8. 협업 워크플로우

Git Flow 전략

✅ 1. Git Flow란?

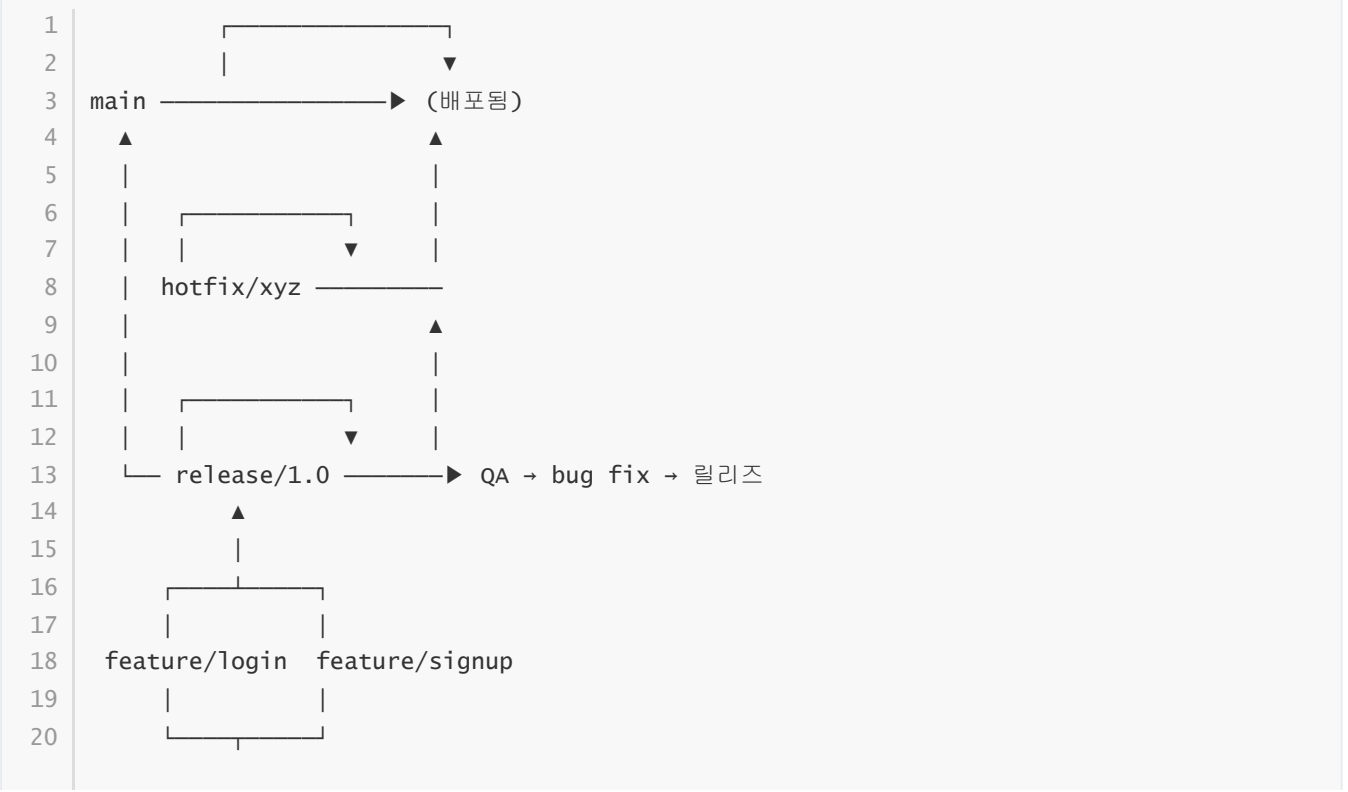
> **Vincent Driessen**이 제안한 Git 브랜치 전략으로,
> 기능 개발, 테스트, 릴리즈, 유지보수 등을 **명확하게 분리된 브랜치**로 관리하는 워크플로우.

- ✅ 대규모 협업과 릴리즈 주기가 명확한 프로젝트에 적합
- ✅ 안정성 유지와 병렬 작업에 강함

✅ 2. 브랜치 종류 요약

브랜치	용도	특징
`main` (또는 `master`)	**프로덕션(배포) 버전**	언제나 배포 가능한 상태 유지
`develop`	**개발 통합 브랜치**	모든 기능 개발의 기준선
`feature/*`	기능 개발 브랜치	`develop`에서 파생 → `develop`으로 병합
`release/*`	릴리즈 준비 브랜치	QA, 버그 수정 후 `main`, `develop`으로 병합
`hotfix/*`	긴급 수정 브랜치	`main`에서 직접 파생 → `main`, `develop`으로 병합

✅ 3. 브랜치 흐름 요약도



```
          ▼
      develop (통합 개발)
```

✅ 4. 사용 예시

🔹 기능 개발 (feature)

```
git checkout develop
git checkout -b feature/login

# 작업 후
git commit -m "feat: 로그인 기능"
git push origin feature/login

# PR: feature/login → develop
```

🔹 릴리즈 준비 (release)

```
git checkout develop
git checkout -b release/1.0.0

# 버전 업, QA 등
git commit -m "chore: v1.0.0 릴리즈 준비"

# 병합
git checkout main
git merge release/1.0.0
git tag v1.0.0

git checkout develop
git merge release/1.0.0

# 브랜치 삭제
git branch -d release/1.0.0
```

🔹 긴급 수정 (hotfix)

```
git checkout main
git checkout -b hotfix/crash-fix

# 수정 작업
git commit -m "fix: 로그인 크래시 수정"

# 병합
git checkout main
git merge hotfix/crash-fix
git tag v1.0.1

git checkout develop
git merge hotfix/crash-fix
```

✅ 5. Git Flow 도구 지원

자동화된 Git Flow 명령어도 존재한다:

```
brew install git-flow

git flow init
# 브랜치 명칭, prefix 등 설정

git flow feature start login
git flow feature finish login

git flow release start 1.0.0
git flow release finish 1.0.0
```

✅ 6. 장점 vs 단점

항목	장점	단점
유지보수	안정성과 QA 분리	브랜치가 많아짐
협업	작업 단위 분리 용이	간단한 프로젝트엔 과함
릴리즈 관리	버전별 배포 용이	CI/CD 자동화 시 복잡

✅ 7. Git Flow가 적합한 경우

- 팀원이 많고 병렬 작업이 활발한 프로젝트
- **배포 주기가 정해져 있는 제품형 소프트웨어**
- 코드 안정성과 리뷰 프로세스를 **엄격히 분리**하고 싶을 때

✅ 8. Git Flow보다 더 단순한 대안

전략	특징
GitHub Flow	`main`만 사용, PR 중심 개발
Trunk Based Development	`main`에 직접 통합 + Feature Toggle 사용
GitLab Flow	Git Flow + 환경 브랜치 (staging, production 등) 혼합

✅ 마무리 요약

브랜치	기반	병합 대상
`feature/`	`develop`	`develop`
`release/`	`develop`	`main`, `develop`
`hotfix/`	`main`	`main`, `develop`
`develop`	여러 기능	릴리즈 브랜치로
`main`	배포 버전	외부 공개, 배포

GitHub Flow 전략

✅ 1. GitHub Flow란?

> GitHub에서 오픈소스 및 웹 서비스를 위한 **민첩한 개발 워크플로우**로 제안한 전략
> 단 하나의 `main` **브랜치**를 중심으로,
> 기능 브랜치 → PR → 리뷰 → Merge → 자동 배포라는 단순한 구조

✅ 2. 핵심 원칙 요약

원칙	설명
`main`은 언제나 **배포 가능한 상태**	
새로운 기능은 **항상 별도 브랜치**에서 개발	
모든 변경은 **Pull Request(PR)**를 통해 리뷰	
PR 병합은 **자동 테스트 통과 후**	
병합되면 자동 또는 수동으로 **배포 트리거**	

✅ 3. 브랜치 흐름 구조

```
main  ← 배포 가능한 최신 코드
  ↑
 Pull Request
  ↑
feature/login ← 기능 브랜치 (로컬 또는 Fork)
```

✅ 4. 실전 워크플로우

🔹 Step 1. 브랜치 생성

```
git checkout -b feature/login
```

🔹 Step 2. 작업 & 커밋

```
git add .
git commit -m "feat: 로그인 UI 추가"
git push origin feature/login
```

🔹 Step 3. Pull Request 생성

- GitHub에서 PR 생성
- Base: `main`, Compare: `feature/login`
- 코드 리뷰 & 자동 테스트 수행

🔹 Step 4. Merge 후 배포

- 리뷰 승인 & 테스트 통과 → Merge
- 병합되면 자동 배포 시스템 (CI/CD) 작동

✅ 5. GitHub Flow vs Git Flow

항목	GitHub Flow	Git Flow
브랜치 수	최소 (`main`, `feature`)	다수 (`main`, `develop`, `release`, `hotfix`)
릴리즈 전략	지속 배포(CD)	명시적 버전 릴리즈
적합 대상	웹서비스, 스타트업	제품 릴리즈 중심 소프트웨어
자동화 연계	강제됨 (CI/CD 필요)	수동 배포도 가능

✅ 6. GitHub Flow에 적합한 프로젝트

✅ 웹 기반 SaaS
✅ 지속 배포(Continuous Deployment)를 추구하는 팀
✅ 협업이 빠르고 리뷰가 짧은 조직
✅ 자동화 테스트가 잘 갖춰진 환경

✅ 7. 예시 PR 구조

```
# feat: 회원가입 기능 추가

## ✅ 변경 사항
- 이메일/비밀번호 폼 추가
- API 호출 로직 구현
- 반응형 디자인 적용

## 🧪 테스트
- [x] 모바일 화면에서 렌더링 확인
- [x] 서버 응답 200 정상 처리

Fixes #23
```

✅ 8. GitHub Actions + GitHub Flow

- `main` 브랜치로 병합되면 자동으로 Actions가 실행됨:

```
on:
  push:
    branches: [main]
```

- PR에 테스트 및 코드 품질 검사 자동 수행

✅ 요약 비교표

요소	GitHub Flow 특징
중심 브랜치	`main` 하나
릴리즈 방식	지속적인 배포 (CD)
브랜치 전략	`feature/*` 만 사용
PR 방식	병합 전 반드시 리뷰 & 테스트
배포 트리거	PR 머지 후 자동 배포 연계
적합 환경	빠른 반복, 소규모 팀, SaaS

✅ 실전 예시 흐름 요약

```
git checkout -b feature/signup
# 개발 후
git push origin feature/signup
# → GitHub에서 PR 생성 → 리뷰 → Merge → 배포
```

trunk-based 개발 방식

✅ 1. Trunk-Based Development (TBD)란?

> 모든 개발자가 하나의 **공통 브랜치(trunk)**를 중심으로
> **짧고 빈번한 커밋**을 반복하며,
> 지속적으로 통합(Continuous Integration)하는 개발 방식.

✅ 2. 핵심 특징

항목	설명
🔁 **trunk (보통 `main` 브랜치)** 에서 모든 개발이 이루어짐	
⚡ **짧은-lived 브랜치** (수시간~하루 이내 사용)	
🔀 **빠른 머지**, 병합 지연 없음	
✅ **항상 배포 가능한 상태 유지**	
🚨 **자동화된 테스트/배포 필수 (CI/CD 연계)**	
🚧 **Feature Toggle**로 미완성 기능 숨김	

✅ 3. GitHub Flow vs Git Flow vs TBD 비교

항목	Git Flow	GitHub Flow	Trunk-Based
중심 브랜치	`main`, `develop`	`main`	`main` (trunk)
브랜치 수명	중~장기	중기	초단기 (수시간~1일)
병합 시점	기능 완성 후	PR 리뷰 후	즉시 또는 짧은 리뷰 후
테스트 체계	수동/CI 병행	CI 중심	CI/CD 필수
기능 분리	브랜치	브랜치	Feature Toggle (플래그)
이상적 환경	전통적 릴리즈	민첩한 팀	DevOps, 대규모 협업

✅ 4. 실전 워크플로우

🟩 Step 1. `main`에서 파생한 짧은 브랜치

```
git checkout -b wip/signup-button
```

🟩 Step 2. 수 시간 내 변경 완료 + 테스트 통과

```
# 코드 작성 후
git add .
git commit -m "feat: 버튼 UI 추가 (비활성 상태)"
```

🟩 Step 3. 자동화 테스트 → 병합

- PR 생성하거나 직접 push 후 `main`에 병합
- **미완성 기능은 Feature Toggle로 숨김**

```
git push origin wip/signup-button
# PR → GitHub Actions → Merge
```

✅ 5. Feature Toggle (기능 플래그)

> 미완성 기능은 브랜치가 아닌 **조건문/플래그**로 숨긴다

```
if (isFeatureEnabled('newSignupButton')) {
  renderNewButton();
} else {
  renderOldButton();
}
```

- 설정 파일, 환경 변수, A/B 테스트 툴 등으로 제어 가능
- 대표 툴: LaunchDarkly, Firebase Remote Config 등

✅ 6. TBD 전략의 이점

장점	내용
🔀 병합 지옥 방지	기능 브랜치가 너무 오래 살아남지 않음
🚀 빠른 배포	항상 배포 가능한 코드
🔎 빠른 피드백	작은 단위로 오류 감지 용이
🤝 협업 효율	통합 지연 없음, 항상 최신 코드

⚠️ 7. 성공적인 TBD를 위한 전제 조건

필수 요소	설명
✅ CI/CD 구축	매 커밋/PR마다 자동 빌드+테스트
✅ Feature Toggle 체계	브랜치 대신 기능 상태 제어
✅ 코드 품질 체크 자동화	Lint, 테스트 커버리지 등
✅ 병합 전략 관리	Fast-forward 병합 또는 squash 전략

✅ 8. 대기업과 오픈소스 사례

조직	적용 방식
Google	전 세계 개발자가 하나의 monorepo + trunk 사용
Facebook	Feature Toggle 기반 Trunk 전략
Netflix	Pull Request 없이 trunk 직접 병합 (자동 테스트 필수)
GitHub	GitHub Flow 기반에서 Trunk 전략 혼합

✅ Trunk-Based Development는 언제 좋을까?

적합 상황	설명
✅ 매일 배포 또는 실시간 배포가 필요한 서비스	
✅ 작은 기능을 빠르게 반복 개발할 때	
✅ 대규모 팀 간 충돌 없이 개발해야 할 때	
✅ DevOps/CI/CD 중심 조직일 때	

🧠 마무리 요약

- `main` 브랜치 = **항상 배포 가능한 trunk**
- 모든 개발은 **짧은-lived 브랜치** 또는 **직접 커밋**
- 미완성 기능은 **Feature Toggle**로 통제
- CI/CD 자동화와 함께 쓰면 **병합 지옥도, QA 지옥도 사라짐**

Fork 기반 협업 전략

✅ 1. 개념 요약

> 저장소를 직접 수정할 권한이 없는 외부 기여자(contributor)가
> 원본 저장소를 **자신의 계정으로 복제(fork)** 하여 개발한 뒤,
> **Pull Request(PR)** 를 통해 변경 사항을 제안하는 방식

✅ 2. 핵심 구조

```
원본 저장소: upstream/repo (예: organization/project)
포크된 저장소: your-username/repo (복제본)

작업 흐름:
1. Fork
2. Clone (본인 repo)
3. 브랜치 생성 & 개발
4. Push
5. Pull Request → upstream 저장소로
6. 리뷰 → 병합
```

✅ 3. Fork 기반 협업 절차

🟢 1. 저장소 Fork

- GitHub에서 `[Fork]` 버튼 클릭 → 자신의 계정으로 저장소 복제됨
- 주소: `https://github.com/your-name/original-project`

🟢 2. 로컬에 Clone

```
git clone https://github.com/your-name/original-project.git
cd original-project
```

🟢 3. 원본 저장소를 upstream으로 등록

```
git remote add upstream https://github.com/original-author/original-project.git
```

```
git remote -v
# origin   → 내 repo
# upstream → 원본 repo
```

🟢 4. 브랜치 생성 후 작업

```
git checkout -b feature/new-ui
# 작업 후
git add .
git commit -m "feat: 새로운 UI 구성 추가"
```

🟢 5. 자신의 저장소(origin)에 Push

```
git push origin feature/new-ui
```

🟢 6. GitHub에서 PR 생성

- Base: `upstream/main`
- Compare: `your-name:feature/new-ui`

🟢 7. 리뷰 & 병합

- upstream의 관리자가 리뷰 후 병합
- 피드백이 있으면 수정 → push → 자동 반영

✅ 4. 원본 업데이트 반영 (동기화)

```
git checkout main
git pull upstream main
git push origin main
```

✅ 5. Fork 기반 전략이 적합한 경우

상황	설명
오픈소스 프로젝트	외부 기여자가 직접 수정 권한이 없는 경우
기업 내부 협업	권한 제한이 있는 조직 구조
수십~수백 명이 참가하는 기여	브랜치 충돌 방지 및 독립된 작업공간 보장

✅ 6. 장점과 단점

장점	단점
🔐 원본에 쓰기 권한 없이도 기여 가능	🔁 원본과의 싱크를 수동으로 맞춰야 함

장점	단점
💡 독립적인 작업공간 제공	❗ 리뷰 전까지는 upstream에 아무 변화 없음
🧪 CI 실행 시 격리 환경 가능	📝 관리자가 PR 병합 전까진 공식 코드가 아님

✅ 7. 실전 Git 명령어 요약

```
# upstream 등록
git remote add upstream https://github.com/original/repo.git

# 최신 동기화
git fetch upstream
git checkout main
git merge upstream/main

# 브랜치 작업
git checkout -b fix/docs
git push origin fix/docs

# PR 생성
# → GitHub에서 Compare & Pull Request 클릭
```

✅ 요약

단계	명령 또는 행동
저장소 복제	GitHub에서 Fork 클릭
로컬 복제	`git clone`
원본 연결	`git remote add upstream`
동기화	`git pull upstream main`
기능 개발	`git checkout -b feature/...`
PR 제출	`push` → GitHub에서 PR 생성

Fork 기반 전략은
✅ 기여자에게 자유로운 개발 공간을 주고,
✅ 코드베이스의 안전성을 유지할 수 있어서
오픈소스 기여의 표준 전략으로 널리 사용된다.

Feature 브랜치 전략

✅ 1. 개요

> 각 **기능(Feature)마다 독립적인 브랜치**를 만들어 개발하고,
> 테스트 후 메인 브랜치(`main`, `develop` 등)에 병합하는 전략.

🌱 핵심 개념:

- 각 작업(feature, bugfix, refactor 등)을 **고립된 브랜치**에서 처리
- 메인 브랜치는 **항상 배포 가능한 상태 유지**
- **Pull Request(PR)** 또는 **Merge Request(MR)** 를 통해 병합

✅ 2. 브랜치 흐름 예시

```
main ──────────────▶ (항상 배포 가능)
   ▲
   |
 develop ──────────▶ (기능 통합용 - 선택사항)
   ▲
   |
feature/login ─────▶ (로그인 기능 개발)
feature/signup ────▶ (회원가입 기능 개발)
```

✅ 3. 실전 절차

🟢 1. 기능 브랜치 생성

```
git checkout -b feature/login
```

> 브랜치 이름 규칙:
>
> - `feature/` 접두사 + 기능명
> - 예: `feature/search-bar`, `feature/payment-gateway`

🟢 2. 기능 개발 및 커밋

```
git add .
git commit -m "feat: 로그인 버튼 추가"
```

🟢 3. Push 및 Pull Request

```
git push origin feature/login
```

- GitHub/GitLab에서 PR 생성
- Base: `main` 또는 `develop`
- Compare: `feature/login`

🟢 4. 코드 리뷰 및 자동 테스트

- 팀원 또는 리드 개발자가 리뷰
- CI가 자동 테스트 수행

🟢 5. 메인 브랜치에 병합

```
# 리뷰 승인 후
git checkout main
git merge feature/login

# 또는 PR에서 병합 버튼 클릭
```

> 병합 전략은 일반적으로 **squash merge** 또는 **rebase** 를 활용해
> 커밋 로그를 깔끔하게 정리함

🟢 6. 브랜치 정리

```
git branch -d feature/login
git push origin --delete feature/login
```

✅ 4. 전략적 이점

장점	설명
✅ 기능별 코드 격리	독립적 개발 가능
✅ 병렬 작업 최적화	여러 기능 동시 개발
✅ 코드 리뷰 용이	PR 중심 협업 가능
✅ 버그 전파 방지	미완성 코드를 main에 넣지 않음

✅ 5. 브랜치 명명 규칙 (추천)

접두사	용도
`feature/`	기능 개발
`fix/`	버그 수정
`hotfix/`	긴급 배포용 수정
`refactor/`	리팩토링
`test/`	테스트 코드
`docs/`	문서 수정
`chore/`	기타 잡일 처리

> 예: `feature/payment-api`, `fix/login-validation`, `docs/readme-update`

✅ 6. Feature 브랜치 전략 + Git Flow vs GitHub Flow

비교 항목	Feature Branch 단독	Git Flow	GitHub Flow
브랜치 수	적당	많음 (`develop`, `release`, `hotfix` 등)	최소 (`main`, `feature`)
병합 흐름	`feature` → `main` or `develop`	`feature` → `develop`	`feature` → `main`
리뷰 방식	PR 기반	PR 기반	PR 기반
적합 대상	모든 규모 프로젝트	릴리즈 중심 프로젝트	CD 중심 프로젝트

✅ 7. 자동화 도구와 함께 쓰면 좋은 전략

도구	역할
GitHub Actions	브랜치 PR마다 CI 실행
SonarQube	품질 검사
Codecov	테스트 커버리지
Slack, Discord	PR 리뷰 알림 연동

✅ 실전 팁

- 각 기능은 작게 나누고 빠르게 병합하자 (micro-PR)
- 한 브랜치에서 너무 많은 작업은 금지 ❌
- 커밋 메시지 컨벤션 (`feat`, `fix`, `chore` 등) 을 지키자
- PR마다 리뷰 체크리스트 운영 👀

🧠 마무리 요약

항목	설명
핵심 아이디어	기능 단위로 분리된 브랜치에서 작업
병합 시점	기능 완성 + 테스트 통과 + 리뷰 승인 후
병합 대상	`main`, `develop` 등
정리 방식	병합 후 브랜치 삭제 (`--delete`)
전략 통합	Git Flow, GitHub Flow, GitLab Flow 등과 조합 가능

Feature 브랜치 전략은 **모든 Git 브랜치 전략의 기반**이고,
단순하지만 강력한 구조를 가진 **모던 소프트웨어 개발의 기본 흐름**이다.

Pull Request + 코드 리뷰 프로세스

✅ 1. Pull Request(PR)란?

> 한 브랜치(보통 feature 브랜치)에서 다른 브랜치(main, develop 등)로
> **코드를 병합(Merge)** 해달라고 요청하는 프로세스.

- **기능 개발 완료 → 리뷰 요청 → 테스트 → 병합** 흐름
- GitHub, GitLab, Bitbucket 등에서 지원

✅ 2. 전체 흐름 요약

```
👨‍💻 개발자 → 브랜치 생성 → 커밋 & Push
        ↓
🔁 Pull Request 생성
        ↓
🔍 코드 리뷰 + CI 실행
        ↓
✅ 리뷰 승인 + Merge
```

✅ 3. PR 생성 절차 (GitHub 기준)

📌 1. 브랜치 작업 후 Push

```
git checkout -b feature/profile-ui
# 작업 후
git add .
git commit -m "feat: 프로필 UI 추가"
git push origin feature/profile-ui
```

📌 2. PR 생성 (GitHub UI)

- `base`: 병합 대상 (보통 `main` 또는 `develop`)
- `compare`: 작업 브랜치 (`feature/profile-ui`)
- 리뷰어 지정, 라벨 추가, 설명 작성

✅ 4. PR 작성 체크리스트 📝

항목	설명
✅ 제목	`feat:`, `fix:`, `refactor:` 등 prefix 포함
✅ 설명	변경 사항 요약, 이유, 테스트 결과
✅ 관련 이슈	`Closes #12`, `Fixes #15` 형식
✅ 변경 범위	화면/UI, 로직, 의존성 등 명시
✅ 스크린샷	UI 변경 시 첨부 (Before/After)
✅ 테스트 여부	단위 테스트, 수동 테스트 통과 여부 작성

📌 예시 PR 본문:

```
### 🛠 변경 사항
- 사용자 프로필 편집 UI 추가
- avatar 업로드 기능 포함

### ✅ 테스트
- [x] 데스크탑/모바일 호환
- [x] 업로드 후 즉시 반영 확인

Closes #45
```

✅ 5. 코드 리뷰 프로세스

🔎 리뷰어의 역할

역할	설명
✅ 코드 가독성 확인	불필요한 복잡도 제거
✅ 로직 검증	요구사항과 일치하는지
✅ 사이드 이펙트	기존 기능과 충돌 없는지
✅ 테스트 확인	자동화 테스트 및 수동 테스트 적용 여부
✅ 보안 검토	인증, 인가, 입력 검증 등 포함 여부
✅ 스타일 준수	Lint, 포맷팅, 커밋 컨벤션 등 확인

✅ 6. 리뷰어 피드백 → 수정 사이클

```
# 피드백 반영 후
git add .
git commit --amend OR git commit -m "fix: 리뷰 반영"
git push origin feature/profile-ui
```

> 강제 푸시 필요할 수 있음:

```
git push --force-with-lease
```

✅ 7. PR 병합 방법

방식	특징	커밋 기록
Merge Commit	일반적인 병합 (`Merge pull request`)	병합 커밋 생성
Squash & Merge	커밋들을 하나로 압축	깔끔한 기록
Rebase & Merge	선형 히스토리 유지	원래 커밋 유지, 충돌 위험 ↑

> 팀 스타일에 따라 선택. 일반적으로 **Squash**를 많이 사용.

✅ 8. CI/CD 연동 (자동 검사)

- PR 생성/수정 시 자동으로 테스트 실행
- 예: GitHub Actions, GitLab CI, Jenkins 등

```yaml
on:
  pull_request:
    branches: [ main ]
```

- 실패 시 병합 차단 가능

✅ 9. 팀에서 자주 설정하는 PR 규칙

규칙	예시 설정
✅ 최소 리뷰어 수	1~2명 이상 승인 필요
✅ 병합 전 테스트 통과	CI 실패 시 머지 차단
✅ PR 템플릿 적용	`.github/PULL_REQUEST_TEMPLATE.md` 사용
✅ Draft PR → Ready 상태로 전환	작업 중인 PR 명확히 표시

✅ 10. 실전 팁

- PR 크기는 **작게**, 기능 단위로
- **일관된 커밋 메시지** 유지 (`Conventional Commits`)
- 리뷰어는 **단순한 approve보단 적극적 피드백**
- PR 리뷰는 **빠르게**, 24~48시간 이내가 이상적
- 리뷰와 병합이 끝나면 브랜치 정리:

```bash
git branch -d feature/profile-ui
git push origin --delete feature/profile-ui
```

🧠 요약

단계	설명
브랜치 작업	`git checkout -b feature/*`
PR 생성	GitHub에서 Pull Request 생성
코드 리뷰	리뷰어 배정 + 피드백
자동화 검사	CI/CD → 테스트, 빌드, 보안 검사
머지 방식	Merge, Squash, Rebase 중 선택
브랜치 정리	병합 후 삭제 (`--delete`)