 프로젝트 폴더가 아닙니다. `pwd`, `ls`로 위치를 확인하고 clone한 폴더로 이동합니다.

Terminal 복사

```
    pwd
    ls
    cd REPO
```

push가 거절됩니다

대부분 원격 저장소에 새 변경이 먼저 올라왔거나, 내 branch에 push 권한이 없을 때 발생합니다.

AI에게 진단 요청 복사

```
    git push가 실패했습니다. 아래 에러를 읽고 원인을 설명해줘.

    [에러 메시지 붙여넣기]

    원하는 출력:
    1. 원인
    2. 안전한 해결 순서
    3. 실행할 명령어
    4. 작업 내용이 사라질 위험이 있는 명령어가 있다면 경고
```

merge conflict가 났습니다

같은 부분을 서로 다르게 고친 상황입니다. 먼저 충돌 파일을 확인하고, 어떤 내용을 살릴지 결정합니다.

Terminal 복사

```
    # 1. 충돌이 난 파일 목록 보기
    git status

    # 2. 파일을 열면 아래 같은 표시가 있음. 원하는 쪽만 남기고 표시를 지운다
```

```
    # <<<<<<< HEAD
    # 내가 고친 내용
    # =========
    # 상대방이 고친 내용
    # >>>>>>> 브랜치이름

    # 3. 표시를 다 지우고 저장한 뒤
    git add 파일경로

    # 4. (merge 중이었다면) 계속 진행
    git status   # "all conflicts fixed" 확인
    git merge --continue

    # 처음부터 그만두고 싶으면
    git merge --abort
```

주의 : 충돌 표시(`<<<<<<<`, `=======`, `>>>>>>>`)가 하나라도 남으면 commit이 막힙니다. 파일을 저장하기 전에 꼭 확인합니다.

AI에게 conflict 설명 요청 복사

```
    merge conflict가 났습니다.

    먼저 충돌 파일 목록을 확인하고, 각 충돌 블록에서 양쪽 변경이 무엇을 의미하는지 초보자도 이해하게
설명해줘.
    내가 어떤 쪽을 선택할지 결정할 수 있게 선택지를 제시해줘.

    주의:
    - 내가 선택하기 전에는 파일을 수정하지 마.
    - 해결 후에는 어떤 테스트를 해야 하는지 알려줘.
```

AI가 너무 많은 파일을 바꿨습니다

바로 commit하지 말고 변경 목록부터 분리합니다.

AI에게 변경 분류 요청 복사

```
    이번 변경이 너무 커졌습니다. git diff를 보고 변경을 분류해줘.

    분류:
    1. issue와 직접 관련 있는 변경
    2. 관련은 있지만 별도 PR로 빼는 게 좋은 변경
    3. 되돌리는 게 좋은 변경
```

```
    각 분류별 파일 목록과 이유를 알려줘.
    아직 되돌리거나 commit하지 마.
```

되돌리고 싶습니다

되돌리기는 작업 단계에 따라 방법이 다릅니다. commit 전과 commit 후를 구분해야 합니다.

Terminal 복사

```
    # 1. commit 전: 특정 파일의 변경을 버리기
    git restore 파일경로

    # 2. commit 전: 전체 변경을 버리기 (신중)
    git restore .

    # 3. commit 후, 아직 push 안 함: 마지막 commit 취소, 변경은 유지
    git reset --soft HEAD~1

    # 4. commit 후 push까지 한 상태: 기록을 지우지 않고 되돌리기 (안전, 추천)
    git revert HEAD
    git push

    # 주의 — 아래는 작업과 기록을 영구 삭제. 절대 혼자 결정하지 마세요
    # git reset --hard
    # git push --force
```

AI에게 안전하게 되돌리기 요청 복사

```
    작업을 되돌리고 싶습니다.

    먼저 현재 상태를 확인해줘:
    - git status
    - git log --oneline -5
    - git diff

    그 다음 내 상황이 commit 전인지, commit 후인지, push 후인지 구분해서 가장 안전한 되돌리
기 방법을 설명해줘.
    작업이 사라지는 명령은 실행하기 전에 반드시 나에게 확인받아.
```

CHAPTER

014

13장 자동화로 검증하기 (GitHub Actions)

Pull Request를 올리면 사람이 직접 확인하기 전에 기계가 먼저 검사하게 만들 수 있다. GitHub Actions는 push나 PR이 발생했을 때 코드를 자동으로 빌드하고 테스트하는 CI(지속적 통합) 도구다. 리뷰어가 "돌려보니 에러가 나네요"라고 말하는 대신, 기계가 "이 PR은 테스트를 통과했습니다"라고 먼저 알려준다.

workflow, 워크플로

`.github/workflows/` 폴더 아래 `.yml` 파일로 작성하는 자동화 단위다. 누가, 언제, 무엇을 할지 적는다.

on, 트리거

언제 워크플로를 실행할지 정하는 조건이다. `push`, `pull_request`가 가장 흔하다.

job, step

잡은 하나의 실행 단위(예: test), 스텝은 잡 안의 세부 동작(예: 패키지 설치 → 테스트 실행)이다. 잡은 서로 의존 관계를 가질 수 있다.

runner

명령을 실제로 실행하는 기계다. GitHub이 제공하는 github-hosted runner와 직접 구축하는 self-hosted runner가 있다. 소규모 팀은 github-hosted로 충분하다.

최소 CI 워크플로는 PR이 열릴 때 lint, test, build를 차례로 돌리고 하나라도 실패하면 PR에 실패 표시를 남긴다. 레포 보호 설정에서 "이 검사를 통과해야만 병합 가능"으로 묶으면, 기계가 통과시킨 PR만 main에 들어간다. AI가 만든 코드도 예외 없이 같은 게이트를 통과한다.

> "**AI에게 붙여넣는 프롬프트**: 이 저장소에 GitHub Actions CI 워크플로를 만들어줘. PR이 열릴 때 lint, 단위 테스트, 빌드를 차례로 실행하고, 하나라도 실패하면 PR을 병합하지 못하게 해. 실패 원인을 PR 코멘트에 남겨."

CHAPTER

015

14장 변경 합치기와 충돌 해결

두 branch를 합칠 때 같은 파일의 같은 줄을 서로 다르게 고쳤다면, Git은 어느 쪽이 맞는지 스스로 판단하지 못한다. 이 상태를 **conflict(충돌)**라고 한다. 충돌은 에러가 아니라 "사람이 판단해 달라"는 신호다.

merge, 병합

한 branch의 변경을 다른 branch로 끌어넣는 방식이다. main에 작업 branch를 합칠 때 쓴다.

fast-forward

main이 가만히 있고 작업 branch만 앞으로 나간 경우, main 포인터를 앞으로 옮기기만 하면 합쳐진다. 병합 커밋이 생기지 않아 히스토리가 일직선이 된다.

3-way merge

양쪽이 모두 새 커밋을 가진 경우, 두 부모를 둔 병합 커밋을 만든다. 히스토리는 갈라지지만 어떤 작업이 합쳐졌는지 흔적이 남는다.

rebase, 리베이스

작업 branch의 출발점을 최신 main 위로 옮겨 히스토리를 일직선으로 정리한다. 커밋이 깔끔하게 쌓이지만, 이미 공유한 branch에 쓰면 동료의 히스토리가 꼬인다.

충돌이 생기면 `git status`로 충돌 파일을 찾고, 파일을 열어 `<<<<<<<`, `=======`, `>>>>>>>` 표시 사이를 직접 편집해 한 쪽(또는 둘을 합친 새 버전)으로 고친 뒤 `git add` → `git commit`한다. conflict 표식을 파일에 남겨두면 안 된다.

force push는 자신이 쓰는 작업 branch에서만 쓴다. main이나 동료가 함께 쓰는 branch에 force push를 하면 다른 사람의 커밋이 지워진다. rebase로 히스토리를 정리한 뒤 내 branch에 올릴 때만 쓴다.

> "**AI에게 붙여넣는 프롬프트**: 지금 충돌이 난 파일을 보여줄게. 양쪽 변경 의도를 설명하고, 어느 쪽을 택하거나 어떻게 합쳐야 할지 판단해줘. conflict 표식은 지우고 최종본만 보여줘."

CHAPTER

016

15장 릴리스와 태그 (버전 관리)

"이 지점이 1.0이다"라고 커밋에 이름표를 붙이는 일이 릴리스다. 코드가 계속 바뀌어도, 특정 시점의 안정된 상태를 이름으로 부르고 돌아갈 수 있다.

tag, 태그

특정 커밋에 붙이는 고정 이름표. `git tag v1.0.0`으로 만든다. branch와 달리 이동하지 않는다.

semantic versioning, 시맨틱 버전

`v1.2.3`처럼 세 자리 숫자로 버전을 표기하는 규칙이다.

- **major(1)**: 기존 사용 방법이 깨지는 큰 변경
- **minor(2)**: 호환되는 새 기능 추가
- **patch(3)**: 버그 수정

어떤 자릿수가 올라갔는지로 변경의 성격을 알린다.

release

GitHub의 릴리스는 태그에 제목과 release notes(무엇이 바뀌었는지 요약)를 얹은 것이다. 태그 날짜와 릴리스 날짜는 다를 수 있다.

automated release notes

GitHub은 병합된 PR 목록과 기여자, 전체 변경로그를 모아 release notes를 자동으로 만들어 준다. 라벨로 카테고리를 나누면 "새 기능", "버그 수정" 등으로 정리된다.

실무에서는 `release/v1` 같은 릴리스 branch에서 최종 검증을 마치고, 시맨틱 버전 태그를 붙여 릴리스를 만든다. 릴리스 노트에는 사용자가 "무엇이 달라졌는지"를 한눈에 알 수 있도록 변경을 요약한다.

> "**AI에게 붙여넣는 프롬프트**: 이번 릴리스에 병합된 PR 목록을 줄게. 시맨틱 버전 기준으로 다음 버전 번호를 정하고, 일반 사용자가 이해할 수 있게 변경 사항을 요약한 release notes를 만들어줘."

CHAPTER

017

16장 참고 문서

이 자료는 아래 공식 문서를 기준으로 협업 흐름을 KLIC 조직 실무용으로 다시 설명했습니다.

* [GitHub Docs: GitHub flow](https://docs.github.com/en/get-started/using-github/github-flow) * [GitHub Docs: About collaborative development models](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/getting-started/about-collaborative-development-models) * [GitHub Docs: About pull requests](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests) * [GitHub Docs: Helping others review your changes](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/getting-started/helping-others-review-your-changes) * [GitHub Docs: Creating an issue](https://docs.github.com/en/issues/tracking-your-work-with-issues/using-issues/creating-an-issue) * [GitHub CLI Manual: gh auth login](https://cli.github.com/manual/gh_auth_login) * [GitHub CLI Manual: gh repo clone](https://cli.github.com/manual/gh_repo_clone) * [GitHub CLI Manual: gh pr create](https://cli.github.com/manual/gh_pr_create)

바이브코딩 GitHub 협업 가이드 · 공동 레포, branch, PR 중심 · AI 코딩 도구와 함께 쓰는 실습 자료