

SOFTWARE / PRINCIPLES & PRACTICE

GitHub

원리와 실전

버전 관리에서 협업과 자동화 및 운영까지

심화 개정판 6.0 / 2026.10.24



읽기 안내와 목차

본문과 사례는 같은 절에서 이어진다. 각 페이지에는 쪽 번호만 표시한다. 연구와 공식 문서의 근거는 본문 인용 번호를 통해 참고문헌으로 연결된다.

기록을 이해하는 사람의 GitHub	4
1장 GitHub의 기본과 첫 저장소	5
2장 웹에서 문서와 변경 제안	15
3장 내 컴퓨터의 작업 환경	25
4장 인증과 변경 기록과 원격 동기화	38
5장 브랜치 통합과 충돌 해결	51
6장 복구와 파일 관리와 이력 읽기	63
7장 이슈 설계와 코드 리뷰	73
8장 오픈소스 기여와 팀 권한	84
9장 큰 저장소와 병합 운영	94
10장 웹 공개와 릴리스 관리	102
11장 클라우드 개발과 컨테이너 활용	112
12장 자동 검사와 재사용 Workflow	119
13장 비밀과 배포 권한과 사고 대응	131
14장 보안 검사와 출처 검증	142
15장 백업과 비용과 운영 인수인계	150

16장 이력 재작성과 원인 조사	158
17장 CLI와 API를 활용한 업무 연동	169
18장 종합 프로젝트와 교육 자료 운영	180
자주 쓰는 명령 빠른 찾기 1	191
자주 쓰는 명령 빠른 찾기 2	193
문제 해결 사전 1	195
문제 해결 사전 2	197
핵심 용어 사전 1	199
핵심 용어 사전 2	201
학습 계획과 역량 점검	203
참고문헌과 기술 자료	204
마지막 점검과 인계	218

기록을 이해하는 사람의 GitHub

GitHub를 능숙하게 사용한다는 것은 명령을 많이 외운다는 뜻만은 아니다. 현재 파일과 다음 커밋 및 공유된 이력을 구분하고 변경을 설명하며 실패한 결과를 복구할 수 있다는 뜻이다. 이 책은 화면의 조작을 객체와 참조 및 협업과 검증의 원리로 연결한다.

공통 저장소 lesson-lab에서 작은 문서 수정으로 시작한다. 브랜치와 병합을 이해한 뒤 리뷰와 자동화 및 공급망과 운영으로 범위를 넓힌다. 각 절의 사례는 앞에서 설명한 원리를 관찰하는 도구다. 명령의 결과가 예상과 다르면 다음 명령을 외우기보다 어떤 상태를 비교했는지 다시 읽는다.

학술 연구는 협업과 리뷰 및 측정의 배경을 제공한다. 공식 문서는 도구의 계약을 확인하는 근거다. 사례의 가상 데이터와 저자의 설계 해석은 연구의 실험 결과와 구분한다. 이 책 자체는 동료 심사를 받은 연구 논문이나 보안 인증 문서가 아니다.

개정판에서는 반복되는 페이지 머리말을 없애고 설명과 사례를 한 절에서 이어 읽도록 조판했다. HTML의 GIF는 실제 서비스 녹화가 아닌 원리 설명용 도식이다. 재생을 멈추거나 상태별로 넘겨 읽을 수 있다.

1장 GitHub의 기본과 첫 저장소

파일과 이력과 원격의 관계를 이해한 뒤 내 계정과 첫 저장소를 준비한다. 소유자와 공개 범위를 확인하는 습관부터 시작한다.

저장소는 파일 묶음과 변경의 근거를 함께 보관한다

버전 관리는 파일을 여러 번 복사하는 방법에서 출발해 생각할 수 있지만 Git의 구조를 이해하려면 상태와 참조를 구분해야 한다. 상태는 특정 시점의 파일 내용이다. 참조는 그 상태를 찾아가는 이름이다. 저장소에는 파일 상태를 표현하는 객체와 이들을 연결하는 이력 및 읽기 쉬운 이름이 존재한다. GitHub는 이 Git 저장소에 검토와 권한 및 자동화와 의사소통 기능을 더한다.

lesson-lab의 안내 문서를 고치는 사례를 보자. 이전 파일을 덮어쓴 뒤에도 과거 커밋을 통해 이전 상태를 읽을 수 있다. 다만 Git이 기록한 적 없는 미저장 편집이나 무시된 파일까지 보존해 주지는 않는다. 버전 관리가 보장하는 범위를 이해해야 백업과 협업의 책임을 정확히 나눌 수 있다.

Git의 분산성은 여러 저장소가 자체 이력을 가질 수 있다는 뜻이다. 로컬에서 커밋을 만드는 데 중앙 서버의 승인이 필요하지 않다. 그러나 어떤 커밋을 팀의 공식 결과로 인정할지는 조직의 절차와 권한 정책이 결정한다. 기술적으로 분산된 이력과 사회적으로 정한 기준 브랜치는 함께 존재한다. 분산 구조가 협업 규칙을 불필요하게 만드는 것은 아니다.

공개된 기록은 협업의 맥락이 된다

Dabbish 등의 연구는 GitHub 사용자를 인터뷰하고 활동을 관찰해 작업의 가시성이 협업과 학습에 어떤 역할을 하는지 살폈다. 변경 이유와 활동 이력이 드러나면 다른 사람이 작업을 이해하고 의존성을 파악하는 데 도움이 될 수 있다는 관점이다. 이는 초기 GitHub의 질적 연구이며 현재 모든 팀의 성과를 수치로 보장하는 결과는 아니다. [\[S12\]](#)

이 관점을 lesson-lab에 적용하면 커밋 메시지는 개인의 메모를 넘어 다음 작업자를 위한 설명이 된다. README 수정이라는 제목보다 새 사용자의 실행 경로를 명확히 했다는 설명이 검토의 목적을 보여 준다. 실제 차이는 코드에서 읽을 수 있지만 그 변경을 선택한 이유는 기록하지 않으면 사라질 수 있다.

공개성이 항상 필요한 것은 아니다. 협업에 필요한 가시성과 외부 공개는 구분한다. 비공개 저장소에서도 팀 내부의 변경 이력과 검토 과정은 공유할 수 있다. 공개 범위는

자료의 성격과 참여자 및 배포 목적을 기준으로 정하며 학습용 사례에는 실제 비밀이나 개인정보를 넣지 않는다.

저장소의 동일성을 세 가지 질문으로 나눈다

버전 관리에서 같다는 말은 파일 내용이 같다는 뜻일 수도 있고 같은 커밋을 가리킨다는 뜻일 수도 있다. 여기에 같은 저장소에서 관리한다는 조직적 의미가 더해진다. 파일의 내용과 커밋의 정체성과 저장소의 소유권은 서로 다른 층이다. 이 구분은 저장소를 복사하거나 이름을 바꾸거나 다른 조직으로 이전할 때 중요하다.

Git 객체의 식별자는 객체 종류와 길이 및 내용을 포함하는 직렬화된 표현에서 계산된다. 파일의 경로는 blob 자체의 내용이 아니라 tree의 항목에 연결된다. 따라서 같은 바이트를 가진 두 파일이 다른 경로에서 같은 blob을 참조할 수 있다. 같은 tree를 가리키더라도 부모와 메시지 등 커밋의 다른 정보가 달라지면 커밋 식별자는 달라질 수 있다. 이 구조는 내용의 재사용과 변경 이력의 식별을 함께 가능하게 한다. [T01] [T38]

lesson-lab의 안내문을 복사해 별도 저장소에 넣었다고 하자. 안내문 바이트가 같다는 관찰은 가능하지만 원래의 리뷰와 배포 승인 및 조직 정책까지 옮겨졌다고 말할 수는 없다. 같은 코드라는 주장과 같은 관리 절차를 거쳤다는 주장을 별도로 확인해야 한다. 소스 파일을 전달할 때 커밋과 배포 기록을 함께 남기는 이유도 여기에 있다.

무결성 확인과 작성자 확인을 구분한다

해시는 예상한 바이트와 현재 바이트를 비교하는 데 유용하다. 그러나 기대하는 해시를 공격자가 함께 바꾸었다면 두 값이 일치한다는 사실만으로 신뢰할 수 없다. 해시를 어디에서 받았는지와 누가 그 값을 보증하는지가 추가로 필요하다. 암호학적 해시와 전자서명이 해결하는 질문이 다른 이유다.

커밋을 식별할 수 있다는 사실은 코드가 올바르거나 안전하다는 뜻도 아니다. 의도적으로 잘못 만든 프로그램도 정확한 식별자를 가진다. 실무에서는 대상 식별과 출처 확인 및 기능 검증을 각각 수행한다. 이 세 판단을 분리하면 서명이 있으니 테스트를 생략해도 된다는 식의 잘못된 결론을 피할 수 있다.

GitHub 저장소의 이름은 사용자가 찾기 쉬운 주소다. 자동화의 중요한 권한 판단에서는 이름이 바뀌거나 재사용되는 상황도 고려한다. 사람이 읽는 이름과 시스템이 유지하는 식별자를 함께 이해하면 저장소 이전 뒤 인증이 실패하는 원인을 더 정확히 설명할 수 있다. 구체적인 OIDC의 식별 규칙은 배포 권한을 다루는 장에서 이어서 살펴본다.

1.1 Git과 GitHub를 처음 만나다

관련 근거 [R01] [R02]

파일의 현재 모습과 바뀐 이유를 함께 보관한다

수업 도구의 최종본을 고쳤는데 예전 기능이 사라졌다고 생각해 보자. 파일 이름에 최종과 진짜최종을 붙이는 방식으로는 어느 변경이 원인인지 찾기 어렵다. 버전 관리는 변경을 작은 기록으로 남겨 비교하고 되돌릴 수 있게 만든다.

Git은 내 컴퓨터에서 파일의 이력을 관리하는 도구다. GitHub는 Git 저장소를 온라인에서 공유하고 변경을 검토하며 작업을 연결하는 서비스다. 인터넷이 끊겨도 로컬 Git 커밋은 가능하다. GitHub에 전달하려면 네트워크와 접근 권한이 필요하다.

말	쉬운 뜻	실제 예
Repository	파일과 이력을 담는 저장소	lesson-lab
Commit	선택한 변경의 기록	안내 문구 수정
Branch	다른 작업을 이어가는 이력의 이름	feature/large-text
Pull request	변경을 검토하고 합칠 제안	글자 확대 기능 검토
Remote	다른 곳에 있는 저장소의 별칭	origin

저장과 커밋과 업로드는 서로 다른 행동이다

편집기의 저장은 파일을 바꾼다. 커밋은 선택한 변경을 Git 이력에 기록한다. Push는 로컬 커밋을 원격 저장소로 보낸다. 웹 편집기에서는 서버에서 커밋하므로 로컬 저장 단계가 눈에 보이지 않을 수 있다.

기억하기: GitHub에 파일이 있다는 사실만으로 실행되는 사이트가 만들어지는 것은 아니다. 소스 보관과 사이트 배포는 별도 과정이다.

파일 하나가 이동하는 과정을 끝까지 따라가기

한 문장을 고쳤을 때 네 곳의 상태

README의 안내를 v1에서 v2로 바꾼 상황을 생각하자. 편집기에서 저장하면 작업 파일만 v2가 된다. GitHub의 화면은 원격 브랜치를 보고 있으므로 이때는 여전히 v1일 수 있다. 저장 버튼을 여러 번 누른다고 원격 상태가 달라지지는 않는다.

내가 한 행동	작업 파일	Index	로컬 커밋	원격
시작	v1	v1	v1	v1
파일 저장	v2	v1	v1	v1
add	v2	v2	v1	v1
commit	v2	v2	v2	v1
push	v2	v2	v2	v2

Index는 다음 커밋에 넣을 파일 상태다. 커밋한 뒤에도 Index는 기존 상태를 가지고 있으므로 단순히 텅 빈 상자가 됐다고 생각하지 않는다. Changes가 없다는 표현은 비교할 차이가 없다는 뜻에 가깝다.

그래서 GitHub를 백업 폴더와 똑같이 쓰면 안 된다

파일을 수정한 시각과 커밋한 시각과 원격에 보낸 시각이 다를 수 있다. 컴퓨터를 바꾸기 전에는 필요한 커밋이 원격에 있는지 확인한다. 미커밋 파일과 무시한 파일은 별도 보존 대상이다.

내 말로 설명하기

동료에게 안내 문구를 바꿨다고 말할 때 어느 브랜치에 어떤 커밋으로 기록했으며 어디까지 전달됐는지 덧붙인다. 이 세 가지를 말할 수 있으면 파일이 안 보인다는 문제도 저장과 기록과 전달 중 어디에서 멈췄는지 좁힐 수 있다.

완료 판단: 파일이 존재하는지만 보지 않는다. 의도한 브랜치의 커밋과 원격 전달 여부를 함께 본다.

실습 1.1 | 내 작업을 버전 관리 언어로 설명하기

목표는 명령어 암기가 아니다. 지금 하려는 행동이 파일 편집인지 이력 기록인지 공유인지 구분하는 것이다. 아직 계정이 없어도 진행할 수 있다.

1. 지금 관리하는 HTML 자료나 문서 하나를 고른다.
2. 바뀌기 전 상태와 바꿀 내용을 각각 한 문장으로 적는다.
3. 변경 하나에 이름을 붙인다. 예를 들어 정답 버튼 안내 수정으로 정한다.
4. 동료에게 검토를 부탁할 부분을 정한다.

5. 공개해도 되는 내용과 공개하면 안 되는 내용을 나눈다.

같은 작업을 네 가지 도구로 한다면

도구	주로 하는 일	이 책의 위치
GitHub 웹	파일 편집과 PR 검토	1-2장
GitHub Desktop	클릭으로 로컬 변경 관리	3장
Git	이력과 브랜치 직접 조작	3-6장
GitHub CLI인 gh	이슈와 PR와 API 조작	17.1절

완료 기준

확인 문제 편집기에서 저장한 파일은 바로 동료의 GitHub 화면에 나타날까?
로컬 편집만 했다면 나타나지 않는다. 필요한 변경을 커밋하고 원격으로 Push해야 한다.
이미 온라인에서 커밋한 경우에는 그 브랜치의 GitHub 화면에서 확인할 수 있다.

1.2 계정과 공개 범위와 안전한 출발

관련 근거 [R03] [R04]

계정은 작업의 소유자다

GitHub 계정을 만들고 이메일을 확인한다. 사용자 이름은 저장소 주소와 활동 기록에 연결된다. 프로필 이름과 로그인 이름은 다를 수 있다. 브라우저 로그인과 Git 명령어 인증도 서로 다르다.

초보 실습은 개인 계정의 Private 저장소에서 시작해도 된다. Public 저장소는 소스가 외부에 보인다. Private도 초대한 사람과 연결된 앱이 권한 범위 안에서 접근할 수 있으므로 비밀 보관함처럼 취급하지 않는다.

설정	권장 출발점	확인할 이유
이메일	계속 접근 가능한 주소	복구와 알림
2단계 인증	인증 앱이나 보안 키 등 설정	비밀번호 외의 보호

설정	권장 출발점	확인할 이유
복구 코드	별도의 안전한 위치에 보관	기기 분실 대응
커밋 이메일	공개 여부를 정한 주소	커밋 기록에 남음
앱 연결	필요한 저장소만 허용	접근 범위 축소

로그인 성공과 권한 보유는 다르다

로그인했어도 다른 사람의 저장소를 수정할 수는 없다. 저장소 소유자와 조직 정책과 부여된 역할이 작업 가능 범위를 정한다. 관리자 메뉴가 보이지 않으면 먼저 현재 계정과 저장소 소유자를 확인한다.

주의: 학생 이름과 성적과 연락처와 API 키는 실습에 쓰지 않는다. 공개 저장소를 Private으로 바꾸더라도 이미 복제된 사본까지 회수되는 것은 아니다.

계정과 소유자와 권한을 따로 확인하기

같은 로그인 화면에서도 작업 권한은 다르다

내 계정으로 로그인해 조직 저장소를 보고 있어도 그 저장소를 소유한다는 뜻은 아니다. 계정은 누구인가를 나타낸다. 소유자는 자원이 어느 개인이나 조직에 속하는지 나타낸다. 역할은 그 자원에서 어떤 일을 할 수 있는지 정한다.

확인 대상	예	잘못 연결하기 쉬운 생각
로그인 계정	개인 사용자	로그인만 하면 모두 수정 가능
저장소 소유자	학교 조직	화면을 열었으므로 내 저장소
저장소 역할	Read 또는 Write 등	읽을 수 있으면 설정도 변경 가능
자동화 권한	앱 설치 범위와 토큰 권한	사람이 할 수 있으면 앱도 모두 가능

실습에서 막혔을 때 읽는 순서

먼저 오른쪽 위 프로필로 현재 계정을 확인한다. 주소에서 소유자와 저장소 이름을 읽는다. 초대를 받았다면 수락 여부를 확인한다. 그 뒤 필요한 권한이 실제로 부여됐는지 확인한다. 비밀번호를 다시 입력하는 일만 반복하면 권한 문제는 해결되지 않는다.

커밋 이메일을 바꾸기 전 알아둘 점

작성자 이메일은 커밋 기록에 붙는 정보다. 이후 설정을 바꿔도 과거 커밋이 저절로 모두 바뀌지는 않는다. 공개를 원하지 않으면 GitHub 계정 설정에서 본인에게 표시되는 noreply 주소를 정확히 가져와 앞으로 사용할 주소로 설정한다.

장기 운영을 시작한다면

프로젝트를 한 사람의 개인 계정에만 의존하게 할지 조직에서 관리할지 정한다. 새 담당자에게 권한을 부여할 경로와 복구할 관리자를 마련한다. 복구 코드를 운영 문서나 공개 저장소에 그대로 적는 방식은 피한다.

계정 복구와 저장소 접근 복구와 외부 배포 복구는 같은 절차가 아니다. 필요한 담당과 위치를 각각 기록한다.

실습 1.2 | 내 계정의 기본 상태 점검

- 1. 프로필 메뉴의 Settings에서 로그인 계정을 확인한다.
- 2. Emails에서 이메일 인증 상태와 공개 설정을 확인한다.
- 3. Password and authentication에서 2단계 인증을 설정한다.
- 4. 복구 코드를 안전하게 보관하고 어디에 두었는지 기억한다.
- 5. Applications에서 연결된 앱과 접근 가능한 저장소를 확인한다.

유료 기능을 만나면 먼저 읽을 세 가지

요금 이름만으로 기능 제공 여부를 판단하지 않는다. 개인 계정인지 조직인지와 Public인지 Private인지가 함께 영향을 준다. Codespaces와 Actions와 대용량 저장소와 AI 기능은 사용량과 결제 설정도 확인한다.

판단 질문	확인 위치의 예
어떤 계정에 비용이 붙는가	개인 또는 조직의 Billing
현재 어느 기능을 켜는가	해당 저장소의 Settings

공개 전 확인 습관

현재 파일만 보지 말고 이력도 확인한다. 과거에 올린 비밀이 남아 있을 수 있다. 공개 전에는 별도 검토를 거친다. 토큰 노출은 13.3절의 사고 대응 순서대로 처리한다.

확인 문제 GitHub 비밀번호를 팀원에게 알려 주면 협업 설정이 쉬워질까?
개인 계정을 공유하지 않는다. 각자 계정을 만들고 저장소 초대나 조직의 팀 권한으로 협업한다. 사람마다 접근을 회수하고 변경 책임을 확인할 수 있어야 한다.

1.3 웹에서 첫 저장소 만들기

관련 근거 [R01] [R05]

설치하지 않고 첫 기록을 남긴다

이 책에서는 lesson-lab을 공통 연습 저장소로 쓴다. 앞으로 나오는 YOUR-ID는 자신의 GitHub 사용자 이름으로 바꾼다. example 계정에 실제로 접속해 수정하려는 뜻이 아니다.

오른쪽 위 생성 메뉴에서 New repository를 연다. 소유자를 자신의 계정으로 선택하고 이름을 lesson-lab으로 정한다. 학습용 설명을 적고 공개 범위를 선택한다. 새 연습 저장소에는 README를 포함해 시작한다.

화면 항목	넣을 값의 예	의미
Owner	자신의 계정	누가 소유하는가
Repository name	lesson-lab	저장소 이름
Description	수업 도구 GitHub 연습	간단한 설명
Visibility	Private 또는 Public	접근 범위
README	생성 선택	첫 안내 문서

저장소 화면 읽기

Code는 파일과 브랜치를 보는 곳이다. Issues는 작업과 문제를 기록한다. Pull requests는 변경 제안을 검토한다. Actions는 자동 실행 기록을 보여 준다. Settings는 권한이 있는 사람이 설정을 관리한다.

파일 목록 위의 브랜치 이름을 확인한다. 기본 브랜치 이름은 저장소에 따라 다를 수 있다. 책에서는 main을 사용한다. 화면에 master 등 다른 이름이 있다면 자신의 저장소 이름을 기준으로 읽는다.

확인: 주소의 YOUR-ID/lesson-lab이 자신의 계정과 일치해야 한다. 조직 저장소를 실수로 만들지 않도록 Owner를 먼저 본다.

README를 만들고 첫 커밋을 확인하는 이유

빈 저장소와 첫 커밋이 있는 저장소

저장소 이름을 만들었다고 곧바로 파일 이력이 생기는 것은 아니다. 새 저장소에 README를 포함하면 첫 파일과 첫 커밋을 가진 상태로 시작하기 쉽다. 로컬에서 이미 Git 이력을 만들었다면 원격에도 별도 첫 커밋을 만드는 선택이 추가 통합을 요구할 수 있다.

출발 상태	먼저 할 일
웹에서 새로 시작	README를 포함해 생성
원격에 이미 파일이 있음	Clone해서 이어서 작업
로컬 이력만 있음	빈 원격을 준비해 연결

Commit changes 창을 읽는 법

첫 입력은 이번 변경을 요약하는 메시지다. 파일 이름을 다시 정하는 입력칸이 아니다. 본문이 따로 있으면 상세 이유를 적을 수 있다. 기록할 브랜치를 선택하는 부분에서는 현재 브랜치에 직접 넣는지 새 브랜치로 보내는지 확인한다.

첫 기록에서 두 화면을 비교한다

README 파일 화면에서는 현재 파일 내용을 본다. 커밋 상세 화면에서는 이전 상태와 비교해 어떤 줄을 추가했는지 본다. 같은 문장이라도 두 화면의 질문이 다르다. 파일 보기의 질문은 지금 무엇이 있는가다. 커밋 보기의 질문은 이 기록에서 무엇이 바뀌었는가다.

뜻대로 되지 않을 때

파일을 편집했는데 목록에서 옛 내용이 보이면 선택한 브랜치부터 확인한다. PR을 열기만 했고 아직 합치지 않았다면 main은 옛 상태일 수 있다. 다른 탭에서 같은 이름의 저장소를 보고 있지 않은지도 확인한다.

첫 실습의 목표는 파일을 많이 올리는 것이 아니다. 파일 내용과 커밋 메시지와 브랜치를 서로 구분하는 것이다.

실습 1.3 | README에 첫 안내 작성

1. lesson-lab에서 README.md를 연다.
2. 연필 아이콘이나 편집 메뉴를 누른다.
3. 아래 내용을 입력한다. Markdown 표시는 그대로 입력한다.
4. Commit changes를 누르고 변경 이유를 짧게 적는다.
5. 이번 입문 실습에서는 main에 직접 커밋한다. 다음 장부터 작업 브랜치를 쓴다.

수업 도구 연습실

이 저장소의 목적

GitHub의 변경 기록과 협업을 연습한다.

현재 기능

- 안내 문서
- 수업용 HTML 예제

실행 방법

index.html을 브라우저로 연다.

결과를 확인하는 방법

README 내용이 바뀌었는지 본다. 커밋 목록에서 방금 적은 메시지를 누른다. 변경 화면의 추가된 줄을 확인한다. 파일 내용이 바뀐 것과 변경 기록이 생긴 것을 모두 확인해야 끝난다.

확인 문제 ZIP 다운로드에는 협업에 필요한 Git 이력이 항상 들어 있을까?

일반적인 소스 ZIP에는 .git 이력이 들어 있지 않다. 계속 개발하려면 Clone을 사용한다. ZIP은 그 시점의 파일을 받아 보는 데 적합하다.

2장 웹에서 문서와 변경 제안

브라우저에서 작은 변경을 제안하고 읽기 쉬운 문서를 만든다. 실제 화면에서 편집 위치를 찾고 검토한 버전의 링크를 공유한다.

문서는 실행 경로를 안내하는 인터페이스다

README는 프로젝트를 홍보하는 표지에 그치지 않는다. 사용자가 무엇을 준비하고 어느 명령을 실행하며 어떤 결과를 기대해야 하는지 알려 주는 인터페이스다. 실행 안내가 실제 코드와 어긋나면 소프트웨어의 사용 가능성이 낮아진다. 따라서 README 변경도 코드 변경과 같은 방식으로 검토하고 확인할 수 있다.

lesson-lab에서 실행 명령을 바꾸는 PR이라면 문서와 설정 파일이 함께 바뀌어야 할 수 있다. 문서만 수정해 존재하지 않는 명령을 안내하거나 설정만 바꾸고 기존 안내를 남기면 계약이 깨진다. 검토자는 문장의 자연스러움뿐 아니라 실제 실행 가능성을 확인한다. 이 원리는 API 문서와 환경 변수 설명 및 배포 안내에도 적용된다.

PR은 두 브랜치 사이의 제안이며 커밋 자체가 아니다

Pull request는 어떤 변경을 어느 기준에 통합할지 논의하는 객체다. Base는 변경을 받을 기준이며 Head는 제안된 작업의 끝이다. PR에는 커밋과 차이 및 대화와 검사 결과가 연결된다. 브랜치를 이동시키면 PR의 내용도 달라질 수 있으므로 검토한 시점의 커밋을 알아야 한다.

GitHub의 PR 차이와 로컬에서 두 커밋을 직접 비교한 차이는 비교 기준이 달라 보일 수 있다. 공통 조상에서 제안 브랜치까지의 변화와 두 최신 스냅샷 사이의 차이는 같은 질문이 아니다. 기능 브랜치가 기준 브랜치보다 뒤쳐져 있을 때 이 차이가 중요해진다. 차이를 해석하기 전에 무엇과 무엇을 비교했는지 확인한다. [T03]

PR 본문은 문제와 변경 및 검증을 연결한다. 검색 결과 수가 갱신되지 않는 문제를 고쳤다면 실패 조건과 수정 지점 및 재현 결과를 함께 적는다. 검토자가 구현자의 전체 작업 과정을 추측하지 않고 핵심 판단을 확인할 수 있게 하는 것이 목적이다. 긴 템플릿을 채우는 것보다 주장과 증거가 일치하는지가 중요하다.

비교 화면의 기준점이 검토 대상을 결정한다

변경 제안은 항상 무엇과 무엇을 비교하는지에 의존한다. 기능 브랜치의 끝과 현재 main의 끝을 직접 비교하는 것과 두 브랜치가 갈라진 공통 조상부터 기능 브랜치의 끝을 비교하는 것은 다른 질문이다. 전자는 현재 두 결과물의 차이에 가깝고 후자는 기능 브랜치 쪽에서 도입한 변경을 이해하는 데 유용하다.

커밋 집합을 읽는 명령에서는 A..B가 B에서 도달할 수 있지만 A에서는 도달할 수 없는 커밋을 뜻한다. A...B는 두 쪽 중 한쪽에서만 도달할 수 있는 커밋의 집합이다. 반면 git diff A..B는 공통 조상과 B의 결과를 비교하는 용법이다. 점의 개수만 외우면 같은 표기를 모든 명령에서 같은 의미로 해석하기 쉽다. 명령이 이력 집합을 다루는지 파일 상태를 비교하는지 먼저 구분한다. [T19] [T03]

관찰하려는 대상	예시 명령	해석
기능 쪽에만 있는 커밋	git log main..feature	main에서 도달하지 못하는 기능 쪽 이력
양쪽의 독립적인 커밋	git log --left-right main...feature	각 쪽에만 속한 커밋의 구분
두 끝점의 파일 차이	git diff main feature	현재 스냅샷 사이의 차이
기능 쪽에서 도입한 파일 변화	git diff main...feature	공통 조상과 feature의 비교

main에 문서 개편이 먼저 들어간 상태에서 feature는 검색 기능만 추가했다고 하자. 두 끝점을 비교하면 main의 문서 변경을 feature가 되돌리는 것처럼 보이는 부분이 생길 수 있다. 이때 비교 화면의 기준점을 알면 실제 의도와 화면의 차이를 구분할 수 있다. 검토자가 먼저 확인할 것은 변경 줄의 수보다 비교의 출발점이다.

문서와 구현을 같은 변경 계약에 묶는다

README의 실행 명령이 코드의 실제 시작점과 다르면 문서가 존재해도 재현은 실패한다. 문서의 예시는 실행 가능한 계약의 한 표현으로 볼 수 있다. 필요한 환경과 입력 및 기대 결과를 포함한 예시는 처음 읽는 사람이 성공 여부를 판정하게 한다. 설치 명령이 있다는 사실과 그 명령이 현재 판본에서 작동한다는 사실은 구분해야 한다.

PR에서 함수의 입력 이름을 바꾸었다면 호출 코드와 예시 및 오류 메시지에도 영향이 있는지 확인한다. 문서 수정과 코드 수정을 별개의 완료 목록으로만 관리하면 변경의

연결이 끊어질 수 있다. 변경 이유 하나에 연결되는 자료를 함께 검토하는 편이 적은 수정으로도 계약의 일관성을 높인다.

2.1 웹에서 브랜치와 PR 경험하기

관련 근거 [R01] [R06]

원본 작업 흐름에서 새 길을 만든다

main이 현재 기준이라면 feature/readme-guide는 안내 문서 수정 작업의 이름이다. 브랜치는 전체 폴더를 복제한 별도 저장소가 아니다. 어느 커밋을 가리키는지 나타내는 이름이며 작업이 진행되면 가리키는 위치가 바뀐다.

브랜치 선택 메뉴에서 main을 기준으로 feature/readme-guide를 만든다. 그 브랜치에서 README를 고친다. main을 다시 선택해 보면 아직 내용이 바뀌지 않은 것을 확인할 수 있다.

PR은 어느 방향으로 합칠지 지정한다

항목	이 실습의 값	읽는 방법
Base	main	변경을 받을 곳
Compare 또는 Head	feature/readme-guide	변경을 가져올 곳
Title	README 실행 안내 보강	무엇을 바꾸는가
Description	이유와 확인 방법	어떻게 검토할까

실수 방지: PR 화면에서 변경 파일과 방향을 확인한다. 제목이 맞아도 Base가 틀리면 원치 않는 브랜치로 제안할 수 있다.

PR 방향을 문장으로 읽기

어느 쪽이 바뀌어야 하는지부터 말한다

작업 브랜치의 안내 수정이 main에 들어가야 한다면 변경을 받는 쪽이 main이다. 그래서 Base를 main으로 선택한다. Compare나 Head는 수정한 작업 브랜치다. PR을 만들기 전에 이 변경이 어느 곳에 들어가야 하는지 문장으로 읽으면 방향을 확인하기 쉽다.

물어볼 질문	확인할 값
변경을 받을 저장소는 어디인가	Base repository
그 저장소의 어느 브랜치인가	Base branch
변경을 제공하는 곳은 어디인가	Head repository와 branch
실제로 바뀌는 파일은 무엇인가	Files changed

PR이 생긴 뒤 커밋을 더 만들면

같은 Head 브랜치에 커밋을 추가해 Push하면 열린 PR에도 그 변경이 이어질 수 있다. 매번 새 PR을 만들 필요는 없다. 다만 수정 후에는 파일 차이와 검사와 검토 상태를 다시 확인한다. 기존 승인 이후 변경을 추가하면 재검토 조건이 적용될 수 있다.

브랜치 삭제를 이해한다

병합한 작업 브랜치를 정리하는 것은 파일을 main에서 지우는 행동과 다르다. 병합 결과는 main의 이력에서 확인한다. Squash나 Rebase 방식이면 원래 작업 커밋의 ID가 그대로 남는다고 가정하지 않는다. 다음 작업은 최신 main에서 새 브랜치를 만들어 시작하는 편이 목적을 구분하기 쉽다.

거꾸로 제안했다면

병합하기 전에 Base와 diff를 바로잡는다. 이미 다른 사람이 검토한 큰 변경이라면 방향을 바꾼 사실도 알린다. 제목을 바꾸는 것만으로 변경 방향이 바뀌지는 않는다.

PR 본문에는 완료했다고 주장한 기능보다 검토자가 확인할 결과와 방법을 적는다. 아직 끝내지 못했다면 Draft 상태와 남은 일을 명확히 쓴다.

실습 2.1 | 내 첫 Pull request

1. feature/readme-guide에서 README에 사용 대상을 한 줄 더 적는다.
2. 브랜치에 커밋한다. 메시지는 사용 대상 안내 추가로 정한다.
3. Pull requests에서 New pull request를 선택한다.
4. Base와 Compare를 위 표대로 지정하고 Files changed를 읽는다.
5. PR을 만들고 대화에 확인한 내용을 적는다.
6. 실습 저장소에서 병합한 뒤 main에 새 문장이 나타나는지 확인한다.

PR 본문 예시

바꾼 내용

README에 사용 대상과 실행 방법을 보강했다.

바꾼 이유

처음 방문한 교사가 시작 위치를 알기 어렵다.

확인 방법

- README 화면에서 안내를 읽었다.
- main과 작업 브랜치의 차이를 비교했다.

병합 버튼은 설정과 권한과 검토 조건에 따라 달라진다. 보이지 않으면 무조건 오류로 생각하지 말고 PR의 상태와 검사 결과를 읽는다. 개인 저장소에서 자신이 만든 PR은 스스로 병합할 수 있지만 자신의 PR에 대한 승인이 팀의 필수 승인 조건을 대신하지는 못한다.

확인 문제 브랜치를 만들자마자 main의 파일도 달라질까?

아니다. 생성 시점에는 같은 커밋에서 시작할 수 있다. 그 뒤 작업 브랜치에 커밋해도 main은 따로 이동시키거나 병합하기 전까지 그대로다.

2.2 README와 Markdown과 파일 관리

관련 근거 [R05] [R07]

읽는 사람에게 첫 실행까지 안내한다

좋은 README는 프로젝트를 만든 사람의 기억을 대신한다. 목적과 대상과 실행 방법과 현재 한계와 도움 요청 방법을 적는다. 프로그램이 아니라 교육 자료 묶음이어도 문서 구조는 같다.

문법	쓰는 곳	입력 예
#	문서 제목	# 수업 실험실
##	하위 제목	## 시작하기
-	순서 없는 목록	- HTML 내려받기
[이름](주소)	링크	[사용법](docs/guide.md)

문법	쓰는 곳	입력 예
![설명](경로)	이미지	![예시](assets/demo.png)
```	코드 구역의 시작과 끝	언어 이름을 뒤에 추가

## 파일 배치와 링크

폴더 이름은 역할이 드러나게 정한다. docs는 설명과 운영 기록에 쓴다. assets는 이미지와 스타일 파일에 쓴다. src는 애플리케이션 코드가 커졌을 때 도입한다. 작은 예제에 불필요하게 복잡한 구조를 만들 필요는 없다.

상대 경로는 현재 문서를 기준으로 찾는다. README에서 assets/demo.png를 찾는 경로와 docs/guide.md에서 같은 그림을 찾는 경로는 다르다. 대소문자가 다른 파일 이름은 배포 환경에서 깨질 수 있다.

이미지 설명에는 그림이 전달하는 의미를 적는다. 학생의 얼굴과 이름이 포함된 캡처는 공개 자료에 그대로 넣지 않는다.

## 상대 경로를 폴더 위치로 계산하기

### 링크는 현재 파일이 있는 위치에서 출발한다

루트 README에서 docs/guide.md를 여는 경로와 docs 폴더 안 문서에서 그림을 여는 경로는 다르다. 링크가 깨졌을 때 주소 전체를 임의로 바꾸기보다 출발 파일과 대상 파일의 위치를 먼저 그려 본다.

출발 파일	대상 파일	상대 경로
README.md	docs/guide.md	docs/guide.md
README.md	assets/demo.png	assets/demo.png
docs/guide.md	assets/demo.png	../assets/demo.png
docs/guide.md	README.md	../README.md

../는 한 단계 위 폴더로 간다는 뜻이다. 웹에서 보이는 링크 글자와 실제 링크 주소는 다를 수 있다. 링크가 안내라는 말로 표시되더라도 괄호 안의 경로가 실제 이동 위치다.

## Preview가 중요한 이유

Markdown 원문은 작성 문법을 보여 준다. Preview는 읽는 사람이 볼 표시를 확인하게 한다. 코드 구역을 닫지 않으면 뒤의 설명까지 코드처럼 표시될 수 있다. 표와 목록도 원문만 보고 끝내지 말고 표시 결과를 확인한다.

## 배포 문서와 원본을 함께 유지

수업 자료 PDF를 수정했다면 어떤 원본에서 만들었는지도 기록한다. 문장 단위 차이를 보고 싶으면 Markdown이나 HTML 원본을 관리한다. 내려받는 사람이 바로 쓸 파일은 Release 등 배포 경로에서 판본과 함께 안내할 수 있다.

대소문자 차이를 놓치지 않는다. 내 컴퓨터에서는 열리던 Demo.png 링크가 배포 환경의 demo.png와 다르게 취급될 수 있다.

## 실습 2.2 | 다른 사람이 읽을 안내 완성

1. Add file의 Create new file에서 docs/guide.md를 만든다.
2. 실행 방법과 알려진 문제를 한 항목씩 적는다.
3. README에 docs/guide.md로 이동하는 상대 링크를 단다.
4. Preview에서 제목과 링크가 제대로 표시되는지 본다.
5. 변경을 새 브랜치로 커밋하고 PR로 합친다.

### ## 빠른 시작

1. Code 메뉴에서 Download ZIP을 선택한다.
2. 압축을 푼다.
3. index.html을 연다.

### ## 사용 안내

[자세한 설명] (docs/guide.md)

### ## 알려진 한계

인터넷 없이도 열리지만 외부 링크는 연결이 필요하다.

## 문서를 유지하는 규칙

실행 방법이 바뀌는 PR에는 README 수정도 포함한다. 링크는 실제로 눌러 본다. 아직 구현하지 않은 기능은 계획으로 표시한다. 소스와 사용법이 다른 상태를 오래 두지 않는다.

확인 문제 PDF와 DOCX도 Git에 넣으면 글자 단위 변경을 편하게 비교할 수 있을까?  
일반적인 Git diff에서는 바이너리 파일의 세부 문장 차이를 보기 어렵다. 편집 원본을 텍스트 형식으로 함께 관리하거나 별도의 비교 도구를 쓴다. 큰 배포 파일은 Release 첨부도 검토한다.

## 2.3 실제 화면에서 저장소와 편집 위치 찾기

관련 근거 [R73] [R74] [R75]

### 화면을 열면 소유자와 브랜치부터 읽는다

동일한 파일 이름이라도 다른 사람의 저장소이거나 다른 브랜치일 수 있다. 주소의 OWNER/REPO와 파일 목록 위 브랜치 선택을 먼저 본다. 그 뒤 파일을 열어 편집한다. 아래는 GitHub 공식 안내에 사용된 실제 화면이다. 예시 계정과 파일을 그대로 수정하는 뜻이 아니다.

GitHub 공식 문서의 저장소 생성 화면 / R73

화면의 표시	읽는 법
Owner	새 저장소의 소유 주체
Repository name	주소에 들어갈 저장소 이름
초록색 사용 가능 표시	그 소유자 아래에서 이름을 사용할 수 있음

### 파일을 고치는 위치

GitHub 공식 문서의 파일 편집 버튼 / R73

파일 상세 화면의 연필 버튼은 웹 편집을 연다. Raw는 원문 보기다. Blame은 줄의 변경 근거를 추적한다. 편집 버튼이 없거나 Fork 안내가 나온다면 현재 권한과 저장소 상태부터 확인한다.

화면의 주황색 테두리는 GitHub 공식 문서가 강조한 위치다. UI의 순서와 문구는 바뀔 수 있다. 버튼 모양보다 기능 이름과 현재 대상을 함께 확인한다.

## 검색 결과와 링크가 가리키는 대상을 검증

### 찾은 파일이 현재 작업 대상인가

검색 결과가 맞는 문장을 보여 줘도 소유자와 저장소와 브랜치가 다른 고칠 곳이 다르다. 검색 결과를 연 뒤 주소와 파일 경로를 다시 읽는다. 비슷한 이름의 Fork와 원본이 함께 있을 때 특히 이 순서가 필요하다.

확인할 위치	실제로 답하는 질문
OWNER/REPO	누구의 어느 저장소인가
브랜치 선택	어느 작업 흐름의 파일인가
파일 경로	같은 이름 중 어느 파일인가
커밋 ID	어느 시점의 내용인가
줄 번호	그 내용의 어디를 가리키는가

### 검색 결과가 없을 때

철자가 틀렸다고 바로 판단하지 않는다. 저장소 범위를 잘못 지정했는지와 파일 경로 조건이 지나치게 좁은지 확인한다. 접근 권한과 검색 대상의 조건도 살핀다. 먼저 repo 조건만 남겨 찾은 뒤 path와 내용 조건을 하나씩 추가하면 어떤 조건에서 사라지는지 알 수 있다.

### 공유 목적에 맞는 주소

작업 중인 최신 README로 안내할 때는 브랜치 주소가 편하다. 버그 조사와 논문과 검토 기록에는 특정 커밋 주소를 사용한다. 영구 링크는 내용을 고정하는 방식이며 저장소의 공개 여부와 접근 가능성까지 보장하는 방식은 아니다.

### 웹 편집을 저장하기 직전

바뀐 줄을 다시 읽는다. 목적이 같은 변경만 포함됐는지 확인한다. 새 브랜치를 만들었다면 이름을 기록하고 PR의 Base와 Head를 확인한다. 편집 성공 메시지 뒤에 원하는 브랜치의 실제 파일을 다시 열어 내용이 반영됐는지 본다.

링크를 공유할 때 주소 안에 비밀이 있는지 확인한다. 비공개 파일의 주소를 전달한다고 상대방에게 읽기 권한이 자동으로 생기지는 않는다.

### 실습 2.3 | 파일을 찾고 정확한 버전을 공유

1. 자신의 lesson-lab에서 README.md를 연다.
2. 브랜치 이름을 확인한 뒤 연필 버튼을 찾는다.
3. 편집을 시작하지 않고 원래 파일 보기로 돌아온다.
4. 입력칸에 커서가 없는 상태에서 y를 눌러 영구 링크를 만든다.
5. 주소의 브랜치 이름 부분이 커밋 ID로 바뀌었는지 본다.
6. 줄 번호를 선택해 특정 부분을 가리키는 링크를 만든다.

### 코드 검색으로 범위를 좁힌다

아래 검색문은 GitHub의 코드 검색용이다. Issue 검색 문법과 완전히 같지는 않다. YOUR-ID는 본인 계정으로 바꾼다.

```
repo:YOUR-ID/lesson-lab path:README.md
repo:YOUR-ID/lesson-lab path:.github/ language:YAML
repo:YOUR-ID/lesson-lab content:answer
```

### 링크를 고르는 기준

최신 안내를 공유하려면 브랜치 링크가 편하다. 연구 보고서나 오류 근거에는 특정 커밋의 영구 링크를 쓰면 같은 파일 상태를 가리킬 수 있다. 저장소가 삭제되거나 접근 권한이 사라지면 영구 링크도 열리지 않을 수 있다.

확인 문제 main의 파일 주소를 보고서에 넣으면 언제나 같은 문장이 열릴까?  
main이 움직이면 파일 내용도 달라질 수 있다. 검토한 상태를 고정하려면 커밋 ID를 포함한 링크를 쓴다.

## 3장 내 컴퓨터의 작업 환경

Desktop과 편집기와 터미널의 역할을 연결한다. Clone한 실제 폴더에서 작업하고 로컬 기록이 원격에 도착했는지 확인한다.

### 화면이 달라도 같은 저장소 상태를 다룬다

GitHub Desktop과 VS Code 및 터미널은 서로 다른 인터페이스다. 같은 로컬 저장소를 열고 있다면 공통된 Git 객체와 Index 및 작업 트리를 다룬다. 한 도구에서 스테이징한 변경은 다른 도구에서도 관찰할 수 있다. 다만 갱신 시점과 표시 방식이 다를 수 있으므로 상태가 의심스러우면 현재 경로와 브랜치를 먼저 확인한다.

초보자가 자주 겪는 문제는 명령 자체보다 작업 위치다. 같은 이름의 lesson-lab 폴더를 두 곳에 만들면 편집기와 터미널이 서로 다른 저장소를 보고 있을 수 있다. `git rev-parse --show-toplevel`로 저장소 루트를 확인하면 비교 기준을 잡을 수 있다. 경로 확인은 사소한 준비가 아니라 재현 가능한 작업의 일부다.

### 구성 값에는 범위가 있다

Git 설정에는 시스템과 사용자 및 저장소 범위가 있다. 특정 저장소의 설정이 사용자 기본값을 덮어쓸 수 있다. 회사 프로젝트와 개인 프로젝트에서 작성자 이메일이 다른 경우 저장소 단위 설정을 사용하는 이유다. `git config --show-origin --get user.email`처럼 값과 출처를 함께 확인하면 예상과 다른 설정의 원인을 좁힐 수 있다.

커밋의 author와 committer는 다른 역할이다. 원래 변경을 작성한 사람과 현재 커밋 객체를 만든 사람이 다를 수 있다. 작성자 이름을 지정하는 것은 신원 인증과 동일하지 않다. 나중에 서명을 배우더라도 서명이 코드의 품질을 보증하는 것은 아니라는 점을 기억해야 한다.

재현 가능한 개발 환경에는 런타임과 의존성 및 실행 명령이 포함된다. 저장소를 Clone했다는 사실만으로 모든 실행 환경이 준비되는 것은 아니다. README에는 설치와 검사 및 실행의 순서를 적고 필요한 환경 변수의 이름을 설명한다. 실제 비밀값은 저장소 밖에서 관리한다.

## 줄바꿈 차이는 실제 내용 차이로 기록될 수 있다

화면에서 같은 문장으로 보여도 파일의 바이트는 다를 수 있다. 운영체제와 편집기가 사용하는 줄바꿈이 다르면 거의 모든 줄이 바뀐 것처럼 나타나기도 한다. .gitattributes는 경로별 텍스트 처리와 줄바꿈 같은 저장소 정책을 표현한다. text와 eol 설정은 작업 트리의 표현과 저장되는 표현을 연결하는 규칙이다. [T21]

```
* text=auto
*.sh text eol=lf
*.png -text
*.pdf -text
```

이 예시는 텍스트를 자동 판별하고 셸 스크립트의 줄바꿈을 LF로 두며 이미지와 PDF를 텍스트 정규화에서 제외한다. 기존 저장소에 도입할 때는 기능 수정과 한 번의 정규화 변경을 나누는 편이 검토하기 쉽다. 대규모 줄바꿈 변화와 실제 로직 수정을 한 PR에 섞으면 중요한 차이를 찾기 어려워진다.

정규화는 모든 파일을 같은 방식으로 바꾸는 작업이 아니다. 바이너리 파일에 텍스트 처리를 적용하면 의미 없는 변형을 만들 수 있다. 팀의 편집기 설정과 Git의 경로별 정책이 충돌하지 않는지도 확인한다. 문제가 생기면 파일을 계속 다시 저장하기보다 어떤 단계에서 바이트가 바뀌는지 살핀다.

## 실행 환경은 의존성 목록보다 넓다

같은 소스와 같은 패키지 목록을 사용해도 운영체제와 CPU 구조 및 시스템 라이브러리가 다르면 결과가 달라질 수 있다. 시간대와 locale 및 기본 문자 인코딩도 출력에 영향을 준다. 정렬된 이름 목록이 다른 순서로 보이거나 날짜가 하루 어긋나는 현상은 로직보다 환경 차이에서 시작될 수 있다.

재현 기록에는 영향을 줄 수 있는 변수만 우선 포함한다. 모든 환경 변수를 덤프하면 비밀을 노출할 수 있고 분석에 필요한 정보가 묻힌다. 런타임 버전과 패키지 잠금 파일 및 실행 명령과 입력 자료를 기본으로 남긴다. 문제가 특정 환경에서만 나타난다면 그 차이를 추가한다. 기록의 목적은 양을 늘리는 것이 아니라 실패를 다시 만들 수 있게 하는 것이다.

로컬에서 실행한 명령이 CI에서 실패하면 명령 자체뿐 아니라 작업 디렉터리와 파일의 대소문자 및 실행 권한을 비교한다. 로컬 파일 시스템이 허용한 경로 표기가 다른 환경에서는 실패할 수 있다. 개별 환경을 공유하는 것은 편집 화면을 같게 만드는 일보다 실행 전제를 명시하는 일에 가깝다.

### 3.1 GitHub Desktop으로 내 컴퓨터 연결

관련 근거 [R08]

#### 화면으로 Git 작업 흐름을 익힌다

Windows에서는 GitHub Desktop을 설치하고 브라우저를 통해 계정에 로그인한다. 공식 배포처는 [desktop.github.com](https://desktop.github.com)이다. macOS에서도 사용할 수 있다. Linux용 공식 GitHub Desktop 배포를 전제로 안내하지 않는다.

File의 Clone repository에서 lesson-lab을 선택한다. Local path는 내가 찾기 쉬운 작업 폴더로 정한다. 압축 파일을 받는 것과 달리 Clone은 Git 이력과 원격 연결을 포함한다.

화면	관찰할 내용
Current repository	지금 작업하는 저장소
Current branch	지금 변경하는 브랜치
Changes	아직 커밋하지 않은 변경
History	로컬 커밋의 기록
Fetch origin	원격 변경 정보를 가져오기
Push origin	로컬 커밋을 원격에 보내기

#### 파일을 고치는 프로그램은 별도로 쓴다

Desktop은 주로 Git 작업을 관리한다. 텍스트 편집기나 VS Code에서 파일을 바꾸고 저장한 뒤 Desktop으로 돌아와 차이를 읽는다. Changes에 보이는 파일을 전부 무심코 선택하지 않는다.

폴더 주의: 같은 이름의 ZIP 폴더와 Clone 폴더를 함께 두면 다른 파일을 편집하기 쉽다. Repository의 Show in Explorer 등으로 실제 위치를 확인한다.

## Desktop의 Changes와 History를 연결해서 보기

### 작업을 고른 뒤 기록하고 그 기록을 보낸다

Changes는 현재 파일과 기록 기준 사이의 차이를 보여 준다. History는 이미 만들어진 커밋을 보여 준다. Changes에서 파일이 사라졌다고 파일이 삭제됐다는 뜻은 아니다. 커밋으로 기록돼 더 이상 미커밋 차이가 아닐 수 있다.

시점	Changes에서 볼 것	History에서 볼 것
파일 수정 직후	바뀐 파일과 줄	아직 이전 커밋
커밋 직후	남은 미커밋 변경	방금 만든 메시지
Push 직후	보통 차이 없음	같은 로컬 기록 유지

Push의 결과는 History만으로 모두 확인하기 어렵다. 로컬에 있던 커밋을 원격으로 보냈는지 GitHub 웹의 같은 브랜치에서도 확인한다. 반대로 웹에서 만든 커밋은 Fetch와 통합 전까지 로컬 History에 없을 수 있다.

### 파일을 고쳤는데 Changes가 비어 있다면

편집기에서 저장했는지 확인한다. Desktop의 Current repository와 실제 파일 경로를 비교한다. 무시 규칙에 걸린 파일인지도 확인한다. 원래 ZIP 폴더와 Clone 폴더가 나란히 있으면 이름이 같아 혼동하기 쉽다.

### 커밋 전에 읽을 세 가지

선택한 파일 이름을 읽는다. 그 파일의 줄 차이를 읽는다. Summary가 그 선택 전체를 설명하는지 확인한다. 안내 문구 수정이라는 메시지만데 프로그램 전체와 설치 파일이 함께 선택됐다면 범위를 다시 나눈다.

기능을 되돌리고 싶을 때 브랜치 삭제나 저장소 폴더 삭제부터 하지 않는다. 먼저 해당 변경이 미커밋인지 커밋됐는지 원격에 공유됐는지 판단한다.

## 실습 3.1 | 로컬 커밋과 Push 구분

1. Desktop에서 lesson-lab의 main을 연다.
2. Current branch에서 feature/local-guide를 만든다.
3. 편집기로 README에 로컬 작업 안내 한 줄을 추가한다.
4. Desktop의 Changes에서 바뀐 줄과 선택 파일을 확인한다.

5. Summary를 적고 Commit to 해당 브랜치를 누른다.
6. Publish branch 또는 Push origin을 누른다.
7. GitHub 웹에서 같은 브랜치를 선택해 커밋을 확인한다.

### 자주 생기는 혼동

현상	먼저 확인할 것
파일을 고쳤는데 Changes가 비어 있음	저장 여부와 실제 폴더
커밋했는데 웹에 안 보임	Push 여부와 웹의 브랜치
Push가 거절됨	인증과 권한과 원격 새 커밋
예상 못 한 파일이 수백 개	빌드 산출물과 .gitignore

### 새 PC에서도 이어서 작업하려면

Push한 커밋은 다른 PC에서 Clone한 뒤 받을 수 있다. Push하지 않은 로컬 커밋과 미커밋 파일은 자동으로 나타나지 않는다. 폴더 동기화 서비스와 Git의 역할을 혼동하지 않는다.

확인 문제 Commit 버튼을 누른 뒤 인터넷 연결이 끊기면 기록이 사라질까?  
 커밋은 로컬에 남는다. 연결이 복구되면 Push할 수 있다. 다만 로컬 디스크 자체를 잃으면  
 아직 Push하지 않은 이력은 별도 백업이 없을 때 복구하기 어렵다.

## 3.2 VS Code와 터미널을 준비한다

관련 근거 [R02] [R09] [R72]

### 사용할 창과 작업 폴더를 먼저 확인한다

명령어 실습은 Windows의 Git Bash를 기준으로 한다. macOS와 Linux의 셸에서도 대부분 같다. PowerShell은 인용과 줄 연결 방식이 다를 수 있으므로 이 책의 여러 줄 명령을 임의로 섞지 않는다. 명령 앞의 달러 표시는 입력하지 않는다. 책에는 입력할 명령만 실는다.

GitHub Desktop에 포함된 내부 Git과 별도로 터미널에서 쓸 Git이 필요할 수 있다. git-scm.com에서 운영체제에 맞게 설치한 뒤 새 터미널을 열어 버전을 확인한다. VS Code는 편집기다. Git과 GitHub CLI는 각각 설치하고 인증한다.

```
git --version
gh --version
pwd
```

gh가 없다는 오류가 나더라도 Git 자체는 사용할 수 있다. gh 실습 전에 cli.github.com에서 설치한다. pwd는 현재 폴더를 확인하는 Git Bash 명령이다.

### 커밋 작성자 정보 설정

```
git config --global user.name 'YOUR-NAME'
git config --global user.email 'YOUR-COMMIT-EMAIL'
git config --global init.defaultBranch main
git config --global --get user.name
git config --global --get user.email
```

구분: user.name과 user.email은 커밋에 붙는 작성자 정보다. GitHub 로그인 비밀번호나 인증 설정이 아니다. 이메일 공개가 걱정되면 GitHub Emails 설정의 본인용 noreply 주소를 정확히 복사해 사용한다.

### 설치와 PATH와 Git 설정의 구별

#### 명령어를 찾는 일과 계정 인증은 다르다

git --version을 실행했을 때 버전이 나오면 현재 터미널이 Git 프로그램을 찾고 실행한 것이다. GitHub 로그인이 완료됐다는 뜻은 아니다. 반대로 브라우저에 로그인돼 있어도 터미널에서 git을 찾지 못할 수 있다.

아래는 서로 다른 상황의 출력 예시다. 실제 버전과 운영체제 메시지는 다를 수 있다.

```
git version 2.x.x
bash: git: command not found
fatal: not a git repository
```

메시지	문제의 위치	먼저 할 일
버전이 없음	실행 파일을 찾음	저장소 폴더 확인
command not found	설치 또는 PATH	설치 확인 후 새 터미널
not a git repository	현재 작업 폴더	저장소 폴더 열기

## PATH는 무엇인가

터미널이 명령 이름으로 프로그램을 찾을 때 참고하는 폴더 목록이다. 설치 직후에 열어둔 터미널은 옛 환경을 가지고 있을 수 있다. 새 터미널에서 다시 확인한다. 문제를 해결하려고 인터넷에서 가져온 긴 설정 명령을 의미를 모른 채 실행하지 않는다.

## global과 local 설정

--global은 이 사용자 계정의 기본 설정이다. 저장소 안의 --local 설정은 그 저장소에 적용한다. 학교 작업과 개인 작업의 작성자 이메일을 다르게 쓰고 싶다면 저장소별 값을 확인한다. `git config --show-origin --get user.email`로 어느 설정 파일에서 값이 왔는지 볼 수 있다.

작성자 이름이 맞아도 Push 인증은 별도다. 설치 문제와 폴더 문제와 계정 문제를 한 번에 바꾸지 말고 한 단계씩 구분한다.

## 실습 3.2 | 저장소 폴더 안에서 명령 실행

1. VS Code에서 Clone한 lesson-lab 폴더를 연다.
2. Terminal의 New Terminal을 연다. 셸을 Git Bash로 선택한다.
3. 아래 명령을 하나씩 실행한다. 결과를 읽은 뒤 다음 줄로 간다.

```
git rev-parse --show-toplevel
git status
git branch --show-current
git remote -v
```

## 출력 해석

출력	뜻	다음 행동
not a git repository	현재 위치가 저장소가 아님	폴더를 다시 연다
working tree clean	미커밋 변경이 없음	작업을 시작해도 됨
modified: README.md	추적 파일이 바뀜	diff로 내용 확인
Untracked files	아직 추적하지 않는 파일	추가할지 판단
origin의 URL	연결된 원격 저장소	자신의 주소인지 확인

실습 명령이 파일을 바꿀 수 있는지 판단한다. status와 diff와 log는 관찰용이다. add와 commit은 로컬 기록을 바꾼다. Push는 원격 상태를 바꾼다. 삭제와 이력 재작성 명령은 대상과 복구 수단을 확인한 뒤 사용한다.

확인 문제 VS Code에 로그인했다면 Git의 모든 Push도 자동으로 되는가?  
도구와 인증 방식에 따라 다르다. 원격 주소가 HTTPS인지 SSH인지 확인하고 그 방식에 맞는 인증을 점검한다. 로그인 화면만 보고 Push 권한이 있다고 단정하지 않는다.

### 3.3 Clone과 기존 폴더의 첫 업로드

관련 근거 [R02] [R05]

#### 이미 저장소가 있다면 Clone부터 한다

같은 프로젝트에서 git init을 여러 번 할 필요는 없다. 원격 저장소가 이미 README를 포함한다면 Clone한 폴더에 파일을 옮기는 방법이 초보자에게 명확하다. 새 로컬 폴더에서 시작할 때만 git init을 사용한다.

```
git clone https://github.com/YOUR-ID/lesson-lab.git
cd lesson-lab
git status
git remote -v
```

상황	추천 출발
GitHub에 이미 파일이 있음	Clone 후 파일 수정
로컬에만 파일이 있음	init 후 빈 원격에 연결
ZIP으로만 받아 둬	새로 Clone한 폴더에 필요한 파일 복사
다른 Git 저장소가 폴더 안에 있음	중첩 저장소인지 먼저 확인

#### 새 로컬 폴더를 올리는 별도 절차

아래는 아직 Git 저장소가 아닌 새 폴더용이다. GitHub에는 README와 라이선스를 만들지 않은 빈 저장소를 준비한다. 기존 lesson-lab 실습 폴더에서 반복 실행하지 않는다.

```
git init -b main
git add README.md
git commit -m 'Add project introduction'
git remote add origin https://github.com/YOUR-ID/NEW-REPO.git
```

```
git push -u origin main
```

빈 원격을 쓰는 이유: 로컬과 원격이 각각 별도 첫 커밋을 만들면 이력이 갈라져 추가 통합이 필요하다. 오류를 피하려고 강제 Push로 덮어쓰지 않는다.

## Clone한 폴더 안에서 무엇을 확인할까

### Clone은 작업 폴더와 Git 정보를 함께 준비한다

파일 목록이 보이는 폴더와 그 아래의 Git 관리 정보가 함께 있어야 이력 명령이 동작한다. Clone 후 다른 폴더를 열어 놓으면 README는 보이는데 Git은 저장소가 아니라고 말하는 상황이 생긴다.

```
git rev-parse --show-toplevel
git remote -v
git branch --show-current
git log -1 --oneline
```

첫 명령의 경로가 지금 편집하는 프로젝트 폴더인지 비교한다. 두 번째 명령에서는 origin 주소의 소유자와 이름을 읽는다. 세 번째는 현재 브랜치다. 마지막은 현재 끝의 커밋이다. 이 네 가지가 다른 화면과 일치해야 같은 프로젝트를 보고 있다고 판단하기 쉽다.

### Clone URL을 복사하는 위치

GitHub의 Code 메뉴에서 HTTPS 또는 SSH 주소를 고른다. 브라우저 주소창에 보이는 파일 상세 주소를 Clone URL처럼 쓰지 않는다. /blob/main/README.md 같은 부분이 붙은 주소는 특정 파일의 보기 주소다.

### 파일을 복사해야 한다면

기존 ZIP 자료를 새 Clone 폴더로 옮길 때 필요한 원본 파일만 고른다. 다른 저장소의 .git 정보를 통째로 덮어쓰지 않는다. 설치 결과물과 개인 설정이 섞이지 않았는지 status와 ignore 규칙을 확인한다.

확인 결과	해석
origin이 예상 주소	원격 연결 출발점이 맞음
main 대신 다른 이름	실제 기본 브랜치에 맞춰 진행
커밋이 전혀 없음	빈 저장소인지 확인

같은 프로젝트를 두 번 Clone할 수는 있다. 다만 각 폴더의 미커밋 변경은 자동 공유되지 않는다. 지금 수정하는 사본의 위치를 계속 확인한다.

### 실습 3.3 | 다른 위치에서 복제해 보기

1. 원래 Clone 폴더의 바깥으로 이동한다.
2. git clone 뒤에 lesson-lab-check라는 새 대상 폴더 이름을 지정한다.
3. 새 폴더에서 README와 커밋 로그를 확인한다.
4. 아직 Push하지 않은 변경이 없다는 사실도 확인한다.

```
git clone https://github.com/YOUR-ID/lesson-lab.git lesson-lab-check
cd lesson-lab-check
git log -5 --oneline
git status
```

#### origin already exists를 만났다면

origin이라는 별칭이 이미 있다는 뜻이다. 먼저 git remote -v로 주소를 확인한다. 주소가 맞으면 다시 추가하지 않는다. 주소가 틀렸을 때만 아래 명령으로 바꾼다.

```
git remote set-url origin https://github.com/YOUR-ID/lesson-lab.git
git remote -v
```

#### 성공 확인은 두 군데에서 한다

로컬 log에 커밋이 있고 웹의 같은 브랜치에도 같은 변경이 보여야 한다. 원격 URL이 자신의 저장소인지 확인하지 않으면 엉뚱한 곳을 보고 오류를 찾게 된다.

확인 문제 private 저장소 Clone에 실패하면 그 저장소가 삭제된 것일까?  
반드시 그렇지 않다. 접근 권한이 없을 때도 찾을 수 없다는 메시지가 나올 수 있다. 주소와 인증 계정과 조직 승인 여부를 함께 확인한다.

## 3.4 Desktop 화면에서 Clone부터 Push까지

관련 근거 [R08] [R76]

#### 새 폴더를 받을 위치를 정확히 정한다

아래는 GitHub 공식 문서의 Clone 대화상자다. macOS 예시이며 Windows에서도 저장소 출처와 로컬 경로를 선택하는 목적은 같다. Local Path에 보이는 경로를 그대로 입력하지 말고 자신의 작업 폴더를 선택한다.

위치	할 일
상단 GitHub.com	로그인 계정의 저장소에서 선택
상단 URL	알고 있는 저장소 주소로 지정
저장소 목록	대상이 맞는지 확인
Local Path	복제 폴더가 생길 위치 확인
Clone	로컬 복제 시작

Clone 후 탐색기에서 실제 폴더를 열어 본다. 원래 ZIP 폴더를 계속 수정하면 Desktop의 Changes가 비어 있을 수 있다.

Desktop에서 버튼을 누르기 전후의 상태 읽기

화면 세 곳을 한 작업으로 연결한다

편집기는 디스크의 파일을 고친다. Desktop은 선택한 저장소의 변경과 이력을 보여 준다. 웹은 원격에 도착한 커밋을 보여 준다. 편집기에 보이는 파일이 어느 폴더인지부터 확인하면 서로 다른 사본을 보고 생기는 혼란을 줄일 수 있다.

행동 직후	기대하는 관찰
파일 저장	Changes에 해당 변경 표시
커밋	History에 새 기록 표시
커밋 후 추가 편집	새 편집만 Changes에 표시
Push	같은 브랜치의 웹에 커밋 표시
Fetch	원격 상태 확인 후 필요한 다음 동작 표시

## Changes가 비어 있을 때

이미 커밋한 상태인지 먼저 본다. 편집기에서 저장했는지와 Desktop의 Current repository가 맞는지 확인한다. 무시되는 파일인지와 실제 내용 차이가 있는지도 확인한다. 빈 화면만으로 작업을 잃었다고 판단하지 않는다.

## Push 버튼을 눌렀는데 웹에서 안 보일 때

웹의 브랜치를 확인한다. main만 열어 두고 작업 브랜치에 Push했을 수 있다. Desktop의 History에서 해당 커밋을 찾고 웹에서 같은 커밋 설명과 ID를 비교한다. 인증 오류가 있었다면 오류를 해결한 뒤 성공 여부를 다시 확인한다.

## 충돌 중에는 동작을 하나씩 한다

현재 브랜치와 가져오는 브랜치를 적는다. 충돌 파일을 고친 뒤 최종 diff를 확인한다. 표식을 지웠다고 내용이 올바르게 통합된 것은 아니다. 기능을 실행해 확인하고 Desktop에서 진행 중인 통합 작업을 마친다.

파일 선택 해제와 Discard는 의미가 다르다. 내용을 버리는 메뉴를 사용하기 전에 필요한 편집을 별도 사본으로 남길지 판단한다.

## 실습 3.4 | 세 화면을 오가며 확인

1. Current repository에서 lesson-lab을 선택한다.
2. Current branch에서 새 작업 브랜치를 만든다.
3. Open in editor 또는 폴더 열기로 README를 수정한다.
4. Changes에서 바뀐 줄과 선택 파일을 읽는다.
5. Summary를 적고 로컬 커밋을 만든다.
6. Publish branch 또는 Push origin을 누른다.
7. View on GitHub로 같은 브랜치의 커밋을 확인한다.

## Desktop과 명령어의 대응

화면의 행동	대응하는 Git 개념
Changes의 체크	다음 커밋에 넣을 변경 선택
Commit 버튼	선택한 변경의 로컬 기록
Fetch origin	원격의 현재 상태 수집
Pull origin	가져온 상태를 현재 작업에 통합
Push origin	로컬 커밋을 원격에 전달
History	커밋 기록 탐색

### 충돌 안내가 나타나면

충돌 파일을 편집기로 열어 양쪽 의도를 읽는다. 파일의 표식을 정리하고 저장한 뒤 Desktop으로 돌아와 해결 상태와 최종 커밋을 확인한다. 어떤 브랜치의 변경을 받는 중인지 모른 채 한쪽만 선택하지 않는다. 취소 가능 여부와 메뉴는 진행 작업의 종류에 맞게 확인한다.

확인 문제 Changes의 모든 체크를 해제하면 파일 내용도 원상복구될까?  
커밋에 넣을 선택이 달라지는 것이며 편집 내용을 버리는 것과는 다르다. 실제 파일 폐기 동작을 별도로 구분한다.

## 4장 인증과 변경 기록과 원격 동기화

접속 권한을 확인하고 다음 커밋에 넣을 변경을 선택한다. VS Code의 화면과 Git 명령을 비교한 뒤 Fetch와 Pull과 Push를 구분한다.

### add는 파일 이름을 예약하는 명령이 아니다

작업 트리는 편집 중인 파일 상태다. Index는 다음 커밋에 사용할 경로와 내용의 상태를 표현한다. git add는 실행 시점의 파일 내용을 Index에 반영한다. 그 뒤 파일을 다시 편집해도 Index가 자동으로 최신 상태가 되지는 않는다. 같은 파일이 스테이징된 변경과 스테이징되지 않은 변경에 동시에 나타날 수 있는 이유다.

lesson-lab의 README에 사용 대상 안내를 추가한 뒤 add를 실행하고 다시 실행 명령을 수정해 보자. git diff는 두 번째 수정만 보여 주고 git diff --staged는 첫 번째 수정을 보여 준다. 이 관찰을 통해 두 명령이 서로 다른 상태를 비교한다는 사실을 확인할 수 있다. 커밋 직전에는 다음 커밋에 실제로 들어갈 차이를 확인해야 한다.

스테이징은 커밋의 논리적 경계를 만드는 장치다. 한 파일에서 버그 수정과 표현 개선을 동시에 했더라도 git add -p로 변경 덩어리를 선택할 수 있다. 다만 선택한 조합이 독립적으로 실행 가능한지는 사람이 판단해야 한다. 기계적으로 작은 커밋을 많이 만드는 것과 설명 가능한 변경 단위를 만드는 것은 다르다.

### 커밋은 스냅샷과 부모 관계를 기록한다

Git의 blob은 파일 내용을 표현하고 tree는 이름과 모드 및 하위 객체의 관계를 표현한다. commit은 최상위 tree와 부모 커밋 및 작성 정보와 메시지를 연결한다. 파일 이름은 blob 자체에 들어 있지 않으므로 같은 내용이 여러 경로에서 재사용될 수 있다. 실제 저장소는 압축과 packfile 등을 사용하므로 논리적 객체 구조와 물리적 저장 방식을 구분한다. [T01]

객체 ID는 객체의 내용에 의존한다. 커밋 메시지나 부모가 바뀌면 같은 파일 스냅샷이라도 다른 커밋 ID가 생길 수 있다. rebase가 이력을 재작성한다고 말하는 이유가 여기에 있다. 해시가 존재한다는 사실은 내용 식별과 무결성 확인에 도움을 주지만 작성자의 정당성이나 코드의 안전성을 자동으로 증명하지는 않는다.

## origin/main은 서버를 실시간으로 바라보는 창이 아니다

origin은 원격 저장소 주소에 붙인 로컬 별칭이다. origin/main은 마지막으로 가져온 원격 main의 위치를 기록하는 원격 추적 참조다. 다른 사람이 서버의 main을 바꾸어도 내 origin/main은 fetch하기 전까지 오래된 상태일 수 있다. 따라서 로컬 화면의 ahead와 behind 정보도 관찰한 원격 상태를 기준으로 해석한다.

fetch는 객체와 참조 정보를 가져오는 과정이며 현재 작업 브랜치에 통합하는 과정과 구분된다. pull은 가져오기와 통합을 함께 수행하며 통합 방식은 옵션과 설정에 따라 달라진다. 먼저 fetch한 뒤 차이를 읽으면 어떤 변경을 받아들일지 명시적으로 판단할 수 있다. [T02]

## 충돌 상태의 Index에는 여러 후보가 공존한다

일반적인 Index 항목은 다음 스냅샷에 들어갈 파일 상태를 표현한다. 병합 충돌 중에는 한 경로에 최대 세 단계의 항목이 존재할 수 있다. stage 1은 공통 기반이며 stage 2와 stage 3은 병합의 두 쪽이다. git ls-files --unmerged는 이 상태를 관찰하는 명령이다. 작업 트리에 보이는 충돌 표식과 Index에 보관된 후보는 서로 다른 표현이다. [T18]

충돌을 해결한다는 것은 표식 문자를 지우는 일에서 끝나지 않는다. 두 후보의 의도와 기반 상태를 읽고 최종 내용을 결정해야 한다. 최종 파일을 git add로 기록하면 다음 커밋의 후보를 확정하는 단계로 넘어간다. 아무 내용이나 저장해 표식만 없애도 Git의 텍스트 충돌은 사라질 수 있으므로 의미의 검증은 별도로 필요하다.

rebase 중에는 어떤 쪽을 ours와 theirs라고 부르는지 직관과 어긋날 수 있다. 현재의 기반 위에 커밋을 다시 적용하는 문맥이기 때문이다. 사람 이름이나 브랜치 이름을 추정해 한쪽 전체를 선택하기보다 각 단계의 실제 내용과 적용 중인 커밋을 확인한다. 충돌 도구의 버튼 이름만으로 판단하지 않는 습관이 중요하다.

## 참조를 갱신할 때도 읽은 상태가 바뀌었을 수 있다

자동화가 브랜치의 현재 값을 읽은 뒤 새 값으로 바꾸기까지 다른 작업이 같은 브랜치를 갱신할 수 있다. 이때 이전 관찰을 전제로 무조건 쓰면 타인의 변경을 덮어쓸 수 있다. git update-ref는 이전 객체 식별자를 함께 주어 현재 값이 예상과 같을 때만 갱신하는 방식을 지원한다. 이는 비교 후 갱신이라는 일반적인 동시성 제어의 한 사례다. [T20]

개념적으로 현재 값이 A일 때만 B로 바꾼다는 조건이다. 조건이 실패하면 다시 읽고 충돌을 해결해야 한다. 실패했다는 이유로 조건을 제거하면 보호 장치의 의미가 없어진다. 원격 Push의 lease 조건도 이런 사고방식으로 이해할 수 있다. 자동화의 편의를 위해 예상값 검사를 지우지 않는다.

이 원리를 이해했다고 모든 사용자가 저수준 명령으로 브랜치를 직접 조작할 필요는 없다. 일반적인 작업은 switch와 commit 및 merge로 수행한다. 내부 갱신의 계약을 알면 도구가 경쟁 상태를 거부했을 때 강제로 우회하기 전에 무엇을 다시 확인해야 하는지 알 수 있다.

관련 근거 [T11]

## 상태 변화 / add 이후 다시 편집하면 무엇이 다른가

작업 트리

v1

Index

v1

HEAD

v1

원격 main

v1

스테이징은 파일 이름이 아닌 실행 시점의 내용을 선택한다. 마지막 화면에서 작업 트리는 v3이고 공유된 커밋은 v2다. 이 차이가 git diff에 남는다.

작업 트리

v3

Index

v2

HEAD

v2

원격 main

v2

스테이징은 파일 이름이 아닌 실행 시점의 내용을 선택한다. 마지막 화면에서 작업 트리는 v3이고 공유된 커밋은 v2다. 이 차이가 git diff에 남는다.

스테이징은 파일 이름이 아닌 실행 시점의 내용을 선택한다. 마지막 화면에서 작업 트리는 v3이고 공유된 커밋은 v2다. 이 차이가 git diff에 남는다.

1. 처음에는 네 위치가 같은 내용을 가리킨다
2. 편집과 저장은 작업 트리를 바꾼다
3. git add는 현재 내용을 Index에 반영한다
4. 다시 편집해도 Index는 자동 갱신되지 않는다

- 5. git commit은 Index의 v2를 기록한다
- 6. git push 뒤에도 미커밋 v3는 로컬에 남는다

### 4.1 HTTPS와 SSH 인증 이해

관련 근거 [R03] [R04] [R09]

#### 인증 방식은 하나를 골라 제대로 익힌다

HTTPS는 브라우저 인증과 자격 증명 관리자를 함께 쓰기 쉽다. SSH는 공개 키를 GitHub에 등록하고 개인 키로 인증한다. GitHub 계정 비밀번호를 Git HTTPS 인증용 비밀번호처럼 입력하는 방식은 사용하지 않는다.

초보자는 GitHub Desktop의 브라우저 로그인이나 gh auth login의 안내를 따라 시작할 수 있다. 토큰을 직접 쓰는 상황이라면 필요한 저장소와 최소 권한과 만료일을 정한다. Fine-grained PAT의 지원 범위와 조직 승인 정책도 확인한다.

항목	공개해도 되는가	관리
SSH 공개 키인 .pub	GitHub에 등록 가능	다른 곳 재사용 여부 판단
SSH 개인 키	공개 금지	암호와 안전한 보관
PAT	공개 금지	최소 권한과 만료와 회수
원격 URL	보통 공개 가능	토큰을 URL에 넣지 않기

```
gh auth login
gh auth status
gh auth setup-git
```

#### SSH를 선택한 경우

기존 키가 있는지 먼저 확인한다. 새로 만들 때 기존 파일을 덮어쓰지 않는다. ssh-keygen -t ed25519 -C 'YOUR-EMAIL'로 생성할 수 있다. 운영체제별 ssh-agent 설정은 공식 안내를 따른다. .pub 내용만 계정의 SSH and GPG keys에 등록한다.

첫 SSH 연결에서 서버 지문을 묻는다면 공식 GitHub의 SSH key fingerprints와 대조한다. 알 수 없는 지문에 습관적으로 yes를 입력하지 않는다.

## 인증 실패를 네 층으로 나누어 진단

### 주소와 신원과 권한과 정책을 순서대로 본다

Push 오류에 인증이라는 말이 보여도 원인은 하나가 아니다. 잘못된 URL을 보고 있거나 다른 계정으로 인증했거나 쓰기 역할이 없거나 조직의 조건을 충족하지 못했을 수 있다.

층	확인할 근거	다음 판단
주소	git remote -v	대상 저장소가 맞는가
신원	gh auth status 또는 SSH 확인	어느 계정인가
권한	초대와 저장소 역할	쓰기 가능한가
정책	브랜치 규칙과 조직 조건	허용한 방식인가

### HTTPS와 SSH를 섞어 고치지 않는다

https://로 시작하는 원격은 HTTPS 방식이다. git@github.com:으로 시작하는 원격은 SSH 방식이다. SSH 키를 잘 등록했어도 HTTPS 원격의 자격 증명 오류가 저절로 해결되지는 않는다. 원격 주소를 바꾸는 선택도 가능하지만 어떤 인증을 사용할지 결정한 뒤 한다.

### PAT가 권한을 늘려 주는 것은 아니다

토큰은 소유자가 원래 가진 접근 범위 안에서 부여한 권한만 사용할 수 있다. 저장소에 쓰기 권한이 없는 사람에게 넓은 토큰을 만들게 해도 관리자 역할이 생기지 않는다. Fine-grained PAT는 소유 주체와 저장소 선택과 세부 권한과 만료와 승인 상태를 함께 확인한다.

### 복사할 정보와 숨길 정보

오류 문장과 원격 저장소 이름과 인증 방식은 문제 해결에 도움이 된다. 실제 토큰과 개인 키와 복구 코드는 공유하지 않는다. 터미널의 전체 자격 증명 파일을 출력해서 보내는 방식도 피한다.

성공 확인: 인증 테스트만 보지 않는다. 실제 대상에서 읽기와 필요한 쓰기 작업이 허용되는지 각각 확인한다.

## 실습 4.1 | 연결과 권한을 따로 진단

1. git remote -v로 HTTPS인지 SSH인지 읽는다.
2. HTTPS라면 gh auth login으로 본인 계정에 인증한다.
3. SSH라면 키 등록 후 ssh -T git@github.com으로 확인한다.
4. git fetch origin을 실행해 저장소 읽기가 되는지 본다.
5. 자신의 작업 브랜치에서 Push해 쓰기 권한도 확인한다.

```
git remote -v
git fetch origin
git branch --show-current
```

SSH 테스트에서 GitHub가 인증 성공을 알리면서 셸 접근을 제공하지 않는다고 설명하는 것은 정상이다. 명령 종료 코드만으로 실패라고 판단하지 말고 메시지를 읽는다.

### 토큰을 꼭 만들어야 할 때

Settings의 Developer settings에서 토큰 종류를 선택한다. Resource owner와 접근 저장소와 작업에 필요한 권한을 정한다. 토큰 값은 생성 직후에만 볼 수 있는 경우가 있다. 코드와 문서와 메신저에 붙이지 않고 자격 증명 도구나 비밀 저장소에 보관한다.

확인 문제 인증은 성공했는데 다른 사람 저장소에 Push가 안 되는 이유는?  
신원 확인과 쓰기 권한이 다르기 때문이다. 협업자로 초대받았는지와 조직 정책을 확인한다.  
권한이 없으면 Fork 후 PR을 제안할 수 있다.

## 4.2 add와 commit과 diff를 정확히 사용

관련 근거 [R10] [R65] [R70] [R72]

### 커밋할 내용을 미리 고른다

Git의 작업 풀더는 편집 중인 파일이다. 스테이징 영역인 Index는 다음 커밋에 넣을 상태다. 커밋은 그 선택을 이력으로 남긴다. 같은 파일이라도 add 이후에 더 고치면 스테이징된 내용과 현재 내용이 다를 수 있다.

```
git status
git diff
git add README.md
git diff --staged
git commit -m 'Clarify lesson setup'
```

명령	무엇을 비교하거나 바꾸는가
git diff	작업 폴더와 Index의 차이
git diff --staged	Index와 현재 커밋의 차이
git add 파일	현재 파일 상태를 Index에 반영
git add -p	변경 덩어리를 골라서 추가
git status	추적 상태와 스테이징 상태 요약

좋은 습관: git add . 전에 status를 본다. 한 번의 커밋에는 하나의 설명 가능한 목적을 담는다. 잠깐 만든 자료와 비밀 설정 파일이 섞이지 않게 한다.

## status의 두 칸과 add 이후의 변경 읽기

### 짧은 상태 표시는 Index와 작업 폴더를 나눠 보여 준다

git status --short의 일반적인 추적 파일 표시는 두 칸을 사용한다. 첫 칸은 Index 쪽 변화이며 둘째 칸은 작업 폴더 쪽 변화다. 아래 각 줄은 같은 파일을 서로 다른 시점에서 봤을 때의 예시다. 한 번의 출력이라고 해석하지 않는다.

```
M README.md
M README.md
MM README.md
```

예시 줄	상태
첫째 줄	편집했지만 아직 Stage하지 않음
둘째 줄	편집 상태를 Stage함
셋째 줄	Stage한 뒤 같은 파일을 다시 편집함

### 세 번째 상태에서 commit하면

Index에 있던 버전이 커밋된다. 그 뒤에 더 편집한 내용은 작업 폴더에 남을 수 있다. 그래서 커밋 직후에도 modified 표시가 보일 수 있다. 커밋이 실패했다고 단정하지 말고 git show HEAD와 두 diff를 나누어 읽는다.

```
git diff --staged
git diff
git show --stat HEAD
```

## 일부 변경만 고르는 이유

한 커밋에 기능 추가와 관련 없는 안내 수정과 자동 포맷이 섞이면 원인을 찾고 되돌리기 어렵다. 단순히 커밋을 작게 만들려는 목표보다 하나의 의도를 설명할 수 있는 단위를 고른다. 다만 기능이 동작하지 않는 중간 상태를 무리하게 만들어서는 안 된다.

add는 파일 이름을 등록하는 동작만이 아니다. 그 순간 선택한 파일 내용을 다음 커밋의 상태에 반영한다.

## 실습 4.2 | 같은 파일에서 두 상태 관찰

1. README에 시작 안내 한 줄을 넣고 저장한다.
2. git add README.md로 스테이징한다.
3. 같은 파일에 두 번째 줄을 더 적고 저장한다.
4. git diff와 git diff --staged를 각각 읽는다.
5. 두 줄을 함께 기록하려면 다시 add한 뒤 커밋한다.

```
git diff
git diff --staged
git add README.md
git diff --staged
git commit -m 'Add two setup notes'
git show --stat HEAD
```

## 커밋 메시지를 쓰는 기준

무엇을 바꿨는지 제목에 적는다. 왜 필요한지는 본문이나 PR에 적는다. Update와 수정처럼 넓은 말보다 실행 안내의 누락 조건 보완처럼 검토할 대상을 좁힌다. 한국어 메시지도 가능하다. 팀의 검색 방식에 맞춘다.

## 한 파일의 일부만 커밋

git add -p는 변경 덩어리별로 선택하게 한다. 서로 붙은 줄은 분리하기 어려울 수 있다. 이럴 때 억지로 쪼개기보다 작업을 다시 정리한다. 커밋 직전 diff --staged를 읽는 습관이 가장 중요하다.

확인 문제 add한 파일을 커밋에서 빼고 싶으면 파일 자체를 삭제해야 할까?  
아니다. git restore --staged 파일로 스테이징을 취소한다. 작업 폴더의 편집 내용은 유지된다. 6.1절에서 복구 명령의 차이를 다시 다룬다.

### 4.3 VS Code Source Control을 읽는 법

관련 근거 [R77] [R14]

#### 파일 편집과 Git 선택을 한 창에서 확인한다

VS Code에서 Git 저장소 폴더를 연다. Source Control 보기는 추적한 파일의 변경과 스테이징 상태를 보여 준다. 확장 기능과 사용자 설정에 따라 단축키와 화면 배치가 다를 수 있다. 왼쪽 아이콘의 모양을 외우기보다 보기 이름으로 찾는다.

화면의 구역	Git의 의미	확인 방법
Changes	아직 스테이징하지 않은 변경	파일을 눌러 diff 읽기
Staged Changes	다음 커밋의 선택	staged diff 확인
Message 입력	커밋 설명	목적 한 가지로 작성
상태 표시줄의 브랜치	현재 작업 기준	전환 전 미커밋 상태 확인
Sync Changes	원격과 동기화하는 작업	포함 동작을 먼저 확인

#### 같은 파일이 두 구역에 나타나는 이유

파일을 Stage한 뒤 다시 고치면 한 파일에 스테이징된 부분과 아직 선택하지 않은 부분이 함께 생긴다. 오류가 아니다. 파일 이름만 보지 말고 각 구역의 diff를 읽는다.

#### 덧붙임과 삭제의 색만으로 판단하지 않는다

녹색과 빨간색은 차이를 읽기 위한 표시다. 파일 이동과 줄바꿈 변경과 자동 포맷도 큰 diff를 만들 수 있다. 기능을 바꾼 줄과 표현만 바뀐 줄을 나누어 검토한다.

Sync는 단순 저장 버튼이 아니다. 원격 통합이나 Push가 포함될 수 있다. 처음에는 Fetch와 Pull과 Push를 구분해서 수행하면 상태를 이해하기 쉽다.

## 부분 스테이징을 작은 검토 단위로 활용

### 파일 하나에도 목적이 두 개일 수 있다

README의 설치 안내와 기여 규칙을 함께 고쳤다고 하자. 목적이 다르면 설치 안내만 먼저 Stage할 수 있다. 파일 전체를 한꺼번에 넣어야 한다는 규칙은 없다. 단 서로 의존하는 변경을 나누어 중간 커밋이 깨지지 않도록 확인한다.

읽는 대상	비교 기준	확인할 내용
Changes의 diff	Index와 작업 파일	아직 선택하지 않은 편집
Staged Changes의 diff	HEAD와 Index	이번 커밋에 들어갈 상태
커밋 후 History	기록된 커밋	의도한 내용이 기록됐는가

### 같은 파일이 양쪽에 있을 때

먼저 선택한 변경이 Index에 있다. 그 뒤 새로 고친 내용은 작업 파일에 있다. 두 diff를 각각 읽으면 어느 부분이 이번 커밋에 들어가는지 알 수 있다. 다시 파일 전체를 Stage하면 새 편집도 선택될 수 있으므로 목적에 맞게 결정한다.

### 화면과 터미널을 서로 검증

Source Control에서 Stage한 뒤 `git diff --staged`를 읽는다. 선택하지 않은 나머지는 `git diff`로 확인한다. VS Code의 체크나 색을 외우기보다 두 비교 기준이 무엇인지 이해하면 다른 편집기로 옮겨도 같은 판단을 할 수 있다.

### 부분 선택이 어려운 변경

서로 붙은 줄이나 대규모 자동 포맷은 목적별 선택이 어려울 수 있다. 우선 포맷만 별도 작업으로 분리하거나 편집을 작은 단위로 되돌려 진행한다. 이미 중요한 내용이 있다면 무리한 Discard보다 사본을 만들어 비교하는 편이 낫다.

커밋 전에 저장되지 않은 탭도 확인한다. 편집기에 보이는 내용과 Git이 읽는 디스크 파일이 같다는 전제로만 판단하지 않는다.

## 실습 4.3 | 일부 줄만 Stage하기

1. README의 서로 떨어진 두 위치를 바꾸고 저장한다.
2. Source Control에서 diff를 연다.
3. 한 변경만 선택해 Stage Selected Ranges 등의 메뉴를 확인한다.

4. Staged Changes와 Changes를 각각 읽는다.
5. 터미널 명령으로 같은 선택을 확인한다.
6. 선택한 변경만 커밋하고 나머지 변경이 남는지 본다.

```
git status
git diff --staged
git diff
```

### 동작이 화면과 다를 때

Git 설치 여부와 저장소 선택을 확인한다. 여러 폴더를 연 작업 공간에서는 다른 저장소의 Source Control을 보고 있을 수 있다. 저장하지 않은 편집 내용과 디스크 파일의 차이도 확인한다. 체크를 취소하는 동작과 Discard Changes처럼 파일 내용을 버리는 동작을 구분한다.

확인 문제 스테이징하지 않은 새 편집까지 이번 커밋에 꼭 들어가야 할까?  
 다음 커밋에는 Index에 선택한 상태가 들어간다. 스테이징하지 않은 편집을 함께 기록하려면 그 변경도 추가로 선택해야 한다.

## 4.4 Fetch와 Pull과 Push와 추적 브랜치

관련 근거 [\[R10\]](#) [\[R11\]](#) [\[R68\]](#)

### 원격의 최신 상태를 관찰한 뒤 통합한다

origin/main은 GitHub의 main 자체가 아니라 마지막으로 가져온 원격 상태를 로컬에 기록한 참조다. fetch는 원격 정보를 갱신한다. 내 작업 브랜치를 자동으로 병합하지는 않는다. pull은 보통 fetch와 통합을 함께 수행하며 통합 방식은 설정이나 옵션에 따라 달라진다.

```
git fetch origin
git log --oneline --left-right HEAD...origin/main
git status -sb
```

명령	변화
git fetch origin	원격 추적 참조 갱신
git pull --ff-only	갈라진 이력은 거절하고 앞으로만 이동
git push	설정된 목적지에 로컬 커밋 전달

명령	변화
<code>git push -u origin 브랜치</code>	첫 Push와 upstream 설정
<code>git branch -vv</code>	브랜치의 추적 관계 확인

## non-fast-forward는 덮어쓰기 요청이 아니다

원격에 내가 받지 않은 커밋이 있으면 Push가 거절될 수 있다. 먼저 fetch로 원격의 새 변경을 확인한다. 변경을 병합하거나 합의된 rebase를 수행한 뒤 다시 Push한다. 처음부터 --force를 붙이지 않는다.

`pull --ff-only`가 실패하면 로컬과 원격이 갈라졌을 수 있다. 오류 문장과 log를 읽고 5.1절과 5.2절의 통합 절차를 따른다.

## ahead와 behind를 보고 바로 판단하지 않기

### 비교 대상은 마지막으로 가져온 원격 정보다

`git status -sb`와 `git branch -vv`에 ahead나 behind가 나타날 수 있다. 이 비교는 로컬에서 알고 있는 추적 대상에 대한 것이다. 누군가 지금 막 원격에 Push했다면 아직 fetch하지 않은 내 컴퓨터는 그 커밋을 모를 수 있다.

표시의 뜻	가능한 다음 작업
로컬만 앞섬	커밋 검토 후 Push
가져온 원격만 앞섬	미커밋 상태 확인 후 통합
양쪽에만 새 커밋 존재	Merge 또는 합의된 Rebase 검토
추적 관계 없음	첫 Push와 upstream 설정 확인

## --ff-only를 입문 경로에서 쓰는 이유

단순히 브랜치 끝을 앞으로 옮겨 통합할 수 있는지 확인하게 한다. 양쪽 이력이 갈라져 있으면 자동으로 새 병합 커밋을 만들지 않고 멈추므로 상황을 읽을 기회가 생긴다. 실패는 파일을 버리라는 요청이 아니다.

## 미커밋 변경이 있을 때

Pull이 기존 편집과 부딪히면 거절되거나 추가 판단이 필요할 수 있다. 필요한 변경은 커밋하거나 적절하게 잠시 보관한다. 모든 미커밋 변경이 있다고 언제나 Pull이 불가능한 것은 아니지만 초보 실습에서는 작업을 정리한 뒤 통합하면 결과를 예측하기 쉽다.

## 작업 종료 전에 남길 기록

어느 브랜치에 커밋했는지와 원격으로 보냈는지를 확인한다. 다음 PC에서 Clone했는데 자료가 없으면 원래 PC의 로컬에만 남은 변경인지부터 찾는다. 클라우드 폴더 동기화와 Git 원격 전달은 다른 체계다.

Push 거절 직후에는 원격을 덮어쓰지 않는다. Fetch 후 양쪽의 새 커밋을 읽는 순서로 돌아간다.

## 실습 4.4 | 웹의 변경을 로컬에 반영

1. 로컬의 main으로 전환한다. 미커밋 변경이 있으면 먼저 정리한다.
2. 웹의 main에서 README에 원격 연습 문장을 추가한다.
3. 로컬에서 fetch한 뒤 로그를 본다.
4. 파일 내용이 아직 안 바뀌었음을 확인한다.
5. pull --ff-only 후 파일과 로그를 다시 확인한다.

```
git switch main
git fetch origin
git log -5 --oneline origin/main
git pull --ff-only
git status -sb
```

## 새 작업 브랜치 보내기

```
git switch -c feature/fetch-note
편집기에서 README를 수정하고 저장
git add README.md
git commit -m 'Document fetch behavior'
git push -u origin feature/fetch-note
git branch -vv
```

확인 문제 origin/main이 최신이라면 GitHub도 지금 그 상태라고 확신할 수 있을까? 마지막 fetch 시점의 상태만 안다. 다른 사람이 그 뒤 Push했을 수 있다. 원격 추적 참조는 실시간 연결 화면이 아니다.

## 5장 브랜치 통합과 충돌 해결

갈라진 작업을 통합하고 충돌한 요구를 함께 반영한다. Stash와 Rebase와 Cherry-pick의 목적을 구분하며 부모 관계로 이력을 읽는다.

### 브랜치는 가벼운 이름이며 이력은 방향 있는 그래프다

커밋은 부모를 가리킨다. 자식 커밋에서 부모를 따라가면 과거 이력을 읽을 수 있다. 이 관계는 순환하지 않는 방향 그래프를 이룬다. 브랜치는 그 그래프의 한 커밋을 가리키는 이동 가능한 참조다. 브랜치 생성은 폴더 전체를 복사하는 작업이 아니며 브랜치를 지웠다고 즉시 모든 관련 객체가 사라지는 것도 아니다.

한 브랜치의 끝이 다른 브랜치 끝의 조상이라면 fast-forward가 가능하다. 기준 참조를 앞으로 이동시키면 되기 때문이다. 양쪽에서 독립적인 커밋이 생겼다면 서로 다른 계보를 통합해야 한다. 일반적인 merge commit은 두 부모를 기록해 두 작업 흐름이 만났음을 보존한다.

### 충돌은 텍스트 통합의 실패이며 의미 통합의 전부가 아니다

세 방향 병합은 공통 조상과 양쪽 변경을 비교한다. 같은 줄 근처를 다르게 바꾸면 도구가 자동 선택하지 못해 충돌을 표시할 수 있다. 하지만 충돌 표시가 없다고 기능이 올바르게 결합된다는 뜻은 아니다. 한 브랜치가 함수 이름을 바꾸고 다른 브랜치가 다른 파일에서 이전 이름을 호출하면 텍스트 병합은 성공해도 실행은 실패할 수 있다.

lesson-lab에서 한 사람은 자료 필드명을 subject에서 category로 바꾸고 다른 사람은 subject를 사용하는 필터를 추가했다고 하자. 파일이 다르다면 Git이 병합 충돌을 표시하지 않을 수 있다. 병합 후 테스트가 필요한 이유는 Git이 파일 차이를 통합하는 도구이지 모든 업무 의미를 이해하는 검증기가 아니기 때문이다.

충돌을 해결할 때는 어느 쪽을 남길지보다 최종 프로그램이 어떤 규칙을 만족해야 하는지 먼저 정한다. 충돌 표식을 지우는 것은 문법적 정리일 뿐이다. 관련 함수와 테스트 및 문서를 함께 읽고 결과를 실행한다. 병합 중단 경로를 알고 시작하면 잘못된 상태에서 억지로 마무리할 필요가 줄어든다.

## rebase는 과거 객체를 움직이는 대신 새 계보를 만든다

rebase는 선택한 커밋의 변경을 새로운 기준 위에 다시 적용해 새로운 커밋을 만든다. 부모 관계가 바뀌므로 ID도 달라진다. 이미 공유한 커밋을 재작성하면 다른 사람의 참조와 어긋날 수 있다. 따라서 개인 작업 이력을 정리하는 경우와 팀이 이미 사용하는 이력을 바꾸는 경우를 구분한다.

merge와 rebase 중 하나가 언제나 더 전문적인 방식은 아니다. 병합 관계를 보존하는 가치와 선형 이력의 읽기 쉬움 및 팀의 복구 절차를 비교해야 한다. Squash도 여러 변경을 하나의 새 커밋으로 묶는 정책이며 원래 커밋의 세부 경계가 기준 브랜치에서 달라진다. 선택한 정책이 리뷰와 원인 조사에 어떤 영향을 주는지 함께 판단한다.

## 텍스트 충돌이 없다는 사실은 통합 성공의 일부다

두 수정이 다른 줄을 바꾸면 Git은 자동으로 합칠 수 있다. 그러나 한쪽이 함수의 단위를 초에서 밀리초로 바꾸고 다른 쪽이 그 값을 표시하는 계산을 추가했다면 문법적으로 합쳐져도 의미가 틀릴 수 있다. 텍스트의 겹침과 프로그램의 동작 및 업무 규칙의 충돌은 서로 다른 층이다.

충돌의 층	관찰되는 현상	필요한 확인
텍스트	같은 부분을 자동 통합하지 못함	기반과 두 후보를 비교한 최종 편집
실행	타입 검사나 테스트 또는 실행이 실패함	연결된 호출과 입력 계약 확인
의미	실행은 되지만 업무 규칙이 달라짐	수용 기준과 실제 사용 시나리오 대조

자료 탐색기에서 한 PR은 삭제한 자료의 즐겨찾기를 유지하고 다른 PR은 삭제 즉시 즐겨찾기도 정리한다고 가정하자. 둘 다 독립적으로는 합리적인 정책일 수 있다. 통합 전에 삭제의 의미를 합의하지 않으면 테스트를 더 추가해도 서로 다른 기대를 검증하게 된다. 기술적 병합에 앞서 업무 규칙의 일치를 확인해야 한다.

## 병합 결과를 재현 가능한 입력으로 검증한다

병합 전 각 브랜치에서 통과했던 입력을 통합 결과에서도 실행한다. 여기에 두 변경이 동시에 작용하는 입력을 추가한다. 각각의 기능을 따로 시험하는 것만으로는 상호작용을 확인할 수 없다. 검색과 즐겨찾기를 동시에 바꾼 경우에는 즐겨찾기 필터를 켜 상태에서 검색어를 변경하는 경로가 중요하다.

리뷰 기록에는 충돌 파일의 목록보다 해결한 규칙을 적는 편이 유용하다. 한쪽을 선택했다는 말보다 저장 형식은 새 버전을 따르고 이전 자료는 읽을 때 변환하도록 했다고 설명한다. 이후 오류가 발견되면 어떤 판단을 되돌리거나 수정해야 하는지 찾을 수 있다.

장기 브랜치는 한 번에 통합해야 하는 가정의 수를 늘릴 수 있다. 변경을 작게 나누고 자주 기준 브랜치와 관계를 확인하면 충돌의 원인을 좁히기 쉽다. 다만 통합 빈도 자체를 목표로 삼아 미완성 동작을 무조건 공개해서는 안 된다. 기능 노출과 코드 통합을 분리할 수 있는 구조가 필요할 수 있다.

상태 변화 / 병합은 두 이력을 한 결과로 연결한다

병합은 두 이력을 한 결과로 연결한다

1 / 5   두 브랜치가 A에서 출발한다

공통 조상

A

main

A

feature

A

병합 결과

미생성

두 브랜치가 갈라진 경우의 병합 모형이다. 충돌 표시가 없어도 병합된 프로그램의 의미는 테스트해야 한다.

병합은 두 이력을 한 결과로 연결한다

5 / 5   M은 양쪽 부모 관계를 보존한다

공통 조상

A

main

M

feature

C

병합 결과

M의 부모 B C

두 브랜치가 갈라진 경우의 병합 모형이다. 충돌 표시가 없어도 병합된 프로그램의 의미는 테스트해야 한다.

두 브랜치가 갈라진 경우의 병합 모형이다. 충돌 표시가 없어도 병합된 프로그램의 의미는 테스트해야 한다.

- 1. 두 브랜치가 A에서 출발한다
- 2. main에 B가 추가된다
- 3. feature에 C가 추가되어 이력이 갈라진다
- 4. 공통 조상과 양쪽 변경으로 결과를 계산한다
- 5. M은 양쪽 부모 관계를 보존한다

5.1 Branch와 Merge와 충돌 해결

관련 근거 [R06] [R10] [R69]

54

## 충돌은 두 변경 사이의 결정을 요구한다

서로 다른 브랜치가 같은 부분을 다르게 고치면 Git이 최종 결과를 자동으로 결정하지 못할 수 있다. 충돌은 파일이 영구히 망가졌다는 뜻이 아니다. 사람이 두 의도를 읽고 합칠 내용을 정해야 한다.

단계	할 일	확인
시작 전	status와 현재 브랜치 확인	미커밋 작업 정리
병합	받을 브랜치에서 merge 실행	오류 메시지 읽기
해결	충돌 파일의 최종 내용을 편집	양쪽 의도 반영
완료	add 후 commit	충돌 표시 제거
검증	실행과 diff 확인	실제 동작 확인

```
<<<<<<< HEAD
현재 브랜치의 문장
=====
합쳐 오는 브랜치의 문장
>>>>>>> feature/topic
```

위 표시 세 종류와 중복된 내용을 정리해 최종 문장만 남긴다. 편집기의 Accept Current나 Accept Incoming을 뜻을 모른 채 누르지 않는다. 양쪽을 모두 고른 결과도 논리적으로 틀릴 수 있다.

Merge 중에는 git merge --abort로 시작 전 상태로 돌아갈 수 있다. 시작 전에 미커밋 변경을 정리해 두어야 취소와 복구를 더 예측하기 쉽다.

## 충돌 표시를 지우는 것과 해결의 차이

### 두 문장의 목적을 합쳐야 한다

한 브랜치는 정답 버튼의 이름을 바꾸고 다른 브랜치는 같은 버튼의 접근성 설명을 고쳤다고 하자. Current만 선택하면 설명이 사라질 수 있고 Incoming만 선택하면 이름 변경이 빠질 수 있다. 정답은 어느 쪽을 고르는지가 아니라 최종 동작이 두 요구를 만족하는가다.

## Merge 중 Current와 Incoming

이 장의 일반적인 Merge 상황에서 Current는 현재 체크아웃한 브랜치 쪽이고 Incoming은 합쳐 오는 쪽이다. Rebase에서는 도구가 보여 주는 ours와 theirs의 문맥이 다를 수 있으므로 같은 버튼 이름을 외워 모든 상황에 적용하지 않는다.

확인 단계	구체적으로 읽을 것
표식 안	양쪽 변경의 요구
표식 밖	연결 코드와 호출 위치
저장 후	충돌 표시가 남았는가
커밋 전	git diff --check와 status
커밋 후	실제 기능과 테스트

### 해결 완료의 기술적인 의미

파일을 편집해 저장한 뒤 add하면 해당 경로의 해결 상태를 Index에 반영한다. Merge라면 다음 커밋으로 통합을 마무리한다. Rebase와 Cherry-pick은 각 작업의 --continue를 사용하므로 진행 중인 명령 종류를 먼저 확인한다.

### 판단할 수 없으면

상황을 모른 채 한쪽을 모두 지우지 않는다. 시작 전에 작업을 보존했다면 해당 작업의 abort 경로를 검토할 수 있다. 양쪽 작성자에게 의도를 확인할 때에는 파일 이름과 충돌한 기능과 예상 최종 동작을 함께 전달한다.

자동 병합된 줄도 읽는다. 코드상 충돌이 없었다는 사실은 요구 사이의 충돌이 없다는 뜻이 아니다.

## 실습 5.1 | 로컬 전용 충돌 연습

아래는 실제 main을 건드리지 않는 별도 연습 폴더용이다. 충돌이 나면 평소 쓰는 편집기로 note.txt를 연다.

```
mkdir conflict-practice
cd conflict-practice
git init -b main
printf '시작 문장\n' > note.txt
git add note.txt
```

```
git commit -m 'Start note'
git switch -c branch-a
printf 'A의 설명\n' > note.txt
git commit -am 'Write A explanation'
git switch main
printf 'B의 설명\n' > note.txt
git commit -am 'Write B explanation'
git merge branch-a
```

충돌을 확인하고 note.txt를 편집해 A와 B의 의도를 함께 반영한다. 표식을 지운 뒤 아래 명령을 실행한다.

```
git add note.txt
git diff --check
git commit -m 'Combine both explanations'
git log --graph --oneline --all
```

확인 문제 충돌이 없으면 결과도 반드시 정확할까?  
 아니다. 자동 병합이 성공해도 기능의 의도가 충돌할 수 있다. 예를 들어 버튼 이름만 바뀌고 연결 코드는 옛 이름이면 실행이 깨진다. 검증은 별도로 필요하다.

## 5.2 Stash와 Rebase와 Cherry-pick

관련 근거 [R12] [R13] [R64]

### 서로 다른 목적의 세 도구

Stash는 진행 중인 변경을 잠시 치운다. Rebase는 커밋을 다른 기준 위에 다시 적용해 이력을 정리한다. Cherry-pick은 특정 커밋의 변경을 현재 브랜치에 새 커밋으로 적용한다. 이름이 비슷한 복구 도구처럼 섞어 쓰지 않는다.

상황	도구	주의
미완성 작업 중 다른 브랜치로 이동	stash	기본값은 미추적 파일 제외
개인 작업 브랜치의 기준 갱신	rebase	커밋 ID가 바뀔 수 있음
다른 브랜치의 수정 한 개 가져오기	cherry-pick	같은 변경을 중복 적용할 수 있음

```
git stash push -u -m 'WIP lesson note'
git stash list
git stash apply
확인 후 필요한 항목만 삭제
git stash drop 'stash@{0}'
```

apply는 적용 후 보관 항목을 유지한다. pop은 적용이 성공하면 항목을 제거한다. 둘 다 충돌할 수 있다. -u는 미추적 파일도 포함하지만 무시된 파일은 포함하지 않는다. Stash를 장기 백업으로 쓰지 않는다.

## 공유한 이력은 재작성 전에 합의한다

Rebase는 변경 내용을 다시 적용하므로 새 커밋 ID를 만든다. 남이 이미 기준으로 삼은 브랜치를 재작성하면 후속 작업이 복잡해진다. 기본 브랜치에서는 팀 정책 없이 사용하지 않는다.

## 임시 보관과 이력 재작성의 목적 구별

### Stash를 쓸 때 먼저 적을 것

어떤 일을 하던 중인지와 왜 잠시 치우는지 이름을 붙인다. Stash 목록이 여러 개라면 번호만 보고 옛 항목을 적용하지 않는다. 미추적 파일을 포함하려면 -u의 범위를 이해한다. 무시된 파일까지 자동으로 모두 보관되는 것은 아니다.

```
git stash list
git stash show --stat 'stash@{0}'
git stash apply 'stash@{0}'
```

적용이 끝난 뒤 파일을 확인하고 필요한 경우 항목을 정리한다. Apply는 항목을 유지한다. Pop은 성공적으로 적용하면 항목을 제거하는 흐름이다. 충돌이 발생할 수 있으므로 보관함을 장기 백업으로 믿지 않는다.

## Rebase가 바꾸는 것

작업 커밋의 변경을 새로운 부모 위에서 다시 적용한다. 새 기준과 충돌하는 부분은 사람이 결정해야 한다. 메시지가나 파일이 비슷해도 부모가 바뀌면 새 커밋이 될 수 있다. 이전 이력을 보존할 이름을 만든 뒤 결과를 비교한다.

## Cherry-pick은 복사 붙여넣기와 어떻게 다른가

특정 커밋의 변경을 현재 브랜치의 맥락에 적용하고 새 기록을 만든다. 다른 수정에 의존하는 커밋만 가져오면 기능이 완성되지 않을 수 있다. git show로 변경 내용을 읽고 필요한 선행 변경이 있는지 확인한다.

물음	먼저 고를 도구
진행 중인 편집을 잠시 치울까	Stash
개인 작업의 기준을 다시 잡을까	Rebase
특정 수정만 다른 브랜치에 적용할까	Cherry-pick

같은 오류를 고치는 도구 셋이 아니다. 무엇을 보존하고 어떤 이력 관계를 만들지부터 정한다.

## 실습 5.2 | 개인 브랜치 기준 갱신

lesson-lab에서 main과 작업 브랜치의 미커밋 변경이 없는 상태로 진행한다. 아직 공유하지 않은 개인 브랜치로 연습하는 편이 쉽다.

```
git switch feature/fetch-note
git status
git branch backup/before-rebase
git fetch origin
git rebase origin/main
```

충돌이 나면 파일을 정리한 뒤 git add 파일과 git rebase --continue를 실행한다. 결정을 미루려면 git rebase --abort로 취소한다. --skip은 그 커밋을 생략하므로 충돌을 피하는 만능 해법이 아니다.

## 특정 커밋 가져오기

```
git show COMMIT-ID
git cherry-pick COMMIT-ID
충돌 해결 후
git add CHANGED-FILE
git cherry-pick --continue
전체 적용을 취소하려면
git cherry-pick --abort
```

COMMIT-ID와 CHANGED-FILE은 실제 값으로 바꾼다. 주석 줄은 설명용이다. 해결 과정에서 continue와 abort를 연달아 실행하는 절차가 아니다.

확인 문제 Rebase한 개인 브랜치가 이미 원격에 있으면 일반 Push가 거절될 수 있을까? 그렇다. 커밋 ID가 바뀌어 이력이 달라질 수 있다. 16.1절의 명시적 lease 절차를 이해하기 전에는 강제 Push로 해결하지 않는다.

### 5.3 Merge와 Rebase를 그림으로 판단

관련 근거 [R12] [R69]

#### 출발 상태는 같아도 이력 결과는 다르다

main의 이력이 A에서 B를 거쳐 C까지 진행됐다고 하자. 작업 브랜치는 B에서 갈라져 D와 E를 만들었다. 두 브랜치는 B까지 같고 그 뒤 갈라졌다.

#### Merge의 결과

main에서 작업 브랜치를 Merge하면 C와 E를 부모로 하는 새 병합 커밋 M이 생길 수 있다. 기존 D와 E는 그대로 남는다. 공통 조상 B와 두 끝의 변경을 함께 고려한다.

#### Rebase의 결과

작업 브랜치를 C 위로 Rebase하면 D와 E의 변경을 다시 적용한 D2와 E2가 생긴다. 이때 main은 여전히 C다. 작업 브랜치를 Rebase했다는 이유만으로 main까지 이동하지는 않는다.

판단 항목	Merge	Rebase
기존 개인 커밋 ID	보존	일반적으로 새 ID
분기와 합류	명시적으로 드러남	선형 흐름에 가까움
공유 브랜치	통합 방식으로 자주 사용	재작성 합의 필요
기능 정확성	별도 검사 필요	별도 검사 필요

HTML의 이력 비교 도구는 위 개념을 설명하는 모형이다. 충돌 해결 내용과 복잡한 Merge 이력까지 실제 Git처럼 계산하는 엔진은 아니다.

#### 이력 모양을 부모 관계로 직접 설명

#### 그래프를 읽을 때 커밋 수보다 부모를 본다

일반 커밋은 앞선 기록을 부모로 가리킨다. Merge 커밋은 둘 이상의 부모를 가질 수 있다. 브랜치 이름은 끝을 가리키는 참조다. 커밋 자체와 브랜치의 현재 위치를 따로 설명해야 Rebase 후 main이 움직였는지 같은 질문에 답할 수 있다.

```
commit SAMPLE-ID
parent FIRST-PARENT-ID
```

parent SECOND-PARENT-ID

위 출력은 부모가 둘인 커밋의 형태를 설명하는 예시다. 실제 ID는 저장소마다 다르다. 부모가 둘이라는 관찰과 그 병합 내용이 올바르다는 판단은 별개다.

관찰	해석
main이 C를 가리킴	main의 끝은 아직 C
작업 브랜치가 E2를 가리킴	다시 적용한 이력의 끝
D와 D2의 ID가 다름	서로 다른 커밋 객체
파일 내용이 같음	커밋의 모든 정보가 같다는 뜻은 아님

### 충돌을 해결한 뒤 확인할 것

충돌 중에 선택한 내용이 원래 의도와 맞는지 검사한다. Rebase에서는 커밋을 차례로 다시 적용하므로 여러 번 멈출 수 있다. 각 단계에서 현재 충돌을 해결하고 진행한다. 끝났다는 메시지만 보지 말고 최종 로그와 diff와 실제 동작을 확인한다.

### 협업에 적용하는 기준

이미 여러 사람이 기준으로 삼는 브랜치라면 재작성의 영향을 먼저 검토한다. 개인 작업 브랜치 정리와 공유 브랜치 통합은 필요한 합의가 다르다. 이력이 읽기 쉬워지는 이점과 다른 사람의 작업 기준이 바뀌는 비용을 함께 판단한다.

실습 그림의 A와 B와 C는 개념 표시다. 실제 커밋 ID를 그대로 대입한 출력으로 오해하지 않는다.

### 실습 5.3 | 커밋 부모를 직접 비교

5.1절의 conflict-practice 또는 다른 연습 저장소에서 부모가 있는 커밋을 대상으로 진행한다. main에 실제 중요한 작업이 있는 저장소에서는 새 연습 폴더를 쓴다.

```
git log --graph --oneline --all
git show --no-patch --pretty=raw HEAD
```

첫 명령은 이력을 간단히 본다. 두 번째 명령은 현재 커밋의 부모와 작성 정보를 읽는다. 첫 커밋은 부모가 없고 보통의 커밋은 부모가 하나다. Merge 커밋은 부모가 여러 개일 수 있다.

## HTML 이력 비교 활동

1. 비교 도구에서 출발 상태를 선택한다.
2. main과 작업 브랜치의 끝을 가리킨다.
3. Merge를 선택하고 M의 부모 둘을 찾는다.
4. Rebase를 선택하고 main이 C에 남는지 확인한다.
5. D와 D2가 같은 변경 의도를 가지더라도 다른 커밋임을 설명한다.

확인 문제 Rebase한 브랜치의 최종 파일이 같으면 커밋 ID도 같아야 할까?  
부모 관계와 메타데이터가 바뀌면 커밋 ID도 달라진다. 파일 상태의 같음과 커밋 객체의 같음은 다르다.

## 6장 복구와 파일 관리와 이력 읽기

되돌릴 대상과 보존할 변경을 먼저 정한다. 무시 규칙과 줄바꿈과 큰 파일을 관리하고 로그와 Diff로 변경 이유를 조사한다.

### 되돌린다는 말에는 여러 작업이 들어 있다

파일 내용을 이전 상태로 복원하는 일과 브랜치 참조를 과거로 이동시키는 일 및 반대 변경을 새 커밋으로 기록하는 일은 서로 다르다. `restore`는 주로 경로의 내용을 다루고 `reset`은 모드에 따라 참조와 Index 및 작업 트리를 바꾸며 `revert`는 기존 변경을 상쇄하는 새 커밋을 만든다. 명령 이름보다 무엇이 바뀌는지 먼저 판단해야 한다. [T04]

`reset --soft`는 참조를 이동시키면서 Index와 작업 트리를 유지한다. 기본 `mixed` 모드는 Index도 목표 상태로 맞춘다. `hard` 모드는 작업 트리까지 바꾸므로 보존하지 않은 변경을 잃을 수 있다. 경로를 지정한 `reset`은 커밋을 이동시키는 형태와 동작이 다르다. 이 구분을 무시하고 모든 `reset`을 같은 복구 명령으로 이해하면 위험하다.

공유된 이력의 문제라면 `revert`가 협업에 유리할 수 있다. 과거 커밋을 지우지 않고 반대 변경을 기록하기 때문이다. 그러나 `merge commit`을 `revert`할 때는 주 부모를 선택하는 의미와 이후 재병합의 영향까지 이해해야 한다. 단순한 파일 복구와 같은 수준의 명령으로 취급하지 않는다.

### reflog는 중요한 단서지만 영구 백업은 아니다

`reflog`는 로컬 참조가 이동한 기록이다. 잘못된 `reset` 뒤 이전 커밋을 찾거나 `detached HEAD`에서 만든 커밋을 다시 이름 붙이는 데 도움이 된다. 기록은 저장소와 설정 및 보존 기간에 영향을 받으며 다른 사람의 서버 이력을 완전하게 대신하지 않는다. 참조되지 않는 객체는 정리될 수 있으므로 발견한 커밋은 필요한 브랜치나 태그로 보존한다. [T05]

복구 전에는 현재 상태를 먼저 기록한다. 작업 파일과 Index 및 커밋된 이력이 어디까지 보존되어 있는지 확인한다. 무시된 파일과 추적되지 않는 파일은 별도 사본이 필요할 수 있다. 문제를 고치려는 명령이 추가로 정보를 없애지 않도록 하는 것이 복구의 첫 원칙이다.

## 저장할 자료의 성격에 맞는 도구를 선택한다

Git은 텍스트 차이를 검토하는 데 강점이 있다. 대형 바이너리 파일은 작은 수정에도 큰 데이터가 추가될 수 있고 사람이 차이를 읽기 어렵다. LFS는 큰 파일 내용을 별도 저장소에 두고 Git에는 포인터를 저장하는 방식이다. 일반 Git 이력만 복사했다고 LFS 실제 객체까지 모두 백업된다고 가정하지 않는다.

줄바꿈과 인코딩은 내용이 같아 보여도 큰 차이를 만들 수 있다. .gitattributes로 저장소 차원의 정책을 남기면 개인 설정 차이로 발생하는 불필요한 변경을 줄일 수 있다. 정책을 바꿀 때 기존 파일 전체가 재정규화될 수 있으므로 기능 수정과 분리해 검토한다. 차이를 읽기 쉽게 만드는 것은 협업 비용을 줄이는 설계다.

## 파일이 안 보이는 것과 객체가 없는 것은 다르다

브랜치 이름을 삭제해도 그 이름이 가리키던 객체가 즉시 사라진다고 단정할 수 없다. 다른 참조가 도달하거나 reflog 등 보존 단서가 남아 있을 수 있다. 반대로 시간이 지나 정리 작업의 대상이 되면 복구에 사용할 객체가 없어질 수 있다. 현재 작업 트리에 없다는 사실과 객체 저장소에 없다는 사실을 나누어 생각한다. [T39]

복구할 가능성이 있는 커밋을 찾았다면 내용을 확인한 뒤 보존할 참조를 만드는 방법을 검토한다. 객체 식별자를 메모에만 적는 것과 저장소가 그 객체에 도달할 수 있는 참조를 유지하는 것은 다르다. 중요한 작업을 임시 기록의 우연한 보존에 의존시키지 않는다.

Git이 추적하지 않은 새 파일은 커밋 이력에서 복원할 수 없다. stash를 사용했더라도 어떤 범위가 포함되었는지 확인해야 한다. 복구 명령을 고르기 전에는 손실 대상이 커밋된 파일인지 미커밋 수정인지 추적되지 않은 자료인지부터 분류한다. 같은 실수처럼 보여도 복구 근거는 다르다.

## 정리 작업과 증거 보존의 목적을 혼동하지 않는다

저장소가 커졌다는 이유로 정리 명령을 반복하기 전에 조사 중인 객체가 있는지 확인한다. 저장 공간을 줄이는 목적과 과거 상태를 복원하는 목적은 충돌할 수 있다. 사고 조사 중에는 별도 사본을 확보하고 필요한 객체와 참조를 먼저 확인하는 편이 안전하다.

git fsck 같은 무결성 검사는 객체와 연결의 문제를 조사하는 데 도움이 된다. 그러나 저장소 구조가 정상이라는 결과는 소스의 기능이 정상이라는 뜻이 아니다. 데이터 구조의 건강과 프로그램의 정확성은 다른 검사 범위다. 보고서에서도 어떤 층을 확인했는지 적어야 결과를 과장하지 않게 된다.

## 6.1 실수 복구의 지도

관련 근거 [R14] [R15] [R16] [R17]

### 되돌리기 전에 무엇을 보존할지 정한다

작업 파일을 버릴지와 스테이징만 풀지와 커밋을 취소할지는 서로 다른 결정이다. 먼저 `git status`와 `git diff`와 `git diff --staged`를 읽는다. 중요한 미커밋 파일은 별도 복사도 한다. 백업 브랜치는 커밋만 보존하며 미커밋 파일까지 보관하지 않는다.

목적	명령의 예	남는 것
스테이징만 취소	<code>git restore --staged README.md</code>	편집 내용 유지
미스테이징 편집 폐기	<code>git restore README.md</code>	Index의 파일로 복원
커밋만 한 단계 이동	<code>git reset --soft HEAD~1</code>	Index와 작업 파일 유지
커밋과 Index 이동	<code>git reset --mixed HEAD~1</code>	작업 파일 유지
공유 커밋 취소	<code>git revert COMMIT-ID</code>	반대 변경의 새 커밋
잃어버린 참조 찾기	<code>git reflog</code>	로컬 참조 이동 기록

### 가장 조심할 동작

`git reset --hard`는 추적 파일의 작업 내용도 지정 상태로 바꾸며 일부 미추적 경로에도 영향을 줄 수 있다. `git clean -fd`는 미추적 파일을 지운다. 여기서는 실행 연습에 사용하지 않는다. 삭제 전 미리 보기는 `git clean -nd`.

복구의 한계: `reflog`는 로컬 참조 기록이다. 다른 PC의 기록이나 한 번도 커밋하지 않은 모든 파일을 되살리는 보증이 아니다. 보존 기간과 정리 상태에도 영향을 받는다.

### Restore와 Reset과 Revert의 대상을 정확히 고르기

#### 복구하고 싶은 상태를 먼저 적는다

파일을 다시 편집할 수 있게 Stage만 취소하려는지와 파일 내용 자체를 버리려는지와 이미 공유한 변경을 취소하려는지는 서로 다르다. 되돌린다는 말만으로 명령을 고르지 않는다.

원하는 결과	선택할 예	작업 파일의 영향
Stage만 풀기	<code>git restore --staged README.md</code>	편집 유지
미스테이징 수정 버리기	<code>git restore README.md</code>	Index 내용으로 변경
마지막 커밋만 풀기	<code>git reset --soft HEAD~1</code>	파일과 Index 유지
공유한 수정의 취소 기록	<code>git revert COMMIT-ID</code>	반대 변경을 적용

## 특히 놓치기 쉬운 Restore의 기준

HEAD가 v1이며 Index가 v2이고 작업 파일이 v3이라고 하자. 이때 `git restore README.md`는 기본적으로 Index의 v2를 작업 파일에 반영한다. 무조건 HEAD의 v1로 돌아가는 것이 아니다. 명령의 기본 원본과 변경 대상이 무엇인지 구분해야 한다.

## 백업 브랜치의 한계

`git branch backup/name`은 현재 커밋을 가리킬 이름을 남긴다. 아직 커밋하지 않은 v3 파일까지 묶어 보존하는 동작은 아니다. 편집 내용을 버릴 가능성이 있는 명령을 쓰기 전에는 미커밋 파일의 별도 보존도 확인한다.

## Revert의 실패도 이력 상태의 일부다

취소할 변경 이후 같은 부분이 많이 바뀌면 Revert도 충돌할 수 있다. 그때는 현재 요구와 취소 목적을 함께 읽어 해결한다. Merge 커밋의 Revert는 기준 부모 결정까지 필요하므로 단순한 HEAD 취소 예제를 그대로 확장하지 않는다.

비밀 노출은 Revert로 해결되지 않는다. 기록에서 반대 변경을 추가해도 과거 비밀은 남을 수 있다.

## 실습 6.1 | 공유 이력을 유지하며 취소

새 로컬 연습 저장소에서 커밋이 두 개 이상 있는 상태를 만든다. 실제 배포 브랜치 대신 연습 브랜치에서 실행한다.

```
git status
git log -5 --oneline
git branch backup/before-revert
git revert HEAD
git log -3 --oneline
git show HEAD
```

Revert가 편집기를 열면 기본 메시지를 확인하고 저장한 뒤 닫는다. Merge 커밋을 revert할 때는 기준 부모를 정해야 하므로 이 단순 예제를 그대로 적용하지 않는다.

### 커밋을 잃어버린 듯 보이면

1. 새 커밋과 정리 명령 실행을 멈춘다.
2. `git reflog --date=local`로 이동 기록을 읽는다.
3. `git show` 후보ID로 내용이 맞는지 확인한다.
4. `git branch rescue/found` 후보ID로 보존할 이름을 만든다.
5. 원래 작업으로 합칠 방법을 검토한다.

```
git reflog --date=local
git show CANDIDATE-ID
git branch rescue/found CANDIDATE-ID
```

확인 문제 Push한 잘못된 커밋은 reset 후 force로 지우는 것이 항상 좋은가?  
공유 브랜치라면 보통 revert로 취소 기록을 남기는 편이 협업에 유리하다. 비밀 유출은 revert로 비밀이 지워지지 않으므로 13.3절의 별도 대응이 필요하다.

## 6.2 .gitignore와 줄바꿈과 대용량 파일

관련 근거 [\[R18\]](#) [\[R19\]](#) [\[R20\]](#) [\[R67\]](#)

### 소스와 실행 환경과 결과물을 구분한다

.gitignore는 앞으로 추적하지 않을 경로의 규칙이다. 이미 추적 중인 파일을 자동으로 이력에서 지우지는 않는다. .env처럼 비밀이 들어가는 실제 파일과 .env.example처럼 빈 항목만 적은 예시를 구분한다.

```
.gitignore
.env
.env.*
!.env.example
node_modules/
dist/
*.log
.DS_Store
```

#### 파일

#### 일반적인 관리 방식

소스와 설정 예시

Git으로 추적

파일	일반적인 관리 방식
의존성 잠금 파일	재현성을 위해 보통 추적
설치된 라이브러리 폴더	추적하지 않음
개인정보와 실제 키	저장소에 넣지 않음
큰 영상과 배포 PDF	외부 보관이나 Release 검토

## 크기 한도와 LFS

공식 안내 기준 웹 추가는 파일당 25 MiB 이하다. 일반 Git 저장소에서는 100 MiB보다 큰 파일을 차단한다. Git LFS는 큰 파일의 실제 내용 대신 포인터를 Git에서 관리한다. LFS도 요금과 저장량과 전송량을 확인해야 하며 무제한 무료 저장소가 아니다.

이력의 과거 커밋에 큰 파일이 남아 있으면 현재 파일만 지워도 Push가 계속 실패할 수 있다. 공유 전 이력 정리와 공유 후 마이그레이션은 영향을 구분해야 한다.

## 무시할 파일을 정하고 이미 추적한 파일을 찾기

### 추적 여부와 무시 여부를 따로 확인한다

.gitignore는 앞으로 다룰 파일 선택을 돕는 규칙이다. 이미 커밋한 파일이 계속 변경 목록에 보인다면 규칙이 오작동한다고 생각하기 전에 그 파일이 추적 중인지 확인한다.

```
git ls-files .env
git check-ignore -v .env
```

첫 명령에 .env가 나오면 현재 Index에서 추적하는 경로다. 두 번째 명령은 무시 규칙의 적용 근거를 찾는 데 쓴다. 추적 중인 파일에는 일반적인 ignore 확인과 기대가 다를 수 있다. 출력 유무를 하나의 보안 검사로 사용하지 않는다.

### 실제 키 대신 설정 예시를 준다

.env.example에는 필요한 변수 이름과 가짜 값 또는 비워 둔 값을 적는다. 실행에 필요한 항목이 무엇인지 설명하면서 실제 키는 따로 주입한다. 예시라는 파일 이름만 붙였다고 안전해지는 것은 아니므로 내용도 확인한다.

## 큰 파일의 현재 상태와 과거 상태

큰 영상을 지운 다음 커밋을 만들었어도 과거 커밋이 그 파일을 포함하면 원격으로 보낼 데이터에 남아 있을 수 있다. 공유 전 연습 이력을 정리하는 일과 여러 사람이 받은 이력을 재작성하는 일의 영향은 다르다. LFS로 바꾸는 작업도 기존 이력의 마이그레이션 여부를 따로 정한다.

### 줄바꿈을 통일할 때

파일 전체가 변경돼 보이는 이유가 실제 내용인지 LF와 CRLF인지 확인한다. 팀의 .gitattributes 기준을 먼저 정한다. 정규화로 생긴 넓은 변경은 기능 수정과 나누어 검토하면 의도를 읽기 쉽다.

추적에서 빼는 것과 과거 기록에서 지우는 것과 유출 키를 폐기하는 것은 각각 다른 작업이다.

### 실습 6.2 | 추적 여부 확인과 줄바꿈 통일

1. .gitignore를 만들고 연습용 .env 파일에 가짜 값만 넣는다.
2. 아래 명령으로 무시 규칙이 적용되는지 확인한다.
3. .gitattributes에 팀의 줄바꿈 기준을 적는다.
4. diff에서 의도하지 않은 전체 파일 변경이 없는지 본다.

```
git check-ignore -v .env
git ls-files .env
```

```
.gitattributes
* text=auto
*.sh text eol=lf
*.yml text eol=lf
*.png binary
*.pdf binary
```

이미 추적 중인 파일을 추적에서만 빼려면 git rm --cached 파일을 사용한다. 로컬 파일은 남지만 다음 커밋부터 추적 대상에서 빠진다. 과거 이력은 그대로다. 실제 비밀 파일이었다면 먼저 키를 회수한다.

### LFS를 선택한 팀의 시작 예

```
git lfs install
git lfs track '*.psd'
git add .gitattributes
git lfs ls-files
```

확인 문제 .gitignore에 .env를 추가하면 과거에 공개한 키도 안전해질까?  
아니다. 이미 추적한 내용과 과거 커밋에는 그대로 남을 수 있다. 먼저 키를 폐기하거나 교체해야 한다.

## 6.3 로그와 Diff와 Git의 내부 구조

관련 근거 [R10] [R21] [R65] [R66]

### 커밋은 파일 묶음의 상태를 가리킨다

Git은 내용 기반 객체를 사용한다. Blob은 파일 내용이다. Tree는 파일 이름과 구조를 연결한다. Commit은 최상위 Tree와 부모 커밋과 작성 정보를 가리킨다. 브랜치와 태그는 이 객체를 찾기 위한 참조다.

커밋 ID는 작성 내용과 부모 관계와 메타데이터의 영향을 받는다. 문장이 같아도 부모나 메시지가 다르면 다른 커밋이 될 수 있다. Rebase 후 ID가 바뀌는 이유를 이 구조로 설명할 수 있다.

명령	찾는 질문
git log --graph --oneline --all	이력이 어떻게 갈라졌는가
git show COMMIT-ID	그 커밋에서 무엇이 달라졌는가
git log -- README.md	이 파일을 바꾼 기록은 무엇인가
git blame README.md	현재 줄은 어느 커밋에서 왔는가
git diff A B	두 시점의 파일 상태 차이는 무엇인가

### 두 점과 세 점의 차이

git log A..B는 B에서 도달 가능하지만 A에서는 도달 불가능한 커밋을 찾는다. git log A...B는 양쪽 중 한쪽에서만 도달 가능한 커밋을 찾는다. git diff A...B는 공통 조상부터 B까지의 파일 차이를 본다. 같은 점 기호라도 log와 diff의 뜻을 구분한다.

Blame은 사람을 찾는 도구가 아니다. 마지막 변경의 근거를 찾아 PR과 이슈를 따라가는 출발점이다. 줄 이동이나 포맷 변경 때문에 더 오래된 맥락을 찾아야 할 수 있다.

## 커밋을 따라가며 변경의 이유 찾기

### 로그는 정렬된 파일 목록이 아니다

커밋은 부모를 가리키며 이력을 연결한다. 브랜치 이름은 그 연결의 끝을 찾는 손잡이다. 단순히 최근 날짜 순서로만 읽으면 분기한 작업과 다시 합친 작업의 관계를 놓칠 수 있다.

```
git log --graph --oneline --decorate --all
git log -p -- README.md
git show HEAD
```

첫 명령은 여러 참조의 관계를 본다. 두 번째는 README와 관련된 패치를 따라간다. 마지막은 현재 커밋을 읽는다. 그래프의 가지 모양만 외우기보다 각 커밋의 부모와 어떤 이름이 끝을 가리키는지 확인한다.

### 파일 차이와 커밋 목록의 질문

`git diff A B`는 두 상태의 파일 차이를 물을 때 쓴다. `git log A..B`는 B에만 속하는 커밋을 찾는 질문이다. 최종 파일이 같아도 여러 경로로 이력을 만들 수 있으므로 두 결과가 같을 필요는 없다.

### Blame에서 멈추지 않는다

현재 줄을 마지막으로 바꾼 커밋은 포맷 변경이나 코드 이동일 수 있다. 해당 커밋의 PR과 더 이전 기록을 읽어 원래 요구를 찾는다. 오류 원인을 사람 이름으로 대체하면 재발 방지에 필요한 맥락을 놓친다.

### 과거 상태에서 실험할 때

Detached HEAD에서 읽기만 했다면 원래 브랜치로 돌아오면 된다. 새 커밋을 만들었다면 떠나기 전에 브랜치 이름을 붙여 보존한다. 파일의 옛 상태를 열어 봤다는 사실만으로 원격이나 main이 바뀌지는 않는다.

커밋 ID의 짧은 표시는 읽기 편한 축약이다. 자동화와 검증에서는 요구한 정확한 식별자와 범위를 사용한다.

## 실습 6.3 | 커밋에서 이유까지 추적

```
git log --graph --oneline --decorate --all -20
git log -p -- README.md
git blame README.md
git show HEAD:README.md
git cat-file -t HEAD
git cat-file -p HEAD
```

cat-file은 객체를 관찰하는 고급 명령이다. 마지막 출력에서 tree와 parent와 author를 찾아본다. 첫 커밋에는 parent가 없을 수 있다. Merge 커밋에는 부모가 여러 개 있을 수 있다.

## Detached HEAD

특정 커밋을 직접 체크아웃하면 브랜치가 아닌 커밋에 HEAD가 붙을 수 있다. 과거 상태를 살펴보는 데 유용하다. 그 상태에서 새 작업을 보존하려면 브랜치를 만든다.

```
git switch --detach HEAD~1
과거 파일을 읽는다
git switch -c experiment/from-past
또는 원래 브랜치로 돌아간다
git switch main
```

HEAD~1은 부모가 있는 커밋에서만 쓸 수 있다. 두 선택지를 순서대로 모두 실행하라는 뜻은 아니다.

확인 문제 브랜치 이름을 지우면 해당 커밋이 즉시 모두 삭제될까?

이름을 지우는 것과 객체를 즉시 지우는 것은 다르다. 다른 참조와 reflog가 가리키거나 아직 정리되지 않았다면 남을 수 있다. 하지만 이를 영구 보관 방식으로 믿어서는 안 된다.

## 7장 이슈 설계와 코드 리뷰

관찰 가능한 완료 조건을 정하고 재현 정보를 양식으로 받는다. 요구와 구현과 검증 근거를 연결해 PR을 검토한다.

### 좋은 이슈는 원인 단정보다 재현 조건을 제공한다

이슈의 목적은 작성자의 확신을 전달하는 것이 아니라 다른 사람이 같은 문제를 관찰할 수 있게 하는 것이다. 실행 환경과 입력 및 기대 결과와 실제 결과를 분리한다. 원인에 대한 추측은 별도로 적는다. 검색 라이브러리가 잘못되었다는 제목보다 과목을 바꾼 뒤 결과 수가 갱신되지 않는다는 설명이 조사에 더 직접적일 수 있다.

Issue Forms는 필요한 정보의 누락을 줄이는 장치다. 모든 항목을 필수로 만들면 보고 비용이 커져 사용자가 문제를 남기지 않을 수 있다. 재현에 꼭 필요한 정보와 선택적인 진단 자료를 구분한다. 비밀과 개인정보가 포함된 로그를 올리지 않도록 안내하는 것도 양식 설계의 일부다.

### 리뷰의 목적에는 이해와 대안 탐색이 포함된다

Bacchelli와 Bird의 연구는 Microsoft의 여러 팀을 관찰하고 인터뷰 및 설문과 리뷰 댓글 분석을 수행했다. 연구는 결함 탐지 외에도 지식 전달과 팀의 상황 이해 및 대안 탐색이 리뷰의 중요한 결과임을 설명한다. 이는 특정 조직의 경험적 연구이며 모든 리뷰 절차의 효과를 동일하게 보장하지는 않는다. [S13]

lesson-lab의 필터 PR에서 검토자는 조건식의 오류뿐 아니라 과목 이름이 어디에서 정의되는지와 새 과목을 추가할 때 무엇을 바꿔야 하는지도 물을 수 있다. 이런 질문은 작성자를 평가하기 위한 심문이 아니라 유지보수에 필요한 지식을 공유하는 과정이다. 설명이 코드와 다르면 둘 중 어느 쪽을 수정해야 하는지 함께 판단한다.

리뷰 댓글은 결함과 제안 및 질문을 구분하면 처리하기 쉽다. 반드시 수정해야 하는 동작 오류와 개인적인 표현 선호를 같은 강도로 전달하면 우선순위가 흐려진다. 근거와 영향을 설명하고 가능한 대안을 제시한다. 작성자는 수정 여부뿐 아니라 수용하지 않은 이유도 기록해 다음 검토자가 같은 논의를 반복하지 않게 한다.

## 리뷰 단위는 사람이 이해할 수 있는 변화다

PR 크기는 파일 수나 줄 수만으로 판단하기 어렵다. 자동 생성 파일 수천 줄보다 권한 조건 한 줄이 더 위험할 수 있다. 변경의 목적 수와 의존성 및 검증 난이도를 함께 본다. 여러 목적이 섞인 PR은 분리하고 하나의 목적을 구현하는 데 필요한 변경은 함께 보여 준다.

검토가 끝난 뒤 새로운 커밋이 추가되면 이전 승인과 현재 코드 사이의 관계를 확인해야 한다. 필수 검사와 승인 무효화 정책은 이 문제를 도구로 보완한다. 그러나 설정이 있다는 이유만으로 리뷰 내용의 질이 보장되지는 않는다. 정책은 사람이 판단할 시점을 제공하고 실제 판단은 변화의 의미를 읽는 과정에서 이루어진다.

## 변경 줄 수와 검토 난이도는 비례하지 않는다

설정 한 줄이 운영 계정의 권한을 확대할 수 있고 수백 줄의 생성 파일 변경은 원본 규칙 한 개에서 비롯될 수 있다. 변경량은 검토의 입력이지만 위험의 직접적인 척도는 아니다. 외부 입력과 권한 및 데이터 형식과 동시성 경계를 건드리는지 함께 살핀다.

검토 자원을 배정할 때 변경의 되돌림 가능성도 고려한다. 정적 문구는 이전 파일로 되돌리기 쉽지만 데이터 마이그레이션은 소스만 되돌려 해결되지 않을 수 있다. 같은 규모의 PR이라도 실패가 지속되는 방식에 따라 필요한 증거가 달라진다. 위험도에 비례해 검토 깊이를 정하면 모든 변경에 같은 긴 목록을 적용하는 것보다 실용적이다.

리뷰어가 코드를 읽는 순서도 중요하다. 목적과 제약 및 검사 결과를 먼저 이해한 뒤 구현을 읽으면 요구와 무관한 세부에 시간을 덜 쓸 수 있다. 작성자는 변경 이유와 대안 및 남은 한계를 짧게 제공한다. 설명이 구현과 어긋나는 경우에는 설명의 자연스러움보다 실제 diff와 실행 결과를 우선한다.

## 댓글이 해결되었다는 상태와 문제 해결을 구분한다

검토 대화의 해결 표시는 논의의 상태다. 제기한 문제가 수정되었거나 합의된 대안으로 처리되었다는 증거를 함께 봐야 한다. 권한 거부 시험을 요청한 댓글이 닫혔는데 실제 시험 기록이 없다면 검토의 목적은 충족되지 않았다. 화면의 완료 상태가 판단을 대신하지 않게 한다.

리뷰 대기 시간을 줄이려면 재촉만 하기보다 변경의 크기와 담당 범위 및 필요한 전문성을 조정한다. 요청판의 권한 변경을 화면 배치 수정과 함께 묶으면 적절한 리뷰어를 찾기 어렵다. 하나의 목적과 하나의 검증 질문에 집중된 변경은 검토자의 인지 부담을 줄인다.

팀의 성과를 리뷰 댓글 수로 평가하면 불필요한 지적이 늘 수 있다. 반대로 댓글이 적다는 이유로 검토가 부실했다고 단정할 수도 없다. 리뷰 후 재작업과 결함 및 이해의 전달을 함께 본다. 숫자는 활동의 흔적이며 그 활동이 해결한 문제를 대신하지 않는다.

상태 변화 / 검토는 특정 변경과 검사 결과를 연결한다

검토는 특정 변경과 검사 결과를 연결한다

1 / 5 초안을 열어 변경 목적을 설명한다

작업 브랜치  
C1

PR  
초안

자동 검사  
대기

기준 브랜치  
B

이 그림은 정책을 설명하는 모형이다. 승인 이후 새 커밋이 추가되었을 때 필요한 재검토 조건은 저장소 규칙으로 정한다.

검토는 특정 변경과 검사 결과를 연결한다

5 / 5 병합 방식에 따른 기준 브랜치 결과를 확인한다

작업 브랜치  
C2

PR  
병합 완료

자동 검사  
해당 결과 확인

기준 브랜치  
M

이 그림은 정책을 설명하는 모형이다. 승인 이후 새 커밋이 추가되었을 때 필요한 재검토 조건은 저장소 규칙으로 정한다.

이 그림은 정책을 설명하는 모형이다. 승인 이후 새 커밋이 추가되었을 때 필요한 재검토 조건은 저장소 규칙으로 정한다.

- 1. 초안을 열어 변경 목적을 설명한다
- 2. 코드 차이와 검사 결과를 함께 본다
- 3. 새 커밋이 추가되면 검토 대상이 바뀐다
- 4. 현재 커밋에 대한 조건 충족을 확인한다
- 5. 병합 방식에 따른 기준 브랜치 결과를 확인한다

# 7.1 이슈와 Projects로 일을 설계한다

관련 근거 [R22] [R23]

## Issue는 작업을 완료 가능한 크기로 정의한다

문제가 생긴 상황과 기대한 결과와 실제 결과와 재현 절차를 적는다. 기능 요청은 누가 어떤 일을 하기 어려운지부터 쓴다. 해결 방법을 먼저 고정하기보다 완료 조건을 명확히 하면 검토가 쉬워진다.

- ## 상황
- 휴대전화에서 정답 버튼을 찾기 어렵다.
- ## 기대 결과
- 작은 화면에서도 답 확인 동작이 드러난다.
- ## 완료 조건
- 360px 너비에서 버튼이 보인다.
  - 키보드로 버튼에 접근할 수 있다.
  - 정답을 열어도 풀이 영역이 잘리지 않는다.

## Project는 여러 작업을 보는 관점이다

Projects는 Issue와 PR을 표와 보드와 로드맵 등으로 묶는다. 저장소 폴더와는 다르다. 같은 일을 다시 복사해 적기보다 기존 Issue를 연결하고 상태와 담당자와 우선순위를 관리한다.

필드	예시	용도
Status	Todo / In progress / Done	진행 상태
Priority	High / Normal / Low	먼저 할 일
Area	콘텐츠 / 접근성 / 배포	작업 영역
Iteration 또는 날짜	검토 주간	일정 보기

완료는 코드가 작성됐다는 뜻만이 아니다. 검토와 확인과 필요한 배포까지 어떤 단계를 포함할지 팀이 정한다.

## 완료 조건을 재현 가능한 문장으로 바꾸기

### 예쁘게 만들기보다 관찰할 행동을 정한다

정답 버튼을 보기 좋게 고친다는 요구만 있으면 작성자와 검토자가 서로 다른 완성을 떠올릴 수 있다. 화면 크기와 조작 방식과 보일 결과를 적으면 같은 조건으로 확인할 수 있다.

모호한 표현	확인 가능한 표현의 예
모바일에 맞게	360px 너비에서 버튼과 풀이가 잘리지 않음
접근성 개선	Tab으로 도달하고 Enter로 동작함
오류 해결	제시한 재현 순서에서 오류가 다시 나지 않음
자료 정리	시작 문서에서 각 자료 링크가 열림

### 담당과 상태와 기한은 다른 필드다

담당자는 누가 책임질지 나타낸다. 상태는 현재 어느 단계인지 나타낸다. 날짜는 언제 확인할지 나타낸다. 담당자를 붙였다고 작업이 시작된 것은 아니며 Done으로 바꿨다고 배포가 완료된 것도 아니다.

### PR 연결의 효과

Issue는 왜 필요한가를 보관하고 PR은 어떤 변경으로 해결했는가를 보관한다. 둘을 연결하면 나중에 기능을 고칠 때 요구와 구현을 함께 찾을 수 있다. 자동 닫기 키워드가 작동할 조건과 단순한 번호 언급은 구분한다.

### 새로운 문제가 보이면

현재 Issue의 범위를 끝없이 넓히지 않는다. 지금 완료 조건에 필요한 수정인지 판단하고 독립적인 개선은 별도 Issue로 기록한다. 관련 작업을 링크하면 놓치지 않으면서 검토 범위를 유지할 수 있다.

교사 공동 작업에도 같은 원리가 적용된다. 자료 수정의 대상과 확인 기준을 적으면 제작자 외의 사람이 검토하기 쉬워진다.

## 실습 7.1 | 수업 도구 개선 보드

1. lesson-lab에 정답 버튼 개선 Issue를 만든다.

2. 오류 재현과 완료 조건을 적는다.
3. 개인 Project를 만들고 Issue를 추가한다.
4. Todo와 In progress와 Done 상태를 준비한다.
5. 담당자를 지정하고 작업 시작 시 상태를 바꾼다.
6. 이후 PR 본문에 Closes #번호를 넣는다.

## 자동 닫기 키워드

Closes #12처럼 적은 PR이 기본 브랜치로 병합되는 등 GitHub의 연결 조건을 만족하면 해당 Issue를 닫을 수 있다. 단순 언급인 #12와 자동 닫기 키워드는 구분한다. 다른 저장소는 OWNER/REPO#12처럼 대상을 명시한다.

## 알림을 관리하는 습관

관련 없는 저장소까지 모두 Watch하면 중요한 요청이 묻힌다. 참여 중인 PR과 담당 Issue와 필요한 Release 알림부터 조절한다. 알림은 읽었는지와 실제로 해결했는지를 구분한다.

확인 문제 Project의 카드 상태를 Done으로 바꾸면 코드도 병합될까?  
아니다. 상태 필드는 관리 정보다. PR 병합과 배포는 별도 동작이다. 자동화 규칙이 있다면 어떤 사건에 어떤 상태를 바꾸는지 확인해야 한다.

## 7.2 Issue Forms로 재현 정보를 빠짐없이 받기

관련 근거 [R80]

### 빈 대화창 대신 필요한 입력을 받는다

오류 보고에 실행 환경과 재현 절차가 빠지면 질문을 반복해야 한다. Issue Form은 필수 입력과 선택 항목을 제공한다. 입력을 받는 형식일 뿐 민감 정보 탐지나 검증을 자동으로 완성하지는 않는다.

```
.github/ISSUE_TEMPLATE/bug.yml
name: 오류 보고
description: 재현할 수 있는 오류를 알려 주세요.
title: '[오류] '
body:
 - type: input
 id: environment
 attributes:
 label: 실행 환경
 description: 운영체제와 브라우저 버전을 적어 주세요.
```

```

 validations:
 required: true
- type: textarea
 id: reproduce
 attributes:
 label: 재현 절차
 description: 한 단계씩 적어 주세요.
 validations:
 required: true

```

이 예제는 실제로 동작하는 최소 양식이다. 다음 페이지의 추가 항목을 body 아래에 이어 붙일 수 있다. labels를 미리 정의하지 않았는데 예시만 따라 넣으면 예상한 분류가 안 될 수 있으므로 여기서는 생략했다.

## Issue Form을 입력 규칙과 처리 절차로 연결

### 양식이 좋은 보고를 자동으로 만들지는 않는다

재현 절차와 기대 결과와 실제 결과를 나누어 받으면 조사에 도움이 된다. 그러나 필수 입력란만 늘리면 작성자가 아무 내용이나 넣을 수 있다. 담당자가 실제로 판단하는 데 쓰는 항목을 우선한다.

YAML 항목	설명할 책임
name과 description	어떤 제보에 쓰는 양식인가
body의 type	어떤 형태로 답하게 하는가
id	입력을 구분하는 고유한 이름인가
attributes	사용자에게 어떤 안내가 보이는가
validations.required	꼭 필요한 정보인가

### 작성자 관점에서 시험

새 Issue 작성 화면에서 양식이 나타나는지 본다. 빈 필수 항목으로 제출을 시도해 동작을 확인한다. 예시 내용으로 하나를 작성한 뒤 읽는 사람이 재현할 수 있는지 살핀다. 시험 Issue는 시험임을 표시하고 처리한다.

## 오류가 날 때 확인하는 순서

파일이 `.github/ISSUE_TEMPLATE` 아래에 있는지 확인한다. YAML 들여쓰기와 배열 구조를 검사한다. type에 필요한 attributes가 맞는지와 id가 중복되지 않는지 확인한다. YAML 문법 검사를 통과해도 GitHub 양식 규칙에 맞는지는 별도 확인이 필요하다.

## 민감한 제보를 받는 경로

공개 Issue에 비밀과 개인정보를 붙이지 않도록 안내한다. 보안 취약점은 저장소의 보안 정책과 비공개 제보 경로를 따르도록 설명한다. 양식의 안내 문장만으로 이미 올라온 민감한 내용이 보호되지는 않는다.

양식 도입 뒤 실제 처리 시간을 본다. 빈 칸이 줄었는지보다 재현과 분류와 답변이 쉬워졌는지 평가한다.

## 실습 7.2 | 기대 결과와 정보 공개 확인 추가

아래 항목을 기존 body 목록 끝에 추가한다. 별도 YAML 파일로 만드는 것이 아니다.

```
- type: textarea
 id: expected
 attributes:
 label: 기대 결과와 실제 결과
 validations:
 required: true
- type: checkboxes
 id: privacy
 attributes:
 label: 공개 전 확인
 options:
 - label: 비밀 키와 개인정보를 넣지 않았습니다.
 required: true
```

1. 양식 파일을 기본 브랜치에 추가한다.
2. Issues의 New issue에서 오류 보고 양식을 연다.
3. 필수 항목을 비운 상태의 안내를 본다.
4. 가짜 오류로 한 번 제출하고 출력 형식을 확인한다.
5. 실습 보고라는 표시를 남기고 정리한다.

## 양식 설계의 기준

처음 제보하는 사람도 답할 수 있는 질문을 쓴다. 로그 전체를 무조건 붙이게 하지 않는다. 계정 ID나 키나 내부 주소가 섞일 수 있기 때문이다. 취약점 보고는 SECURITY.md의 비공개 경로로 안내한다.

확인 문제 개인정보 미포함 체크를 받으면 제보 내용에 개인정보가 없는 것이 보장될까?  
사용자의 확인을 돕는 절차일 뿐이다. 운영자는 필요한 검토와 삭제와 사고 대응 절차를 별도로 갖춰야 한다.

## 7.3 좋은 PR과 좋은 코드 리뷰

관련 근거 [R06] [R24]

### 검토자가 판단할 근거를 준다

좋은 PR은 변경 범위가 설명 가능하고 왜 필요한지 드러난다. 동작 변경과 대규모 포맷 변경을 섞으면 중요한 차이를 읽기 어렵다. 기능을 완성하기 전에도 Draft PR로 방향을 공유할 수 있다.

PR 항목	답을 내용
문제	어떤 상황에서 누가 막히는가
변경	결과가 어떻게 달라지는가
검증	실제로 실행한 방법과 결과
한계	아직 확인하지 못한 조건
관련 Issue	추적할 근거와 완료 조건

### 리뷰 순서

먼저 요구와 설계를 읽고 전체 파일 구조를 본다. 다음으로 기능과 오류 처리와 권한과 데이터 흐름을 읽는다. 마지막으로 표현과 이름과 스타일을 본다. 줄 단위 지적만 하면 목적과 구조의 오류를 놓치기 쉽다.

리뷰 의견은 관찰한 상황과 영향을 설명하고 필요한 조치를 적는다. 필수 수정인지 제안인지 질문인지 구분한다. 검토자는 Approve와 Comment와 Request changes의 의미를 알고 선택한다.

테스트를 했다는 문장만 적지 않는다. 어떤 입력과 환경에서 어떤 결과를 확인했는지 적는다. 캡처는 화면을 보여 주지만 서버 권한이나 저장 성공을 증명하지 못할 수 있다.

## 리뷰를 요구와 구현과 증거로 나누기

### 먼저 문제를 재현한다

PR을 열자마자 사소한 이름부터 지적하지 않는다. 연결된 Issue의 문제를 읽고 바뀔 기능이 어떤 상황을 개선하는지 확인한다. 가능하면 바뀌기 전의 문제와 바뀐 뒤의 결과를 같은 절차로 비교한다.

검토 층	확인 질문
요구	완료 조건을 해결하는가
구현	데이터 흐름과 오류 처리가 맞는가
권한	허용한 사람만 작업할 수 있는가
검증	주장한 결과를 뒷받침하는가
운영	배포와 복구 시 영향이 무엇인가

### 화면과 서버를 구분한 검증

화면에서 관리자 버튼을 숨겼다는 사실은 서버가 작업을 거절한다는 증거가 아니다. 권한이 필요한 기능은 요청을 받는 신뢰할 수 있는 쪽에서 검사해야 한다. PR 설명에도 단순 화면 확인과 실제 권한 확인을 나누어 적는다.

### 의견을 남길 때

어느 조건에서 어떤 문제가 생길 수 있는지와 왜 고쳐야 하는지를 설명한다. 필수 수정인지 개선 제안인지 질문인지 드러낸다. 작성자는 수정 커밋과 확인 결과를 답하고 검토자는 남은 조건을 확인한다.

### 병합 직전의 마지막 확인

검토 뒤 새 커밋이 추가됐는지 본다. 실패한 검사나 남은 대화가 있는지 읽는다. 실제로 실행하지 못한 검증을 했다고 적지 않는다. 검증 환경의 한계를 알면 다음 검토자가 확인할 일을 이어받을 수 있다.

PR의 길이를 줄이는 목적은 대화를 숨기는 것이 아니다. 한 변경의 근거를 검토자가 찾기 쉽게 만드는 것이다.

### 실습 7.3 | 내 변경을 검토 가능한 PR로

1. 정답 버튼 Issue를 위한 브랜치를 만든다.
2. 완료 조건에 필요한 최소 변경을 커밋한다.
3. PR 본문에 아래 양식을 채운다.
4. Files changed를 본인이 먼저 읽는다.
5. 검토 의견을 반영하고 어떤 점을 고쳤는지 답한다.

#### ## 문제와 결과

작은 화면에서 정답 버튼을 찾기 어려웠다.  
버튼을 문제 바로 아래 배치했다.

#### ## 검증

- 360px와 1280px 화면에서 확인했다.
- Tab과 Enter로 열 수 있었다.
- 정답을 두 번 열고 닫아도 내용이 유지됐다.

#### ## 관련 작업

Closes #12

### 리뷰 후 새 커밋

필수 승인 후 변경이 추가되면 다시 검토가 필요한지 확인한다. 보호 규칙에 따라 기존 승인이 무효화될 수 있다. Resolve conversation은 의견 대화를 정리하는 기능이며 실제 수정이 맞다는 보증이 아니다.

확인 문제 초록색 검사와 Approve가 있으면 반드시 병합 버튼이 활성화될까?  
다른 필수 조건이나 충돌이나 브랜치 규칙이 남아 있을 수 있다. PR 하단의 병합 가능 상태와 요구 사항을 확인한다.

## 8장 오픈소스 기여와 팀 권한

Fork의 원본과 내 작업을 구분하고 조직의 역할을 설계한다. Rulesets와 CODEOWNERS가 실제 검토 흐름에서 작동하는지 시험한다.

### Fork는 별도 저장소이며 권한도 별도로 존재한다

Fork는 원본과 관계를 가진 다른 저장소다. 로컬 Clone은 저장소를 내 컴퓨터로 가져오는 행동이다. 두 개념을 혼동하면 어느 원격에 Push할지와 어느 저장소에서 PR을 만들지 헷갈리기 쉽다. 자신의 Fork를 origin으로 두고 원본을 upstream으로 추가하는 관례는 이름을 통해 역할을 구분하는 방법이다. 이름 자체가 권한을 부여하지는 않는다.

오픈소스 기여에서는 코드를 작성하기 전에 프로젝트의 기여 지침과 이슈 상태를 읽는다. 유지관리자가 원하지 않는 큰 변경을 먼저 완성하면 서로의 검토 비용이 커진다. 작은 문제의 범위와 수용 조건을 합의한 뒤 변경을 제안하는 과정은 기술적 능력만큼 중요하다.

### 권한은 사람과 자원 및 행동의 관계다

조직의 팀은 사람을 묶는 단위이며 저장소 역할은 허용된 행동의 범위를 정한다. CODEOWNERS는 특정 경로의 검토 책임자를 표현한다. 그러나 소유자 파일이 있다는 사실만으로 필수 승인이 자동으로 강제되는 것은 아니다. 적용되는 브랜치 정책과 승인 요구를 함께 확인해야 한다.

Ruleset은 어떤 참조에 어떤 조건을 적용할지 정하는 정책이다. 적용 대상과 예외 및 우회 권한이 함께 중요하다. 규칙의 이름이 엄격하더라도 실제 기본 브랜치가 대상에서 빠져 있으면 기대한 통제가 작동하지 않는다. 관리자가 우회할 수 있는 경로와 서비스 계정의 권한도 검토 범위에 포함한다.

최소 권한은 사람을 믿지 않는다는 선언이 아니다. 계정 탈취와 실수 및 자동화 오류가 발생했을 때 영향을 제한하는 구조다. 검토만 하는 사람에게 관리 권한까지 줄 필요는 없다. 자동화는 필요한 저장소와 작업 및 유효 기간에 맞는 자격 증명을 사용한다.

## 운영 규칙은 도구 설정과 문서가 일치해야 한다

문서에는 두 명의 승인이 필요하다고 적었지만 실제 규칙은 한 명만 요구한다면 팀의 계약이 두 개로 갈라진다. 반대로 도구가 필수 검사를 요구하는데 해당 검사가 더 이상 실행되지 않으면 병합이 막힐 수 있다. 정책 변경에는 문서와 설정 및 테스트 PR을 함께 검토한다.

거버넌스는 작은 팀에도 필요하다. 누가 릴리스를 결정하고 누가 비밀을 교체하며 누가 장애 때 연락받는지 정한다. 모든 권한을 한 사람에게 모으면 평소에는 편리할 수 있지만 부재와 계정 문제에 취약해진다. 역할의 분리와 대체 경로를 프로젝트 규모에 맞게 설계한다.

## 정책이 있다고 해서 모든 경로에 적용되는 것은 아니다

브랜치 정책은 대상 패턴과 실행 주체 및 예외 설정에 따라 효력이 달라진다. main에 적용한 규칙이 release 브랜치에도 적용되는지와 관리자나 자동화 계정이 우회할 수 있는지를 확인해야 한다. 조직의 ruleset과 저장소의 규칙이 함께 작동할 때도 실제 적용 범위를 살핀다. 정책 문서의 표현과 서비스의 설정을 대조하는 작업이 필요하다. [T40]

규칙을 강화한 뒤 정상 작업이 막힌다고 모든 예외를 열어 주면 정책의 목적이 사라진다. 막힌 경로가 설계된 예외인지 기존 자동화의 잘못된 가정인지 구분한다. 배포 봇이 직접 main에 쓰는 관행을 바꾸어 PR과 검사 경로로 이동하는 것이 더 적절할 수 있다. 예외는 문제를 해결하는 여러 선택지 중 하나다.

정책의 요소	확인할 질문
대상	어떤 저장소와 브랜치 및 태그에 적용되는가
주체	사람과 앱 및 토큰 중 누구에게 적용되는가
조건	검사와 승인 및 서명이 어떤 시점에 필요한가
예외	누가 어떤 이유로 우회할 수 있는가
증거	실제 허용과 거부를 어디에서 확인할 수 있는가

## 긴급 변경도 사후 재현이 가능해야 한다

장애를 복구하기 위해 긴급 권한이 필요할 수 있다. 이 경우 권한을 받은 사람과 대상 변경 및 사용 기간과 결과를 남긴다. 긴급 상황이 끝났는데 예외가 영구적으로 유지되면 이후의 위험이 커진다. 비상 경로는 평상시의 편의 경로와 구분해 관리한다.

이 원칙은 사람을 불신하기 위한 장치다. 당시의 제한된 정보와 시간 압박 속에서 내린 결정을 나중에 이해할 수 있게 한다. 복구가 끝난 뒤 정상 검사 경로에 해당 변경을 반영하고 누락된 검증을 수행한다. 긴급 조치의 성공과 정상 운영 상태의 회복은 별도의 완료 조건이다.

### 8.1 Fork와 오픈소스 기여

관련 근거 [R25] [R26]

#### 원본과 내 사본과 로컬을 구분한다

Fork는 다른 저장소를 내 계정 쪽으로 나누어 이어갈 수 있는 GitHub 저장소다. Clone은 저장소를 로컬에 복제하는 동작이다. Fork한 뒤 그 사본을 Clone해 작업할 수 있다. Fork 가능 여부와 접근 범위는 저장소와 조직 정책의 영향을 받는다.

이름	보통 가리키는 곳
upstream	원래 프로젝트
origin	내가 만든 Fork
local	내 컴퓨터의 Clone
PR의 Base	원본 프로젝트의 대상 브랜치

#### 기여 전 확인

LICENSE와 CONTRIBUTING과 CODE_OF_CONDUCT와 기존 Issue를 읽는다. 큰 기능을 먼저 구현하기보다 유지관리자가 원하는 방향인지 Issue로 논의한다. 처음에는 재현 가능한 오류 보고와 문서 오타 수정도 좋은 기여다.

다운로드할 수 있다고 자유롭게 재배포할 수 있는 것은 아니다. 라이선스가 없다면 사용과 수정과 배포 권한이 충분히 주어졌다고 가정하지 않는다. 자신의 기여와 프로젝트 라이선스의 조건을 함께 확인한다.

Private 저장소 Fork는 일반 Public 실습과 접근 규칙이 다를 수 있다. 회사나 학교 내부 코드를 개인 계정으로 옮기는 행동은 먼저 해당 정책을 확인한다.

## Fork 동기화에서 main을 깨끗하게 유지하기

### 세 장소의 역할을 이름으로 정한다

원본 저장소와 내 Fork와 로컬 Clone을 구분한다. 보통 origin은 내 Fork로 두고 upstream은 원본으로 정한다. 이 별칭 자체는 특별한 권한을 주지 않으며 사용자가 지정한 주소다.

작업	주로 향하는 곳
원본의 새 변경 수집	fetch upstream
내 작업 브랜치 공유	push origin
원본에 수정 제안	원본 Base로 PR
원본 기준 갱신	Fork의 main을 원본과 동기화

### Fork의 main에 독자 작업을 쌓으면

원본의 새 main과 내 main이 갈라질 수 있다. 그러면 단순한 fast-forward 동기화가 거절된다. 기능 수정은 별도 브랜치에서 하고 main은 원본을 따라가는 기준으로 유지하면 흐름을 설명하기 쉽다.

### 기여 범위가 맞지 않을 때

원본 프로젝트가 원하는 변경인지 확인한다. 프로젝트 방향과 다른 실험을 자신의 Fork에서 진행하는 것은 가능할 수 있지만 원본에 병합될 것을 미리 보장할 수는 없다. 라이선스와 기여 지침과 검증 절차를 읽는다.

### 외부 PR의 자동 검사가 제한된다면

처음 기여자의 실행 승인이 필요하거나 비밀이 전달되지 않는 등 보안 경계가 적용될 수 있다. 검사를 통과시키겠다는 이유로 권한 높은 이벤트에서 외부 코드를 실행하는 우회 경로를 만들지 않는다. 유지관리자의 정책에 맞는 검증 경로를 찾는다.

원본을 Fetch하는 일과 원본에 Push할 권한을 얻는 일은 다르다. 주소 별칭을 upstream으로 만들었다고 쓰기 권한이 생기지 않는다.

## 실습 8.1 | 안전한 Fork 동기화 흐름

본인 소유의 두 번째 연습 저장소나 기여를 허용한 프로젝트에서 진행한다. ORIGINAL-OWNER와 PROJECT는 실제 값으로 바꾼다.

```
git clone https://github.com/YOUR-ID/PROJECT.git
cd PROJECT
git remote add upstream https://github.com/ORIGINAL-OWNER/PROJECT.git
git remote -v
git fetch upstream
git switch main
git merge --ff-only upstream/main
git push origin main
git switch -c docs/fix-start-guide
```

Fork의 main에 독자 커밋을 쌓지 않으면 원본 동기화가 쉬워진다. main이 갈라졌다면 위 fast-forward가 실패할 수 있다. 원인을 확인하고 개인 변경을 별도 브랜치에 보존한다.

### PR을 보낼 때

내 Fork의 작업 브랜치에서 원본의 대상 브랜치로 제안한다. GitHub의 Base repository와 Head repository를 모두 확인한다. 외부 기여의 Actions는 실행 승인과 비밀 접근이 제한될 수 있다. 유지관리자의 보안 정책을 우회하려 하지 않는다.

확인 문제 원본이 바뀌면 내 Fork도 항상 자동으로 같아질까?  
자동으로 모든 변경이 합쳐지는 것은 아니다. 웹의 Sync fork나 로컬 fetch와 통합을 통해 동기화 상태를 관리한다.

## 8.2 조직과 팀 권한과 Rulesets

관련 근거 [R27] [R28]

### 사람 대신 역할에 권한을 연결한다

개인 저장소와 조직 저장소는 운영 방식이 다르다. 조직에서는 팀을 만들어 담당 범위에 맞는 권한을 부여하고 인원 이동 때 팀 구성원을 관리하는 편이 명확하다. 권한은 필요한 범위부터 부여한다.

조직 저장소 역할의 예	주된 목적
Read	읽기와 검토

조직 저장소 역할의 예	주된 목적
Triage	Issue와 PR 분류
Write	코드 기여
Maintain	일상적인 저장소 관리
Admin	민감한 설정과 접근 관리

역할별 상세 권한은 공식 표로 확인한다. 조직 소유자와 저장소 Admin과 프로젝트 관리자는 같은 뜻이 아니다. 학교 조직이라면 학생 개인 계정의 탈퇴와 담당 교사 교체까지 고려한다.

### Rulesets로 기본 브랜치를 보호한다

PR을 통한 병합과 필수 검사와 승인과 강제 Push 제한을 설정할 수 있다. Public과 Private와 요금제에 따라 제공 범위가 다르다. 관리자가 우회할 수 있는지와 예외 권한을 누가 가졌는지도 문서화한다.

한 명이 운영하는 저장소에 외부 필수 승인 1명을 무작정 요구하면 병합이 막힐 수 있다. 보안 목표와 실제 검토자를 함께 설계한다. 필수 검사 이름도 실제 실행되는 이름과 맞아야 한다.

### 보호 규칙을 켜기 전 작은 시험 PR 만들기

#### 규칙 이름보다 실제 차단 결과를 본다

main 보호라는 이름의 Ruleset을 만들었어도 대상 패턴이 틀렸거나 비활성 상태면 원하는 효과가 없다. 어느 브랜치에 어떤 상태로 적용되는지와 예외 주체를 확인한다.

시험 상황	확인할 기대
검사가 실패한 PR	필수 검사 조건 때문에 병합 제한
승인 전 PR	정한 승인 조건 표시
조건을 모두 충족한 PR	예상 역할의 병합 가능
관리자 또는 예외 주체	우회 정책과 기록 확인

## 검사 이름이 왜 중요한가

보호 규칙은 특정 검사의 완료를 기다릴 수 있다. Workflow나 Job 이름을 바꾸면 예전 이름을 기다리는 조건이 남을 수 있다. 경로 필터로 검사 자체가 실행되지 않을 때의 결과도 함께 확인한다.

## 사람 수와 승인 수를 맞춘다

필수 승인 조건을 만족시킬 실제 검토자가 필요하다. 자기 PR의 승인을 스스로 대신할 수 있다고 생각하지 않는다. 검토자가 없는 시간의 긴급 복구를 어떻게 다룰지도 정하되 일상적인 우회로 변하지 않게 한다.

## 규칙 변경도 운영 변경이다

갑자기 모든 저장소에 같은 규칙을 강제하면 배포와 자동 업데이트가 막힐 수 있다. 연습 저장소에서 시험한 뒤 대상과 영향과 되돌릴 방법을 기록한다. 보호를 강화하는 목적과 실제 업무 흐름을 함께 확인한다.

검토 기록과 예외 사유를 남긴다. 누가 우회했는지만 알기보다 어떤 문제를 복구했고 어떤 후속 검증을 했는지가 필요하다.

## 실습 8.2 | 연습 저장소에 보호 규칙 적용

1. Settings에서 Rules의 Rulesets를 확인한다.
2. main을 대상으로 하는 브랜치 규칙을 만든다.
3. PR 요구와 강제 Push 차단을 설정한다.
4. 이미 한 번 성공한 CI 검사 이름을 필수 검사로 고른다.
5. 검토자가 있다면 승인 조건을 추가한다.
6. 시험 PR로 막히는 조건과 허용되는 조건을 확인한다.

## 시행 전 검토표

항목	확인 질문
대상	main만 적용하는가
상태	실제 활성 상태인가
검사	경로 필터 때문에 영원히 대기하지 않는가
승인	승인 가능한 사람이 있는가
예외	긴급 복구 시 누가 어떤 기록을 남기는가

### 운영자 교체

개인 비밀번호를 넘겨주지 않는다. 새 담당자에게 조직 역할을 부여하고 자동화 소유자와 결제 담당과 도메인과 배포 권한을 각각 점검한다. 이전 담당자의 접근을 회수하기 전에 필요한 자원을 이전한다.

확인 문제 규칙을 만들었다면 관리자도 반드시 동일하게 차단될까?  
우회 설정과 역할과 규칙 종류에 따라 다르다. 규칙 존재 여부만 보지 말고 활성 상태와 예외 주체와 실제 시험 결과를 확인한다.

## 8.3 CODEOWNERS와 협업 양식

관련 근거 [R29] [R24]

### 검토 책임과 안내 문서를 저장소에 둔다

CODEOWNERS는 파일 경로별 검토 책임자를 지정한다. 담당자는 저장소 쓰기 권한을 가진 실제 사용자나 접근 가능한 팀이어야 한다. 이 책의 @YOUR-ID와 @YOUR-ORG/maintainers는 실제 계정으로 바꾼다.

```
.github/CODEOWNERS
* @YOUR-ID
/docs/ @YOUR-ORG/content-team
/.github/ @YOUR-ORG/maintainers
```

뒤에서 일치하는 규칙이 우선한다. 예를 들어 docs 파일은 별도 팀 규칙을 적용받는다. CODEOWNERS가 있다고 승인이 자동으로 필수가 되지는 않는다. 브랜치 보호나

Rulesets에서 코드 소유자 승인 요구를 별도로 설정한다. PR은 Base 브랜치 쪽 파일을 기준으로 책임자를 판단한다.

파일	역할
CONTRIBUTING.md	기여와 실행과 검증 방법
SECURITY.md	취약점 비공개 제보 경로
CODE_OF_CONDUCT.md	참여 규범과 대응 경로
.github/PULL_REQUEST_TEMPLATE.md	PR 작성 도움
.github/ISSUE_TEMPLATE/	오류와 제안 접수 양식

취약점의 실제 키와 악용 가능한 상세를 공개 Issue에 그대로 적게 만들지 않는다. 보안 제보 안내에는 공개하지 말아야 할 정보와 비공개 경로를 적는다.

## CODEOWNERS에서 경로 일치와 승인을 확인

### 뒤에서 맞는 규칙이 어떤 것인지 읽는다

CODEOWNERS의 여러 줄이 한 파일에 맞을 수 있다. 마지막으로 일치하는 규칙을 기준으로 책임자가 결정되므로 파일의 순서가 의미를 가진다. 폴더별 규칙을 적은 뒤 전체 경로 규칙을 맨 마지막에 두면 기대와 다르게 적용될 수 있다.

```
* @YOUR-ID
/docs/ @YOUR-ORG/content-team
/.github/ @YOUR-ORG/maintainers
```

이 예시에서 docs 아래의 파일은 해당 팀 규칙을 읽는다. 이름이 적힌 팀은 실제로 존재하고 필요한 저장소 권한을 가져야 한다. 예시 문자열을 바꾸지 않고 복사하는 것만으로 검토 책임자가 설정되지는 않는다.

### Base 쪽 정의가 중요한 이유

검토를 제안한 작업 브랜치에서 CODEOWNERS를 자신에게 유리하게 바꿨다고 Base의 검토 요구가 자동으로 대체되는 것은 아니다. 어떤 기준 브랜치로 PR을 여는지와 그쪽 책임 정의를 함께 확인한다.

## 양식과 강제 규칙의 차이

PR 양식에 확인란을 넣는 것은 작성과 검토를 돕는 안내다. 필요한 검사가 통과해야 병합할 수 있게 하려면 보호 규칙과 자동 검사가 별도로 맞아야 한다. 확인란을 눌렀다는 행위만으로 실제 테스트를 증명할 수는 없다.

## 운영 문서의 길이 조절

자주 빠지는 정보부터 양식에 넣는다. 기능 변경에만 필요한 상세 항목을 단순 오타 수정에도 모두 강요하면 의미 없는 복사가 늘어난다. 최소 공통 항목과 필요할 때 추가할 설명을 나누어 유지한다.

소유자 파일은 권한 부여 문서가 아니다. 실제 역할과 팀 접근과 승인 요구를 각각 설정하고 시험한다.

## 실습 8.3 | PR 기본 양식 만들기

1. 아래 내용을 `.github/PULL_REQUEST_TEMPLATE.md`에 저장한다.
2. 새 PR에서 양식이 나타나는지 확인한다.
3. 실제로 필요하지 않은 항목은 줄인다.
4. CODEOWNERS에는 본인 계정부터 넣고 검토 요청 동작을 확인한다.

## 문제와 결과

## 확인한 내용

- [ ] 바뀐 기능을 직접 실행했다.
- [ ] 필요한 문서를 수정했다.
- [ ] 비밀과 개인정보가 없는지 확인했다.

## 검증 방법과 결과

## 관련 Issue

## 배포 시 확인할 점

## 팀 양식이 길어지면 생기는 문제

모든 PR에 같은 항목을 강제로 채우게 하면 의미 없는 체크가 늘 수 있다. 문서 수정과 인증 변경은 필요한 근거가 다르다. 공통 최소 항목만 두고 복잡한 변경은 PR 본문에 추가 설명을 쓰게 한다.

확인 문제 CODEOWNERS에 계정을 쓰면 그 사람에게 쓰기 권한도 생길까?  
아니다. 접근 권한은 별도로 부여한다. 파일은 이미 권한이 있는 사람이나 팀의 검토 책임을 표현한다.

## 9장 큰 저장소와 병합 운영

저장소 구조와 체크아웃 범위를 목적에 맞게 선택한다. 병합 전략과 Merge queue로 여러 변경이 합쳐질 때의 검증을 관리한다.

### 큰 저장소의 비용은 파일 크기 하나로 설명되지 않는다

저장소의 무게에는 객체 수와 이력 길이 및 큰 파일과 참조 수가 영향을 준다. Clone 속도와 작업 트리 크기 및 검색 시간은 서로 다른 비용이다. shallow clone은 이력의 깊이를 줄이고 partial clone은 필요한 객체의 전송을 지연하며 sparse checkout은 작업 트리에 펼칠 경로를 제한한다. 이 기법들은 같은 문제를 해결하지 않는다.

lesson-lab이 여러 수업 도구를 포함하는 Monorepo로 커졌다고 하자. 한 팀은 공통 라이브러리 변경을 한 번에 검토하기 쉬워질 수 있지만 작은 앱 하나를 수정할 때도 큰 CI가 실행될 수 있다. 저장소 통합의 이익과 빌드 및 권한과 소유권의 비용을 함께 평가한다. 폴더를 합치는 것만으로 의존성 관리가 해결되지는 않는다.

Submodule은 다른 저장소의 특정 커밋을 상위 저장소에서 참조하는 방식이다. 하위 프로젝트의 최신 브랜치를 자동으로 따라간다고 생각하면 재현성이 깨진다. 상위 커밋이 가리키는 하위 커밋을 명시적으로 확인하고 초기화와 갱신 절차를 문서화한다. 별도 권한과 외부 저장소의 가용성도 의존성의 일부다.

### 병합 대기열은 조합의 검증 문제를 다룬다

PR A와 PR B가 각각 현재 main에서 통과해도 둘을 함께 넣은 결과가 통과한다는 보장은 없다. A가 라이브러리를 갱신하고 B가 이전 동작을 전제로 코드를 추가할 수 있기 때문이다. 병합 대기열은 기존 브랜치와 대기 중인 변경을 조합해 검사하는 절차를 제공한다. 필요한 워크플로가 대기열의 이벤트에서도 실행되는지 확인해야 한다.

통합 비용을 줄이려면 대기 시간을 관찰한다. 개발이 끝난 뒤 검토를 기다리는 시간과 CI 대기 및 재작업 시간을 구분한다. 커밋 작성 속도가 빨라져도 검토 병목이 그대로라면 전체 전달 시간은 줄지 않을 수 있다. 병렬 작업을 늘리는 결정은 통합 능력과 함께 평가해야 한다.

장기 브랜치는 큰 변경을 격리할 수 있지만 기존 브랜치와의 차이가 누적된다. 짧은 브랜치와 기능 플래그는 통합을 자주 하면서 공개 시점을 조절하는 다른 전략이다. 어떤 전략이 적합한지는 배포 구조와 테스트 수준 및 팀의 운영 능력에 달려 있다. 전략의 이름보다 통합 실패를 어떻게 발견하고 복구할지 설명할 수 있어야 한다.

## 이력과 객체와 작업 트리는 서로 다른 축이다

shallow clone은 가져오는 커밋 이력의 깊이를 제한한다. partial clone은 필요한 객체 일부를 나중에 가져오도록 제한할 수 있다. sparse checkout은 작업 트리에 펼칠 경로를 줄인다. 세 기능은 줄이는 대상이 다르므로 같은 최적화로 취급하지 않는다. blob:none 필터를 사용해도 checkout에 필요한 파일 내용은 결국 내려받을 수 있다. [T22]

방법	주로 줄이는 범위	검토할 제약
shallow	과거 커밋 이력	공통 조상과 태그 및 이력 조사에 필요한 정보
partial	일부 객체의 초기 전송	이후 객체 요청과 서버 지원 및 네트워크 의존
sparse	작업 트리에 펼친 경로	선택하지 않은 경로에 대한 도구의 가정

문서 사이트만 수정하는 작업과 과거 결함을 bisect로 조사하는 작업은 필요한 자료가 다르다. 전자는 일부 경로만 펼쳐도 충분할 수 있지만 후자는 긴 이력이 필요하다. 저장소를 작게 만드는 방법은 작업 목적에서 선택해야 한다. 처음의 다운로드 속도만으로 모든 상황에 같은 옵션을 적용하지 않는다.

## 최적화의 비용이 뒤로 이동할 수 있다

partial clone의 초기 다운로드가 빠르더라도 과거 파일을 자주 읽으면 이후 요청이 늘 수 있다. 오프라인 조사가 필요한 사람에게는 필요한 객체를 미리 확보하는 과정이 중요하다. 지금 사용하지 않는 자료를 나중에 가져오는 설계는 자료가 필요할 때 서버에 접근할 수 있다는 전제를 가진다.

CI의 짧은 작업에서는 제한된 이력이 합리적일 수 있다. 그러나 릴리스 버전을 태그에서 계산하거나 변경된 경로를 공통 조상과 비교한다면 부족한 이력이 잘못된 결과를 만들 수 있다. 실패하면 무조건 전체 clone으로 바꾸기보다 어떤 계산이 어떤 정보에 의존하는지 확인한다. 필요한 범위를 정확히 가져오면 속도와 정확성을 함께 관리할 수 있다.

관련 근거 [T10]

## 9.1 큰 저장소와 Monorepo와 Submodule

관련 근거 [R57] [R58]

### 저장소의 경계를 운영 목적에 맞춘다

Monorepo는 여러 구성요소를 한 저장소에 둔다. 공통 변경을 한 PR로 처리하기 좋지만 CI 범위와 권한 경계와 소유권을 설계해야 한다. 여러 저장소는 배포와 접근 권한을 분리하기 쉽지만 공통 변경의 조정 비용이 늘어난다.

전략	얻는 점	관리할 비용
Monorepo	공통 변경과 도구 통일	큰 CI와 영향 범위
여러 저장소	권한과 수명 분리	버전 간 호환 조정
Submodule	다른 저장소의 특정 커밋 참조	별도 초기화와 갱신
Subtree 또는 패키지	코드 공유의 다른 방식	동기화와 배포 규칙

### 필요한 내용만 가져오기

Partial clone의 `--filter=blob:none`은 파일 내용을 필요할 때 받도록 할 수 있다. Sparse checkout은 작업 폴더에 나타낼 경로를 줄인다. 서로 다른 기능이며 서버 지원과 도구 호환을 확인한다. `--depth 1`의 얇은 Clone은 과거 이력 분석에 제약이 생긴다.

### Submodule의 함정

부모 저장소는 하위 저장소의 특정 커밋을 기록한다. 하위 폴더에서 수정했다면 하위 저장소에 먼저 커밋과 Push를 하고 부모에서 참조 변경을 커밋해야 동료가 같은 상태를 받을 수 있다.

폴더를 옮기기 전에 접근 권한 요구를 점검한다. Monorepo 안의 특정 폴더만 일반 저장소 권한으로 비밀스럽게 숨길 수 있다고 가정하지 않는다.

## 저장소 크기와 권한과 배포 단위 함께 정하기

### 큰 저장소 문제를 Clone 속도만으로 보지 않는다

여러 도구를 한 저장소에 두면 공통 수정을 한 PR로 만들기 쉽다. 대신 어떤 파일 변경이 어느 검사와 배포에 영향을 주는지 설계해야 한다. 팀의 검토 책임과 접근 범위도 함께 정한다.

구조 선택	먼저 결정할 질문
Monorepo	공통 변경과 검사 범위를 어떻게 찾는가
여러 저장소	호환 버전과 동시 수정을 어떻게 조정하는가
Submodule	하위 커밋을 누가 발행하고 갱신하는가
패키지 공유	버전과 배포 권한을 어떻게 운영하는가

### Sparse와 Partial의 차이

Sparse checkout은 작업 폴더에 나타낼 경로를 줄인다. Partial clone은 필요한 객체를 받는 시점을 조절할 수 있다. 한 기능을 쓴다고 다른 기능의 효과까지 모두 얻는 것은 아니다. 네트워크 없이 과거 파일을 열 계획이라면 필요한 객체가 준비됐는지도 확인한다.

### Submodule 변경이 전달되는 두 단계

하위 저장소에서 코드를 커밋하고 접근 가능한 원격에 Push한다. 그 뒤 부모 저장소에서 하위 커밋 참조 변경을 커밋한다. 부모만 Push하고 하위 커밋을 보내지 않으면 동료가 참조 대상을 받지 못할 수 있다.

### CODEOWNERS와 폴더 접근

폴더별 검토 책임은 정할 수 있어도 그것이 일반적인 저장소 읽기 권한을 폴더별로 숨기는 장치라고 생각하지 않는다. 일부 팀만 읽어야 하는 비밀 자료라면 자원 경계와 저장소 구조를 별도로 설계한다.

구조 선택의 장점만 적지 않는다. 새 담당자가 실행하고 갱신하고 복구할 비용도 함께 비교한다.

### 실습 9.1 | 필요한 경로 중심으로 읽기

이 명령은 새 Clone 폴더용이다. docs와 .github가 있는 저장소를 대상으로 진행한다.

```
git clone --filter=blob:none --sparse https://github.com/YOUR-ID/
 lesson-lab.git lesson-sparse
cd lesson-sparse
git sparse-checkout set docs .github
git sparse-checkout list
git log -5 --oneline
```

Cone 모드에서는 최상위 파일도 보일 수 있다. Sparse는 다른 경로의 이력이 삭제됐다는 뜻이 아니다. 실제로 필요한 파일을 읽을 때 네트워크 요청이 생길 수 있다.

## Submodule 사용 팀의 Clone

```
git clone --recurse-submodules https://github.com/OWNER/PARENT.git
이미 Clone했다면
git submodule update --init --recursive
git submodule status
```

두 명령 흐름은 각각의 상황에서 선택한다. 연습 저장소에 Submodule이 없다면 억지로 추가하지 않는다. 배포 도구가 하위 저장소 인증과 초기화를 지원하는지도 확인한다.

확인 문제 하위 저장소에서 수정만 하면 부모 저장소를 Clone한 동료도 최신 내용을 받을까?  
하위 커밋을 접근 가능한 원격에 Push하고 부모가 그 커밋을 참조하도록 갱신해야 한다. 떠 있는 로컬 수정만으로는 전달되지 않는다.

## 9.2 Merge 전략과 Merge queue와 규모 확장

관련 근거 [R06] [R59] [R37]

### 병합 이력은 팀이 읽을 작업 기록이다

Merge commit은 분기와 합류를 보존한다. Squash merge는 PR의 변경을 기본 브랜치의 한 커밋으로 모은다. Rebase merge는 개별 변경을 새 커밋으로 이어 붙인다. 팀이 되돌리고 조사하고 배포하는 단위에 맞춰 선택한다.

전략	적합한 상황	주의
Merge commit	개별 커밋과 분기 맥락 보존	그래프가 복잡해질 수 있음
Squash	PR 하나가 작업 한 단위	작업 브랜치의 작은 이력은 합쳐짐

전략	적합한 상황	주의
Rebase merge	의미 있는 커밋의 선형 이력	커밋 ID 변화

Squash 후 같은 작업 브랜치를 계속 사용하면 이전 변경이 다시 비교에 나타날 수 있다. 보통 최신 main에서 새 브랜치를 만들어 다음 작업을 시작한다.

## 바쁜 기본 브랜치의 통합 문제

PR 두 개가 각각 통과해도 합쳐졌을 때 실패할 수 있다. Merge queue는 최신 기준과 대기 변경의 조합을 검증해 병합 순서를 관리한다. 지원 요금제와 저장소 유형을 확인한다. 필수 CI가 merge_group 이벤트도 처리해야 큐의 검증 요청에 응답할 수 있다.

큐를 켜는 일만으로 검사 품질이 좋아지지는 않는다. 통합 실패를 잡을 테스트와 안정된 검사 이름이 먼저 필요하다.

## 각 PR의 성공과 조합의 성공을 구분

### 독립 검사가 통합 오류를 놓칠 수 있다

PR A가 버튼 이름을 바꾸고 PR B가 옛 이름으로 새 기능을 연결했다고 하자. 각자의 기준에서 검사가 통과해도 두 변경을 합치면 동작이 깨질 수 있다. 바쁜 기본 브랜치에서는 최신 기준과 대기 변경을 함께 확인하는 절차가 필요하다.

### Merge queue가 기다리는 검사

큐에서 만든 검증 요청에 CI가 응답해야 한다. 일반 PR 이벤트만 처리하도록 설정했다면 merge_group에 필요한 검사 결과가 생기지 않을 수 있다. 이벤트와 검사 이름과 보호 조건을 함께 확인한다.

정책	운영에서 물을 질문
Squash 기본	PR 하나가 되돌리기 단위인가
Merge commit 기본	개별 커밋과 분기 맥락이 필요한가
Rebase merge 기본	커밋 열을 선형으로 읽을 준비가 됐는가
Queue 사용	조합 검사가 실제로 실행되는가

## 병합 전략을 바꿀 때

과거 커밋의 모양만 예쁘게 만드는 문제가 아니다. 오류 조사와 Revert와 릴리스 노트와 자동화가 어떤 단위를 사용하는지 본다. 한 저장소 안에서도 작업 종류에 따라 정책이 다를 수 있지만 선택 기준을 문서화한다.

## 실패한 큐 항목을 다룬다

다시 넣기만 반복하지 않는다. 조합에서 실패했는지와 간헐 테스트인지와 실제 기능 충돌인지 구분한다. 실패 로그와 관련 변경을 연결해 이후에도 같은 판단을 반복할 수 있게 한다.

Queue는 좋은 검사를 대신하지 않는다. 검증하지 않는 조건은 순서를 자동화해도 계속 놓칠 수 있다.

## 설계 실습 9.2 | 큐에 대응하는 CI

12.1절 CI의 on 블록에 merge_group을 추가하는 예시다. 큐를 쓰지 않는 저장소는 필수 실습이 아니다.

```
on:
 pull_request:
 push:
 branches: [main]
 merge_group:
 types: [checks_requested]
```

## 팀의 병합 정책 문서

1. 기본 병합 방법 하나를 정하고 이유를 적는다.
2. PR을 어느 크기로 나눌지 예시를 정한다.

3. 승인 뒤 변경이 생겼을 때 재검토 규칙을 적는다.
4. 실패하는 검사와 간헐 실패를 누가 처리할지 정한다.
5. 긴급 수정도 남겨야 하는 기록을 정한다.

### 지표 읽기

PR 개수와 커밋 수만으로 성과를 평가하지 않는다. 검토 대기 시간과 배포 실패율과 복구 시간과 반복 오류를 함께 본다. 오래 열린 PR은 작성자 문제뿐 아니라 검토 용량과 작업 범위의 문제일 수 있다.

확인 문제 PR마다 초록색이었다면 큐 검사는 불필요할까?

다른 변경과 합친 상태가 달라질 수 있다. 통합 조합의 검증과 각 PR 단독 검증은 서로 다른 근거다.

## 10장 웹 공개와 릴리스 관리

Pages의 경로와 배포 결과를 확인한다. 배포 Workflow를 구성하고 태그와 Release와 실행 중인 판본을 연결한다.

### 같은 판본이라는 말을 검증할 수 있어야 한다

커밋은 소스 상태를 식별한다. 태그는 특정 객체에 붙이는 이름이다. Release는 사용자에게 배포판을 설명하고 파일을 제공하는 GitHub의 객체다. Deployment는 실제 실행 환경에 결과를 배치한 기록이나 과정이다. 이들을 모두 버전이라고 부르면 어떤 대상을 비교하는지 모호해진다.

lesson-lab의 v1.0.0 태그가 커밋 A를 가리킨다고 하자. 그 소스로 만든 배포 파일과 현재 Pages 사이트가 반드시 같은 판본인지는 별도로 확인해야 한다. 태그 이름을 이동시키거나 빌드 설정이 달라지면 이름은 같아도 결과가 달라질 수 있다. 릴리스 정책에서는 공개한 이름의 변경 가능성과 아티팩트의 식별 방법을 명확히 한다.

배포 메타데이터에 소스 커밋과 빌드 실행 번호를 남기면 장애 때 현재 결과의 출처를 찾기 쉽다. 사용자가 보고한 화면이 어느 판본인지 모르면 이미 수정된 오류를 다시 조사하거나 다른 환경을 비교할 수 있다. 출처 기록은 대규모 서비스만의 관행이 아니라 작은 정적 사이트에도 유용하다.

### 정적 호스팅의 경계를 이해한다

GitHub Pages는 정적 사이트를 게시하는 용도로 사용한다. 브라우저에 전달된 JavaScript와 설정 값은 사용자가 읽을 수 있다. 따라서 비밀 API 키를 숨겨 두는 서버 역할로 사용해서는 안 된다. 클라이언트에서 외부 API를 부를 때는 공개해도 되는 정보와 서버를 거쳐야 하는 정보를 구분한다.

프로젝트 사이트의 경로 접두사는 자원 경로 문제를 만들 수 있다. 로컬에서는 /style.css가 작동하더라도 저장소 이름이 포함된 하위 경로에 배포하면 다른 위치를 가리킬 수 있다. 상대 경로와 빌드의 base 설정을 확인하고 실제 배포 주소에서 링크와 새로고침을 검사한다. 로컬 성공을 배포 성공으로 곧바로 해석하지 않는다.

롤백은 이전 결과를 다시 제공하는 절차다. 정적 사이트에서는 비교적 단순해 보이지만 외부 API 계약과 저장된 브라우저 데이터가 바뀌었다면 이전 코드와 호환되지 않을 수 있다. 소스만 되돌리는 절차와 사용자 데이터의 호환성을 함께 검토한다.

## 재빌드와 재현 가능한 빌드는 다른 수준이다

소스를 다시 실행해 비슷한 결과를 얻는 것과 같은 입력 조건에서 같은 바이트를 생성하는 것은 다르다. reproducible build는 명시된 소스와 환경 및 절차를 바탕으로 결과를 재현하는 관점에서 다뤄진다. 빌드 시각과 파일 순서 및 무작위 값이 산출물에 들어가면 코드가 같아도 해시가 달라질 수 있다. [T42]

교육용 PDF를 빌드한다고 하자. 내용이 같아도 메타데이터의 생성 시각이나 글꼴 버전이 달라지면 파일 바이트가 달라질 수 있다. 기능적으로 같은 결과인지와 바이트가 같은 결과인지 평가 목적을 먼저 정한다. 배포한 파일을 정확히 식별하려면 실제 산출물의 해시와 소스 커밋을 함께 기록한다.

소스 커밋만으로 배포 결과를 모두 설명할 수 없는 경우도 있다. 빌드 환경의 설정과 외부 자료 및 의존성 버전이 결과를 바꿀 수 있기 때문이다. 릴리스 기록에는 이 입력들을 연결한다. 어떤 파일을 배포했는지 모르면 소스가 정상이어도 사용자가 받은 결과를 조사하기 어렵다.

## 배포와 공개를 나누면 되돌림의 범위가 보인다

새 파일을 서버에 올리는 배포와 사용자에게 기능을 노출하는 공개는 분리할 수 있다. 예를 들어 새 검색 계산을 배포해도 일부 사용자에게만 활성화할 수 있다. 문제가 생겼을 때 기능 노출을 끄는 방식은 파일 전체를 다시 배포하는 방식과 다른 복구 경로다. 어느 방법이 필요한지는 변경의 성격과 운영 구조에 달려 있다.

데이터 형식을 함께 바꾼 배포에서는 이전 코드가 새 자료를 읽을 수 있는지 확인한다. 코드만 이전 판본으로 되돌리면 오히려 실패할 수 있다. 릴리스의 단위가 소스와 설정 및 자료 중 어디까지 포함되는지 정해야 한다. 되돌릴 수 있다는 주장은 실제로 호환되는 상태로 돌아갈 수 있을 때 성립한다.

## 10.1 GitHub Pages로 HTML 수업 도구 공개

관련 근거 [R30] [R31]

### 정적 파일을 웹 주소로 제공한다

GitHub Pages는 HTML과 CSS와 JavaScript 같은 정적 사이트에 적합하다. 브라우저에서 작동하는 수업 시각화와 안내서와 포트폴리오를 배포할 수 있다. 서버에서 비밀 키를 숨기거나 데이터베이스 쓰기 권한을 처리하는 기능은 별도 설계가 필요하다.

사이트 종류	주소의 일반적인 형태
사용자 사이트	<code>https://YOUR-ID.github.io/</code>
프로젝트 사이트	<code>https://YOUR-ID.github.io/lesson-lab/</code>

사용자 사이트의 저장소 이름은 `YOUR-ID.github.io`다. 프로젝트 사이트는 일반 저장소 이름으로 만든다. 이 책은 `lesson-lab` 프로젝트 사이트를 사용한다. Public 저장소의 무료 출발 경로를 기준으로 하며 Private의 지원은 요금제를 확인한다.

### 경로와 공개 범위를 확인한다

`assets/app.css`처럼 상대 경로를 쓰면 프로젝트 하위 경로에서 찾기 쉽다. `/assets/app.css`처럼 슬래시로 시작하면 도메인 루트에서 찾으므로 프로젝트 사이트에서 깨질 수 있다. 빌드 도구를 쓰면 `base` 경로 설정을 맞춘다.

Pages 주소의 접근 범위는 소스 저장소의 Private 설정과 같다고 가정하지 않는다. 별도 접근 통제 지원 여부를 확인한다. 민감한 자료를 올리지 않는다. 무료 상업 서비스 호스팅용으로 쓰지 않도록 Pages 이용 제한도 읽는다.

### Pages 주소와 파일 경로를 함께 계산

#### 프로젝트 사이트에는 저장소 경로가 붙는다

도메인이 `YOUR-ID.github.io`이고 저장소가 `lesson-lab`이라면 프로젝트 사이트의 시작 경로는 보통 `/lesson-lab/`이다. 브라우저가 그림과 스타일을 어느 위치에서 찾는지 이 기준으로 생각한다.

HTML에 쓴 경로	프로젝트 화면에서 찾는 위치
<code>assets/app.css</code>	현재 사이트 아래 <code>assets</code>
<code>./assets/app.css</code>	현재 사이트 아래 <code>assets</code>
<code>/assets/app.css</code>	도메인 루트의 <code>assets</code>

두 번째와 세 번째의 차이는 점과 슬래시다. 루트 경로가 꼭 틀린 것은 아니지만 그 위치에 실제 파일이 있어야 한다. 프로젝트 하위 경로를 사용하는 사이트에서는 필요한 설정과 달라질 수 있다.

## 배포 성공과 화면 성공을 나눈다

Workflow가 성공해도 index.html이 잘못된 폴더에 있거나 내부 경로가 틀리면 원하는 화면이 안 나올 수 있다. 배포한 커밋과 결과 폴더와 주소를 확인한 뒤 브라우저에서 버튼과 그림과 링크를 직접 확인한다.

## 주소를 직접 입력하면 실패하는 경우

브라우저에서 화면 내부 이동은 되는데 특정 하위 경로를 새로 열면 404라면 단일 페이지 앱의 라우팅과 정적 호스팅 경로를 점검한다. 사용한 프레임워크의 base와 경로 처리 방식에 맞는 배포 설계가 필요하다. 무조건 파일을 다시 Push하는 일만으로 해결되지 않을 수 있다.

## 데이터 저장을 추가하려면

브라우저 화면을 공개하는 일과 비공개 데이터의 저장 권한을 설계하는 일은 다르다. 비밀을 HTML에 숨겨 넣는 방식으로 서버 권한을 대체하지 않는다. 필요한 서비스의 인증과 저장 정책을 별도로 정한다.

소스 저장소의 Private 설정만 보고 배포 화면도 비공개라고 단정하지 않는다. 실제 배포 주소의 접근 조건을 확인한다.

## 실습 10.1 | 첫 정적 페이지 배포

1. lesson-lab의 main에 index.html을 만든다.
2. 아래 코드를 저장하고 커밋한다.
3. Settings의 Pages에서 Deploy from a branch를 선택한다.
4. Branch를 main과 /(root)로 지정하고 저장한다.
5. Actions 또는 배포 상태가 끝나면 Pages 설정에 표시된 주소를 연다.

```
<!doctype html>
<html lang='ko'>
<meta charset='utf-8'>
<meta name='viewport' content='width=device-width'>
<title>수업 실험실</title>
<h1>수업 실험실</h1>
<p>GitHub Pages 배포 연습</p>
<button id='answer'>정답 보기</button>
<p id='result' hidden>연습 정답은 42다.</p>
<script>
document.getElementById('answer').onclick = () => {
 document.getElementById('result').hidden = false;
};
```

```
</script>
</html>
```

## 404 진단 순서

배포 완료 여부를 본다. 파일 이름이 index.html인지와 선택한 폴더가 맞는지 확인한다. 사용자 사이트와 프로젝트 사이트의 주소를 혼동하지 않는다. 수정 후 새 배포가 끝나기까지 잠시 걸릴 수 있다.

확인 문제 Pages에서 로그인 버튼을 그리면 안전한 로그인 기능이 생길까?  
화면만 생긴다. 인증과 권한 검사는 신뢰할 수 있는 서버와 인증 서비스 등에서 별도로 구현해야 한다. 브라우저 코드는 사용자가 읽고 바꿀 수 있다.

## 10.2 Pages 배포를 Workflow로 관리

관련 근거 [R79] [R37]

### 내보낼 폴더를 별도로 만든다

Pages 배포는 저장소 전체를 올리는 작업과 다르다. 공개할 파일만 _site 폴더에 모아 Artifact로 업로드한다. 여기서는 10.1절의 index.html 한 개를 제공하는 사이트를 가정한다. 그림과 스타일을 별도 파일로 썼다면 필요한 공개 파일도 명시적으로 복사한다.

```
.github/workflows/pages.yml의 앞부분
name: Publish lesson page
on:
 push:
 branches: [main]
 workflow_dispatch:
permissions:
 contents: read
concurrency:
 group: lesson-pages
 cancel-in-progress: false
jobs:
 build:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v7
 - run: |
 mkdir -p _site
 cp index.html _site/index.html
 - uses: actions/upload-pages-artifact@v4
 with:
```

```
path: _site
```

위 내용과 다음 페이지 코드를 하나의 `pages.yml`로 연결한다. 다음 페이지 deploy는 build와 같은 jobs 아래의 형제 항목이다.

Settings의 Pages에서 Source를 GitHub Actions로 선택한다. 기존 브랜치 배포 방식과 현재 사용하는 방식을 혼동하지 않는다. 공개할 수 없는 파일을 Artifact에 넣지 않는다.

## Pages 배포에서 소스와 결과물을 구분

### 저장소에 있는 모든 파일이 배포되는 것은 아니다

Workflow가 어느 폴더를 산출물로 묶는지 확인한다. 소스가 맞아도 잘못된 폴더를 올리면 페이지가 없거나 이전 구조로 보일 수 있다. 정적 사이트 생성기를 사용한다면 소스 폴더와 빌드 결과 폴더가 다를 수 있다.

단계	성공을 판단할 근거
Checkout	의도한 커밋을 받았는가
검사 또는 빌드	실패 없이 필요한 파일을 만들었는가
Upload artifact	정확한 결과 폴더를 묶었는가
Deploy	배포 작업과 환경이 성공했는가
주소 접속	실제 경로와 기능이 동작하는가

### 첫 페이지는 열리는데 이미지가 깨질 때

이미지 파일이 결과물에 포함됐는지 확인한다. 파일 이름의 대소문자와 상대 경로를 본다. 프로젝트 사이트의 경로 아래에서 루트 기준 주소를 사용하면 예상과 다른 위치를 가리킬 수 있다. 개발 PC에서 열리는 경로와 배포 주소의 경로를 비교한다.

### 실행은 성공했는데 옛 내용이 보일 때

배포한 커밋과 보고 있는 주소를 대조한다. 작업 브랜치와 배포 대상 브랜치를 혼동하지 않았는지 확인한다. 그 뒤 브라우저 캐시와 중간 빌드 결과를 확인한다. 먼저 모든 캐시를 지우기보다 어느 단계에서 옛 내용이 남았는지 찾는다.

## 권한 오류를 읽는 방법

토큰 권한과 배포 환경과 실행 이벤트를 확인한다. 필요한 범위를 맞추는 대신 모든 권한을 넓히지 않는다. PR의 검사와 작업과 신뢰한 브랜치의 실제 배포 작업을 나누어 생각하면 외부 기여의 실행 범위를 정하기 쉽다.

Pages의 정적 파일은 방문자에게 전달될 수 있다. 브라우저에서 실행하는 코드나 HTML에 서버 비밀을 넣지 않는다.

## 실습 10.2 | 배포 Job 연결

```
deploy:
 needs: build
 runs-on: ubuntu-latest
 permissions:
 pages: write
 id-token: write
 environment:
 name: github-pages
 url: ${ steps.deployment.outputs.page_url }
 steps:
 - name: Deploy
 id: deployment
 uses: actions/deploy-pages@v4
```

1. 앞 페이지와 이 코드를 들여쓰기 그대로 합친다.
2. 설정의 Pages Source를 GitHub Actions로 바꾼다.
3. main에 병합한 뒤 build와 deploy의 순서를 확인한다.
4. Environment와 배포 URL과 커밋을 확인한다.
5. 페이지를 열어 정답 버튼을 눌러 본다.

## 왜 needs와 권한을 분리하는가

needs가 없으면 Artifact가 준비되기 전에 배포가 따로 시작될 수 있다. 검사와 빌드에는 읽기 권한을 주고 배포 Job에만 Pages 쓰기과 OIDC 요청 권한을 준다. 실제 보호 규칙이 허용하는 브랜치인지도 확인한다.

## 배포가 실패하면

업로드한 Artifact 경로와 index.html의 위치를 본다. Pages 설정과 Environment 보호 규칙을 확인한다. 다른 사이트 생성기를 사용한다면 빌드 결과 폴더와 프로젝트 base 경로도 맞춘다.

확인 문제 저장소 루트 전체를 Artifact로 보내면 항상 더 편하고 안전할까?  
불필요한 소스와 설정이 공개될 수 있다. 배포에 필요한 공개 결과물만 별도 폴더로 모으는  
편이 범위를 확인하기 쉽다.

## 10.3 Tag와 Release와 배포의 차이

관련 근거 [R32] [R33]

### 같은 버전을 다시 찾아 실행할 수 있게 한다

Tag는 이력의 특정 지점에 붙인 이름이다. Release는 태그를 바탕으로 변경 설명과 내려받을 파일을 제공한다. Deployment는 실제 실행 환경에 배포하는 과정과 상태다. Release를 만들었다고 사이트가 자동으로 갱신되는 것은 아니다.

구분	답하는 질문
Commit	어떤 변경 상태인가
Tag	그 상태의 버전 이름은 무엇인가
Release	사용자는 무엇을 받는가
Deployment	어느 환경에서 실행 중인가

버전 정책의 예로 v1.2.3을 쓸 수 있다. 호환성을 깨는 변화와 기능 추가와 수정의 구분을 팀에서 정한다. 문서나 교육 자료도 배포일과 수정 번호와 변경 내역을 함께 남기면 잘못된 판본을 줄일 수 있다.

### 릴리스 노트에 담을 것

새 기능과 수정과 호환성 변화와 업데이트 방법을 적는다. 개발 커밋 제목을 모두 나열하기보다 사용자의 행동이 달라지는 점을 설명한다. PDF와 HTML을 함께 배포한다면 두 파일이 같은 원본과 같은 버전인지 확인한다.

공개한 버전 태그를 임의로 다른 커밋으로 옮기면 내려받을 사람마다 같은 버전의 내용이 달라질 수 있다. 수정 버전을 새로 발행하는 정책이 명확하다.

## 버전 이름과 파일과 실행 환경의 연결

### 태그를 만든 위치를 확인한다

버전 이름을 적기 전에 현재 브랜치와 커밋을 확인한다. 실수로 작업 중인 브랜치 끝에 태그를 붙이면 README의 설명과 실제 배포 파일의 내용이 어긋날 수 있다.

```
git status
git log -1 --oneline
git show --no-patch v0.1.0
```

위 태그가 존재하는 상황에서 마지막 명령으로 대상을 읽는다. Annotated tag라면 태그 자체의 정보와 대상 커밋을 함께 볼 수 있다. 버전 이름과 파일 생성일과 공개일이 항상 같아야 하는 것은 아니다.

### Release 첨부를 확인할 때

PDF와 HTML을 같은 판본으로 배포한다면 제목과 버전과 포함 기능이 일치하는지 본다. 내려받은 파일이 실행되는지도 확인한다. 소스 ZIP과 완성된 배포 파일을 받는 경로는 사용자에게 각각 설명한다.

### 배포 복구는 별도 질문이다

Release를 이전 버전으로 안내한다고 이미 실행 중인 서버가 자동으로 돌아가지는 않는다. 어느 배포 서비스가 어느 커밋이나 산출물을 사용하는지 확인해야 한다. 데이터 구조가 변했다면 코드만 되돌렸을 때의 호환성도 검토한다.

기록할 연결	남길 근거
소스 상태	커밋 ID
사람이 읽는 판본	태그와 Release 설명
실제 파일	파일 이름과 필요한 해시
실행 환경	배포 기록과 확인 주소

같은 공개 버전 이름의 내용을 조용히 바꾸지 않는다. 수정 판본과 변경 이유를 구분해서 남기면 사용자와 운영자가 같은 대상을 이야기할 수 있다.

### 실습 10.3 | 첫 배포판 만들기

main의 최신 내용과 검사 결과가 맞는지 확인한 뒤 진행한다. v0.1.0이 이미 있다면 새 버전을 선택한다.

```
git switch main
git pull --ff-only
git status
git tag -a v0.1.0 -m 'First learning release'
git push origin v0.1.0
```

웹의 Releases에서 Draft a new release를 연다. 위 태그를 선택하고 변경 설명을 적는다. 필요한 배포 파일을 첨부하고 공개한다. gh를 쓴다면 다음 방식도 가능하다.

```
gh release create v0.1.0 --verify-tag --title '첫 연습 배포판' --notes-file RELEASE_NOTES.md
```

RELEASE_NOTES.md를 미리 작성한다. --verify-tag는 원격에 태그가 있는지 확인하도록 한다. 파일 첨부는 배포 목적에 맞게 추가한다.

확인 문제 Release의 소스 ZIP에 비밀 설정까지 넣어야 실행이 편할까?

비밀은 배포 파일에 넣지 않는다. 필요한 환경 변수 이름과 설정 절차를 설명하고 실제 값은 안전한 실행 환경에 별도로 주입한다.

## 11장 클라우드 개발과 컨테이너 활용

브라우저 편집과 실행 환경의 차이를 이해하고 AI의 변경을 검토한다. 컨테이너 이미지와 패키지 공개를 실제 실행 환경과 연결한다.

### 편집 환경과 실행 환경을 구분한다

github.dev는 브라우저에서 저장소를 편집하는 경험을 제공하지만 완전한 원격 실행 환경과 동일하지 않다. Codespaces는 실행 자원과 개발 환경을 제공한다. 어느 도구에서 코드를 읽고 어느 환경에서 테스트를 실행했는지 구분해야 한다. 파일이 열렸다는 사실은 의존성 설치와 실행 검증이 끝났다는 뜻이 아니다.

개발 컨테이너 설정은 팀이 필요한 도구와 환경을 문서화하는 방법이다. 같은 설정을 사용하면 초기 준비의 차이를 줄일 수 있다. 그러나 외부 서비스와 비밀 및 하드웨어와 네트워크까지 동일하게 만드는 것은 아니다. 환경을 코드로 표현하는 목적은 재현성을 높이고 차이를 설명하기 쉽게 하는 데 있다.

### 컨테이너 이미지의 태그와 다이제스트를 구분한다

태그는 읽기 쉬운 이름이며 갱신될 수 있다. 다이제스트는 이미지 내용의 식별에 사용된다. latest라는 태그만 기록하면 나중에 같은 이름이 다른 이미지를 가리킬 수 있다. 배포와 조사에서 정확한 결과를 재현하려면 사용한 이미지의 식별자를 남긴다.

이미지 안에 비밀을 넣으면 나중에 파일을 삭제하는 단계가 있어도 이전 레이어에 남을 수 있다. 빌드 입력과 런타임 비밀을 분리하고 필요한 자격 증명을 제한된 방식으로 전달한다. 공개 저장소에 올리지 않았다는 사실만으로 이미지와 빌드 로그까지 안전하다고 판단하지 않는다.

AI 코딩 도구를 개발 환경에 연결할 때도 권한 경계를 확인한다. 읽기 권한과 파일 수정 및 명령 실행과 외부 전송은 서로 다른 능력이다. 저장소의 지시 문서와 검증 명령을 제공하되 외부 문서에 포함된 지시를 프로젝트 소유자의 승인처럼 해석하지 않도록 한다. 도구의 편의성과 실행 범위는 함께 설계해야 한다.

### 환경 선언은 실행될 프로그램의 일부다

개발 컨테이너 설정에는 이미지와 도구 및 생성 후 실행 명령 등이 포함될 수 있다. 저장소를 열 때 자동으로 수행되는 작업은 편리하지만 그 자체가 실행 코드다. 신뢰하지 않는 저장소에서 환경을 준비할 때는 설정 파일의 명령과 마운트 및 외부 네트워크 사용을 검토한다. 환경 파일은 단순한 설명문으로 취급할 수 없다. [T41]

컨테이너 안에서 실행한다는 사실만으로 호스트와 완전히 분리되었다고 가정하지 않는다. 작업 폴더와 자격 증명 및 소켓을 어떤 방식으로 연결했는지에 따라 접근 범위가 달라진다. 편의를 위해 넓은 마운트를 제공했다면 그 선택의 영향을 알아야 한다. 환경의 재현성과 실행의 격리는 서로 다른 품질 속성이다.

새 팀원이 환경을 열 수 있게 하려면 비밀 없이 재현되는 부분과 별도 발급이 필요한 부분을 구분한다. 개인의 인증 파일을 이미지에 넣으면 빠르게 시작할 수 있어 보여도 소유권과 만료 및 유출 대응이 어려워진다. 공통 도구는 선언하고 개인 권한은 실행 시점의 승인된 경로로 연결하는 편이 관리하기 쉽다.

## 잠금 파일은 공급망 전체의 스냅샷이 아니다

패키지 잠금 파일은 의존성 해결 결과를 고정하는 데 도움이 된다. 그러나 설치 스크립트가 외부 자료를 받거나 시스템 패키지가 바뀌거나 기본 이미지가 이동하면 결과가 달라질 수 있다. 어느 층을 고정했는지 확인해야 재현 실패의 원인을 좁힐 수 있다.

문제가 생기면 소스 변경과 잠금 파일 및 이미지 다이제스트와 실행 로그를 함께 비교한다. 한 층의 버전만 확인하고 동일한 환경이라고 결론 내리지 않는다. 완벽한 고정을 무조건 추구하기보다 현재 프로젝트에서 중요한 재현 수준과 갱신 주기를 정한다. 보안 수정과 재현성 사이의 운영 정책도 필요하다.

### 11.1 github.dev와 Codespaces와 AI 협업

관련 근거 [\[R34\]](#) [\[R35\]](#) [\[R36\]](#)

#### 편집 환경과 실행 환경을 구분한다

github.dev는 브라우저에서 파일을 편집하기 편한 환경이다. 일반적인 서버 터미널과 실행 환경이 항상 제공되는 것으로 생각하지 않는다. Codespaces는 클라우드 개발 환경으로 터미널과 실행 환경을 제공하며 기계 사용과 저장에 비용이 발생할 수 있다.

선택	알맞은 상황	먼저 확인
웹 편집기	파일 하나 수정	커밋 브랜치
github.dev	여러 파일 빠르게 읽고 편집	저장과 커밋 상태
Codespaces	설치 없이 빌드와 실행	비용과 포트 공개

선택	알맞은 상황	먼저 확인
로컬 VS Code	장기 작업과 로컬 자원	Git과 인증

Codespaces에서 브라우저 탭을 닫는 것과 환경을 중지하는 것은 같지 않을 수 있다. 작업을 Push한 뒤 중지 상태와 보관 중인 환경을 확인한다. 컴퓨팅과 저장 사용량을 구분한다.

### AI에게는 검토 가능한 작업을 맡긴다

작업 목표와 완료 조건과 수정할 파일과 검증 방법을 구체적으로 준다. 비밀이나 학생 개인정보를 프롬프트에 넣지 않는다. AI가 작성한 코드도 diff와 테스트와 권한 흐름을 사람이 확인한다. 사용 가능한 AI 기능과 요금과 정책은 계정별로 확인한다.

AI의 설명은 실행 결과가 아니다. 코드 리뷰가 승인됐어도 테스트와 보안 요구를 생략하지 않는다.

### 브라우저 개발 환경의 저장과 실행과 중지

#### 편집할 수 있음과 실행할 수 있음은 다르다

github.dev는 브라우저에서 파일을 읽고 편집하는 경로다. Codespaces는 별도의 실행 환경을 제공한다. 파일 창이 비슷해 보여도 터미널과 설치와 실행과 청구가 같은 방식으로 동작한다고 생각하지 않는다.

질문	확인할 위치와 의미
편집 내용이 저장됐는가	현재 편집기의 파일 상태
Git에 기록됐는가	Source Control과 커밋
원격에 전달됐는가	Push와 웹 브랜치
실행 머신이 멈췄는가	Codespaces의 실제 상태

### 탭을 닫기 전에

필요한 변경을 기록하고 원격 전달 여부를 확인한다. 실행 환경을 더 쓰지 않는다면 중지 상태와 보관 중인 환경을 확인한다. 머신 실행 비용과 저장 비용을 구분하고 계정의 최신 사용량과 정책을 읽는다.

## AI에게 시킨 변경을 받을 때

요구한 파일 밖의 변경이 있는지 본다. 새 외부 서비스나 의존성이 들어왔는지 확인한다. 실행했다는 설명과 실제 로그를 구분한다. 코드가 익숙하지 않으면 파일별 목적과 데이터 흐름을 먼저 설명하게 한 뒤 작은 변경부터 검토한다.

### 좋은 추가 요청의 예

정답 버튼이 어떻게 동작하는지 입력부터 화면 변경까지 설명해 달라고 요청할 수 있다. 어떤 검사를 직접 실행했고 어떤 조건은 확인하지 못했는지도 묻는다. 막연히 완벽하게 고쳐 달라는 말보다 검토할 근거를 얻기 쉽다.

AI가 만든 코드와 사람이 만든 코드에 서로 다른 권한 기준을 적용하지 않는다. 비밀과 개인정보와 운영 영향은 같은 기준으로 검토한다.

## 실습 11.1 | AI 작업 요청문 작성

아래 요청문을 자신의 연습 프로젝트에 맞게 바꾼다. 이번 실습은 AI 서비스 가입이나 유료 실행을 요구하지 않는다.

### 목표

index.html의 정답 버튼을 키보드로 사용할 수 있게 해라.

### 범위

정답 열기와 닫기 동작과 접근성 이름을 확인해라.

데이터 수집 기능은 추가하지 마라.

### 완료 조건

Tab으로 버튼에 도달한다.

Enter와 Space로 동작한다.

360px 화면에서 내용이 잘리지 않는다.

### 제출 내용

변경 이유와 diff와 실제 검증 결과를 제시해라.

실행하지 못한 검사는 그렇게 표시해라.

## 생성 결과 검토

요청 밖의 파일 변경이 있는지 확인한다. 새 라이브러리와 외부 통신과 권한 변경을 따로 읽는다. 비밀번호와 토큰이 코드에 들어가지 않았는지 본다. 변경이 이해되지 않으면 파일별 설명을 요청하고 작은 PR로 나눈다.

확인 문제 AI가 정상 작동한다고 말하면 바로 배포해도 될까?

실제 환경과 입력으로 확인해야 한다. 설명과 실행 증거를 구분하고 실패 조건도 점검한다.

## 11.2 Container registry에 실행 환경 배포

관련 근거 [R81] [R43]

### 소스 판본과 실행 이미지 판본을 연결한다

GitHub Container registry인 ghcr.io에는 OCI 컨테이너 이미지를 배포할 수 있다. 처음 만든 패키지의 공개 범위와 연결 저장소의 접근 권한을 확인한다. 저장소가 Public이라고 패키지도 의도한 공개 범위라고 가정하지 않는다.

```
Dockerfile
FROM python:3.13-slim
WORKDIR /app
COPY index.html /app/index.html
USER 10001
EXPOSE 8080
CMD ["python", "-m", "http.server", "8080", "--bind", "0.0.0.0"]
```

이 이미지는 연습용 정적 서버다. 실제 인터넷 운영의 인증과 TLS와 운영 정책을 완성한 서버는 아니다. COPY 대상을 공개할 파일 하나로 좁혀 소스 전체와 .env를 무심코 넣지 않는다.

### 로컬 확인

Docker Engine 또는 호환 환경이 필요하다. 설치가 없다면 구조를 읽는 선택 실습으로 진행한다.

```
docker build -t lesson-page:local .
docker run --rm -p 127.0.0.1:8080:8080 lesson-page:local
```

브라우저의 localhost:8080에서 확인한 뒤 실행 터미널에서 Ctrl+C로 종료한다. 학습용 이미지 태그와 운영에서 사용하는 변경 불가능한 digest를 구분한다.

### 컨테이너 빌드와 패키지 공개의 경계

#### Dockerfile과 Workflow가 각각 정하는 것

Dockerfile은 이미지 안에 무엇을 넣고 어떻게 실행할지 정한다. Workflow는 어떤 커밋을 언제 빌드하고 어느 레지스트리에 보낼지 정한다. 이미지 빌드 성공과 패키지 Push 성공과 서비스 배포 성공을 따로 확인한다.

대상	대표 확인 사항
빌드 컨텍스트	불필요한 파일이 포함되지 않는가

대상	대표 확인 사항
.dockerignore	로컬 비밀과 큰 파일을 제외했는가
이미지 태그	어느 소스 판본인지 추적 가능한가
패키지 권한	필요한 읽기와 쓰기만 있는가
실행 환경	포트와 설정과 비밀 주입이 맞는가

## 태그와 이미지 내용을 연결

같은 태그 이름이 나중에 다른 이미지를 가리킬 수 있다. 조사와 재현에는 소스 커밋과 이미지 Digest를 함께 기록한다. latest라는 이름만으로 언제 만든 무엇인지 설명하기 어렵다.

## 비밀을 이미지에 굶지 않는다

파일을 복사하거나 빌드 인수에 값을 넣으면 이미지나 기록에 남을 수 있다. 필요한 비밀은 용도에 맞는 빌드 비밀 전달 방식이나 실행 환경의 비밀 주입을 사용한다. 마지막 단계에서 파일을 지웠다는 이유만으로 앞선 레이어에 남은 값까지 사라졌다고 판단하지 않는다.

## Push가 거절될 때

패키지 이름과 소유 주체와 Workflow 토큰 권한을 확인한다. 기존 패키지의 연결과 접근 설정도 살핀다. 오류를 해결하려고 개인 토큰을 곧바로 넓은 권한으로 발급하지 않는다. 어느 주체에게 어느 작업이 허용되지 않았는지부터 읽는다.

이 실습의 패키지 공개와 실제 서비스 운영은 다른 단계다. 실행 검증과 배포 환경과 복구 기준은 사용하는 서비스에 맞게 추가한다.

## 실습 11.2 | main 변경을 GHCR에 발행

아래 Workflow는 자신의 연습 저장소에 패키지를 쓰는 실제 자동화다. 활성화하기 전에 비용과 패키지 공개 범위를 확인한다. 외부 PR 이벤트에는 실행하지 않는다.

```
.github/workflows/container.yml
name: Publish container
on:
 push:
 branches: [main]
permissions:
```

```

 contents: read
 packages: write
jobs:
 publish:
 runs-on: ubuntu-latest
 steps:
 - uses: actions/checkout@v7
 - name: Build and publish
 env:
 GH_TOKEN: ${github.token}
 run: |
 image="ghcr.io/${GITHUB_REPOSITORY,,}"
 printf '%s' "$GH_TOKEN" | docker login ghcr.io \
 -u "$GITHUB_ACTOR" --password-stdin
 docker build -t "$image:$GITHUB_SHA" .
 docker push "$image:$GITHUB_SHA"

```

패키지 화면에서 커밋 SHA 태그와 접근 설정을 확인한다. 공개 Registry에서 받은 기본 이미지의 출처와 갱신도 관리한다. 운영에서는 FROM의 digest 고정과 취약점 검사와 출처 검증을 추가로 설계한다.

확인 문제 이미지 태그가 커밋 SHA 문자열이면 태그가 절대 이동할 수 없을까?  
이름을 그렇게 붙여도 Registry의 태그 정책과 쓰기 권한에 따라 덮어쓸 수 있다. 정확한 산출물 지정에는 이미지 digest와 검증 근거를 사용한다.

## 12장 자동 검사와 재사용 Workflow

첫 검사를 구성하고 실행 로그에서 실패 지점을 찾는다. Matrix와 결과물 보관을 익힌 뒤 공통 검사를 재사용 Workflow로 분리한다.

### 자동 실행과 지속적 통합은 같은 말이 아니다

자동화는 반복 작업을 도구가 수행하는 것이다. 지속적 통합은 변경을 공유 기준에 자주 통합하고 통합 결과를 검사하는 개발 관행이다. 매주 한 번 실행되는 긴 워크플로도 자동화지만 짧은 피드백을 제공하는 통합 과정과는 다를 수 있다. CI의 가치는 YAML 파일의 존재보다 실패를 발견하는 시점과 대응 방식에 있다.

Vasilescu 등의 연구는 GitHub에서 기여 방식과 프로젝트 특성이 자동 빌드 성공과 어떤 관련을 갖는지 분석했다. 인용한 자료는 수정 사항을 반영한 2015년 판본이다. 관찰된 관련성을 CI 도입의 보편적인 인과 효과로 확대하지 않고 개발 맥락을 고려해야 한다는 근거로 읽는다. [S15]

lesson-lab의 CI는 먼저 설치와 정적 검사 및 핵심 테스트를 실행할 수 있다. 검사마다 무엇을 확인하는지 문서화한다. 린터는 일부 코드 규칙을 확인하지만 업무 요구의 정확성을 증명하지 않는다. 테스트도 작성한 사례의 범위를 넘어서 모든 입력에 대한 보증을 제공하지 않는다.

### 실행 그래프와 실패 의미를 설계한다

워크플로의 job은 의존 관계를 통해 실행 그래프를 이룬다. 독립적인 검사는 병렬로 실행할 수 있지만 배포는 필요한 검사 결과를 기다려야 한다. 순서를 무조건 직렬로 만들면 피드백이 느려지고 필요한 의존성을 빼면 검증되지 않은 결과가 배포될 수 있다.

Matrix는 여러 런타임과 운영체제 조합을 검사하는 도구다. 조합이 늘수록 비용과 대기 시간이 증가한다. 실제 지원 범위와 실패 가능성이 높은 경계를 기준으로 조합을 선택한다. 모든 가능한 조합을 실행하는 것과 필요한 위험을 충분히 확인하는 것은 다른 목표다.

캐시는 다시 만들 수 있는 자료의 재사용에 적합하다. 아티팩트는 특정 실행의 결과를 전달하고 보관하는 데 사용한다. 캐시를 배포 결과의 유일한 근거로 삼으면 오래된 자료와 키 충돌 때문에 재현성이 약해질 수 있다. 무엇을 재사용하고 무엇을 판본별로 보관할지 구분한다.

## 재사용은 계약을 만드는 일이다

재사용 워크플로는 입력과 출력 및 필요한 권한을 공개 인터페이스처럼 다룬다. 호출하는 저장소가 어떤 파일과 비밀을 제공해야 하는지 명확히 한다. 공통 워크플로를 바꾸면 여러 프로젝트가 영향을 받을 수 있으므로 변경 이력과 호환성 정책이 필요하다.

워크플로 실패를 조사할 때는 마지막 빨간 줄만 보지 않는다. 최초의 의미 있는 실패와 그 결과로 생긴 후속 실패를 구분한다. 의존성 다운로드 실패 뒤 테스트 파일이 없다는 오류가 났다면 테스트 자체가 원인이 아닐 수 있다. 같은 입력으로 다시 실행한 결과와 일시적인 외부 실패 여부도 함께 기록한다.

## 캐시는 속도를 위한 입력이며 검증 결과가 아니다

캐시는 이전 실행의 일부 데이터를 재사용해 비용을 줄인다. 캐시 키가 같다는 이유로 현재 코드의 정확성이 확인되는 것은 아니다. 운영체제와 런타임 및 의존성 잠금 파일처럼 결과에 영향을 주는 요소를 키에 반영하고 캐시가 없는 실행에서도 빌드가 성립하는지 확인한다. 부분 일치 복원이 사용될 때는 정확히 같은 입력이 복원된다고 가정하지 않는다. [T26]

캐시와 배포 산출물은 목적이 다르다. 캐시는 다시 계산할 수 있는 중간 자료이고 산출물은 특정 실행의 결과로 식별해야 하는 파일이다. 배포할 파일을 넓은 복원 조건의 캐시에서 가져오면 어느 코드에서 만들어졌는지 모호해질 수 있다. 검사한 산출물과 배포한 산출물을 연결하는 흐름이 필요하다.

신뢰 수준이 낮은 실행이 만든 데이터를 높은 권한의 실행에서 사용하면 캐시도 신뢰 경계를 넘는다. 플랫폼의 현재 접근 제어와 별개로 어떤 실행이 쓰고 어떤 실행이 읽는지 확인한다. 캐시에 비밀을 저장하지 않으며 복원한 실행 파일을 무조건 신뢰하지 않는다. 이 원칙은 패키지 캐시와 생성 코드 및 테스트 도구에도 적용된다.

## 실패를 통과로 바꾸는 설정은 의미를 바꾼다

실험적인 검사를 계속 진행하도록 설정할 수는 있지만 그 결과를 필수 검사 통과와 같은 의미로 사용하면 안 된다. 작업을 계속 실행하는 것과 변경을 승인하는 것은 다른 판단이다. 최종 판정은 어떤 작업이 필수이며 실패와 취소 및 건너뛴을 어떻게 처리하는지 명시해야 한다.

테스트가 불안정하게 실패하면 성공할 때까지 재실행하는 관행이 생기기 쉽다. 그러나 반복 실행은 오류의 원인을 숨길 수 있다. 입력과 시간 및 외부 서비스 의존성을 기록하고 재현 조건을 찾는다. 재시도를 허용한다면 횟수와 이유를 남겨 최초 실패율과 최종 성공률을 구분한다. 지표의 정의가 바뀌면 품질 판단도 달라진다.

CI 실행 시간은 작업 시간의 단순 합보다 의존성 경로에 영향을 받는다. 서로 독립적인 검사는 병렬화할 수 있지만 배포가 필수 검사를 기다리는 관계는 유지해야 한다. 빠른 실행을 위해 필요한 의존성을 제거하면 검증 전에 공개되는 경로가 생길 수 있다. 성능 개선도 판정의 의미를 보존하며 수행한다.

상태 변화 / CI는 단계별 주장을 확인한다



설치 성공과 테스트 성공 및 배포 성공은 다른 주장이다. 실제 워크플로에서는 필요한 의존 관계와 권한을 명시한다.

1. 검사할 소스 판본을 고정한다
2. 의존성을 준비하고 검사를 실행한다
3. 실패하면 같은 결과를 운영으로 보내지 않는다
4. 수정한 새 판본을 다시 검사한다
5. 검사한 판본과 배포한 판본을 연결한다

# 12.1 CI의 원리와 첫 GitHub Actions

관련 근거 [R37] [R38] [R39]

## 변경이 들어올 때 자동으로 확인한다

CI는 변경을 합칠 때 반복 검사를 자동으로 수행하는 방식이다. GitHub Actions에서는 이벤트가 Workflow를 시작하고 Job이 실행 환경인 Runner에서 Step을 수행한다. Step은 셸 명령이나 재사용 가능한 Action을 호출한다.

요소	예	의미
Event	pull_request	언제 시작할까
Workflow	.github/workflows/ci.yml	전체 실행 정의
Job	validate	독립 실행 단위
Runner	ubuntu-latest	명령을 실행할 환경
Step	python scripts/check.py	순차 작업

## 첫 자동 검사는 작게 시작한다

수업 HTML의 파일 존재와 기본 표식을 확인하는 연습용 검사부터 만든다. 이후 HTML 검사와 접근성과 단위 테스트 등 실제 프로젝트에 맞는 도구로 확장한다. 초록색 표시의 의미는 내가 작성한 검사 범위만큼이다.

이 책의 Actions 예제는 2026-10-21 확인한 공식 저장소의 checkout@v7과 setup-python@v7 표기를 사용한다. 학습용 태그 예제이며 운영에서는 공식 출처의 검토한 전체 커밋 SHA로 고정한다. 자체 Runner는 요구 버전도 확인한다.

## 첫 Workflow를 한 줄씩 읽기

### YAML의 들여쓰기는 포함 관계다

name은 사람이 볼 이름이다. on은 실행 사건을 정한다. jobs 아래의 validate는 이 예제의 Job 식별자다. steps 아래 목록은 그 Job에서 실행할 순서다. 탭과 공백을 임의로 섞지 말고 일관된 들여쓰기를 유지한다.

줄 또는 구역	이 예제에서 필요한 이유
pull_request	변경 제안에서 검사
push의 main	병합 후 기본 브랜치도 검사
contents: read	소스를 읽을 최소 권한
runs-on	검사할 실행 환경
checkout	저장소 파일을 Runner로 준비
setup-python	사용할 Python 버전 선택
run	실제 검사 스크립트 실행

## Runner에 파일이 있다고 가정하지 않는다

내 PC에서 파일이 보인다는 사실과 새 Runner에 파일이 준비됐다는 사실은 다르다. checkout Step 뒤의 작업 폴더를 기준으로 스크립트 경로를 계산한다. 실패 메시지에 파일이 없다고 나오면 경로와 대소문자와 checkout을 먼저 본다.

## AssertionError를 읽는 법

12.1절의 검사는 문자열을 확인하다 조건이 거짓이면 실패한다. 로그의 마지막 실패 줄만 읽지 말고 어느 assert에서 멈췄는지 본다. title 태그를 고쳤다면 HTML 자체가 틀린 것인지 검사 방식이 너무 좁은 것인지 구분한다.

## 이 예제에서 증명한 것

검사 스크립트가 지정한 문장이 파일에 있는지는 확인했다. 키보드 조작과 실제 브라우저 표시와 서버 보안까지 검증한 것은 아니다. 프로젝트가 커지면 의도에 맞는 파서와 기능 테스트를 추가한다.

초록색을 목표로 검사 코드를 지우지 않는다. 실패 원인을 이해하고 요구에 맞는 검사를 유지하는 것이 목표다.

## 실습 12.1 | HTML 기본 검사 자동화

scripts/check.py와 .github/workflows/ci.yml 두 파일을 만든다. 먼저 로컬에서 python scripts/check.py가 성공하는지 확인한다. Python 3 설치가 필요하다.

```
scripts/check.py
from pathlib import Path

page = Path('index.html').read_text(encoding='utf-8')
assert '<title>' in page
assert "lang='ko'" in page
assert "id='answer'" in page
print('HTML baseline checks passed')

.github/workflows/ci.yml
name: Lesson CI
on:
 pull_request:
 push:
 branches: [main]
permissions:
 contents: read
jobs:
 validate:
 runs-on: ubuntu-latest
 timeout-minutes: 5
 steps:
 - uses: actions/checkout@v7
 - uses: actions/setup-python@v7
 with:
 python-version: '3.13'
 - run: python scripts/check.py
```

확인 문제 위 검사가 성공하면 HTML 문법과 접근성도 모두 검증됐을까?  
아니다. 단순 문자열 확인만 수행한다. 따옴표 형식을 바꿔도 실패할 수 있다. 학습용 예제의 한계이며 실제 프로젝트에는 목적에 맞는 파서와 테스트를 도입한다.

## 12.2 Actions 실패 진단과 Matrix와 재사용

관련 근거 [\[R37\]](#) [\[R40\]](#) [\[R41\]](#)

### 실패한 Job과 Step을 좁힌다

전체 로그를 처음부터 읽기보다 실패한 Step을 찾고 그 위의 실제 오류를 읽는다. 파일 경로와 실행 폴더와 의존성 설치와 버전과 권한을 차례로 확인한다. 성공한 로컬 환경과 CI 환경이 같은지도 본다.

오류	먼저 볼 것
파일 없음	checkout 여부와 경로와 대소문자
명령 없음	설치 Step과 PATH
권한 거절	permissions와 토큰 범위
검사 대기 지속	트리거와 필수 검사 이름
간헐 실패	시간과 네트워크와 테스트 격리

## Matrix와 의존 관계

Matrix는 여러 버전이나 환경 조합으로 Job을 펼친다. needs는 다른 Job의 성공 이후 실행하도록 연결한다. workflow_call은 여러 저장소나 Workflow가 재사용하는 진입점을 만든다. 반복이 실제로 생겼을 때 추출하면 입력과 권한 설정이 명확해진다.

## Cache와 Artifact를 구분한다

Cache는 의존성을 다시 받는 시간을 줄이는 데 쓴다. Artifact는 실행 결과와 보고서와 빌드 산출물을 보관하고 전달한다. 비밀은 어느 쪽에도 넣지 않는다. 캐시가 복구됐다고 빌드 결과가 검증된 것은 아니다.

여러 OS와 버전을 모두 돌리면 실행량이 늘어난다. 실제 지원하는 환경부터 골라 비용과 신뢰도를 함께 관리한다.

## Workflow 로그에서 실패 원인을 좁히기

### 실행과 Job과 Step을 순서대로 찾는다

Actions 목록에서 해당 커밋의 실행을 고른다. 실패한 Job을 연다. 그 안에서 실패한 Step을 찾아 실제 오류와 앞선 설치 결과를 읽는다. PR이 여러 번 갱신했다면 오래된 실행을 보고 있지 않은지도 확인한다.

로그의 종류	질문
checkout 실패	저장소 읽기 권한과 네트워크가 맞는가
설치 실패	버전과 잠금 파일과 설치 경로가 맞는가

로그의 종류	질문
검사 실패	어떤 입력과 기대 결과가 달랐는가
업로드 실패	실제 결과 파일이 만들어졌는가
권한 거절	이 Job에 허용한 범위가 충분한가

## Matrix에서 하나만 실패하면

지원 환경 사이의 차이를 찾는다. 예를 들어 한 Python 버전에서만 실패하면 사용 문법과 의존성 호환과 실행 환경을 비교한다. 모든 조합이 초록색이 되도록 실패한 환경을 조용히 빼기보다 실제 지원 범위를 결정한다.

## Cache가 문제를 감출 수도 있다

캐시가 있으면 설치가 빨라질 수 있지만 재현성이 보장되는 것은 아니다. 캐시 키와 의존성 잠금 기준이 맞는지 확인한다. 깨끗한 설치에서도 같은 결과가 나오는지 필요한 범위에서 검증한다. 캐시 파일에 비밀을 넣지 않는다.

## Artifact가 없는 경우

검사나 빌드가 앞에서 실패해 결과물을 만들지 못했을 수 있다. 결과 경로가 바뀌었는지도 확인한다. 실패 보고서를 보관하려면 실패 시 실행 조건과 파일 생성 여부를 설계해야 한다. 업로드 성공만으로 결과 내용이 맞는 것은 아니다.

재실행은 원인을 좁히는 한 방법이다. 반복할 때마다 결과가 달라진다면 간헐 실패를 별도 문제로 기록한다.

## 실습 12.2 | 두 Python 버전으로 검사

12.1절 Workflow의 validate Job을 아래 형태로 바꾼다. 전체 파일에는 기존 name과 on과 permissions를 유지한다.

```
jobs:
 validate:
 strategy:
 fail-fast: false
 matrix:
 python: ['3.12', '3.13']
 runs-on: ubuntu-latest
 timeout-minutes: 5
 steps:
 - uses: actions/checkout@v7
```

```
- uses: actions/setup-python@v7
 with:
 python-version: ${ matrix.python }
- run: python scripts/check.py
```

## 실패 실험

연습 브랜치에서 index.html의 title 태그를 잠시 제거한다. CI가 실패한 Step과 AssertionError를 확인한다. 파일을 복원하고 새 커밋으로 다시 성공하는지 본다. 실패 검사를 삭제해서 초록색으로 만들지 않는다.

## 검사 이름을 바꿀 때

Rulesets의 필수 검사가 옛 이름을 기다릴 수 있다. Workflow 변경과 필수 검사 설정을 함께 점검한다. Matrix 검사 이름이 달라지는 점도 확인한다. 실제 병합 판단에는 전체 검사를 모으는 안정된 최종 Job을 둘 수도 있다.

확인 문제 실패한 실행을 다시 돌려 성공했다면 원인이 해결된 것일까?  
일시적 현상일 수 있다. 간헐 실패가 반복되면 원인을 기록하고 재현 조건을 좁혀야 한다.  
재실행만으로 덮으면 신뢰도가 떨어진다.

## 12.3 재사용 가능한 CI를 실제로 분리

관련 근거 [R78] [R37]

### 검사 로직과 호출 시점을 나눈다

동일한 검사를 여러 Workflow가 호출한다면 공통 부분을 workflow_call로 분리한다. 재사용 Workflow는 Step이 아니라 Job의 uses에서 호출한다. 같은 저장소의 상대 경로 호출은 호출자와 같은 커밋의 정의를 사용한다.

```
.github/workflows/reusable-check.yml
name: Reusable lesson check
on:
 workflow_call:
permissions:
 contents: read
jobs:
 check:
 runs-on: ubuntu-latest
 timeout-minutes: 5
 steps:
 - uses: actions/checkout@v7
 - uses: actions/setup-python@v7
 with:
```

```
python-version: '3.13'
- run: python scripts/check.py
```

## 비밀을 무심코 상속하지 않는다

이 검사는 비밀이 필요하지 않으므로 secrets를 넘기지 않는다. 외부 서비스가 필요한 재사용 Workflow는 이름을 정한 비밀만 전달한다. secrets: inherit는 편리하지만 상속 범위가 넓어질 수 있다. 호출 단계가 내려가면서 토큰 권한을 임의로 높일 수 있다고 생각하지 않는다.

사용한 Action 태그는 학습용이다. 운영에서는 검토한 공식 커밋 SHA와 Runner 호환 조건을 확인한다.

## 재사용 Workflow의 호출 계약을 점검

### 파일 두 개는 서로 다른 책임을 가진다

호출하는 Workflow는 언제 실행할지와 무엇을 넘길지 정한다. 호출되는 Workflow는 허용하는 입력과 비밀과 작업을 정의한다. 둘을 함께 읽어야 값의 출발점과 사용 지점을 찾을 수 있다.

위치	확인할 질문
on.workflow_call	외부 호출을 받도록 선언했는가
inputs	이름과 자료형과 기본값이 맞는가
호출 job의 uses	올바른 파일과 참조를 가리키는가
with	선언된 입력을 맞게 넘기는가
secrets	필요한 비밀만 전달하는가
permissions	각 단계의 권한이 필요한 범위인가

### 실행 전에 손으로 연결해 본다

입력 하나를 골라 호출 파일의 값부터 실제 명령에서 사용하는 위치까지 따라간다. 입력 이름의 오타와 Boolean을 문자열처럼 다루는 실수를 찾는다. 외부에서 온 값을 셸 코드 안에 직접 끼워 넣는지 확인하고 데이터로 전달할 방법을 사용한다.

## 호출 실패와 작업 실패를 나눈다

파일 경로나 참조나 입력 계약이 잘못되면 작업을 시작하기 전에 실패할 수 있다. 작업이 시작된 뒤 테스트가 실패했다면 로그의 실제 명령과 환경을 읽는다. 실패 위치를 나누면 YAML 수정과 프로그램 수정을 혼동하지 않는다.

## 공용 Workflow를 바꿀 때

어떤 저장소가 어떤 참조로 호출하는지 파악한다. 입력 제거와 기본값 변경은 기존 호출에 영향을 줄 수 있다. 작은 호출 저장소에서 검증한 뒤 판본과 변경 안내를 남긴다. 외부 저장소의 Workflow는 고정된 참조의 변경 정책과 공급망 신뢰도 함께 검토한다.

재사용은 반복을 줄이는 수단이다. 너무 많은 선택지를 한 파일에 모으면 어떤 조합이 검증됐는지 알기 어려워질 수 있다.

## 실습 12.3 | 호출 Workflow 만들기

아래 파일은 기존 12.1절 ci.yml을 대체할 수 있다. 같은 이벤트로 기존 CI와 새 CI를 함께 남기면 중복 실행되므로 실제 목적에 맞게 하나를 정리한다.

```
.github/workflows/ci.yml
name: Lesson CI
on:
 pull_request:
 push:
 branches: [main]
permissions:
 contents: read
jobs:
 validate:
 uses: ../github/workflows/reusable-check.yml
```

1. 재사용 파일과 호출 파일을 같은 PR에 넣는다.
2. scripts/check.py와 index.html이 있는지 확인한다.

3. PR의 Actions 로그에서 호출 관계를 읽는다.
4. title 태그를 제거한 시험 변경으로 실패도 확인한다.
5. 복원한 뒤 필수 검사 이름이 맞는지 다시 본다.

### 공통화할 때 놓치기 쉬운 것

재사용 경로와 호출 시점과 권한과 비밀과 실패 로그를 읽는 위치를 문서화한다. 모든 프로젝트를 하나의 거대한 Workflow에 맞추기보다 입력 계약이 안정적인 반복 작업부터 나눈다.

확인 문제 재사용 Workflow를 Step의 uses에 넣으면 같은 뜻일까?  
아니다. 재사용 Workflow는 Job에서 호출한다. 여러 Step을 묶는 Composite Action과 호출 위치와 기능이 다르다.

## 13장 비밀과 배포 권한과 사고 대응

외부 입력과 실행 권한의 경계를 검토한다. OIDC와 배포 환경을 통제하고 비밀 유출 시 차단과 조사와 복구를 순서대로 수행한다.

### 실행할 코드의 출처와 사용할 권한을 함께 본다

자동화의 위험은 코드가 실행되는 위치와 사용할 수 있는 권한이 결합될 때 생긴다. 외부 PR의 코드를 검사하는 작업과 운영 비밀을 사용하는 배포 작업은 신뢰 수준이 다르다. 두 작업을 한 job에 섞으면 검증되지 않은 코드가 민감한 자원에 접근할 수 있다.

GitHub 공식 문서는 `pull_request_target`의 워크플로가 높은 신뢰 맥락에서 실행된다는 점과 외부 PR 코드를 가져와 실행하는 패턴의 위험을 설명한다. 현재 문서에는 공개 저장소에 대한 기본 정책의 평가 모드와 향후 적용 일정도 안내되어 있다. 정책의 현재 상태를 확인하되 설계 자체에서 신뢰되지 않은 코드와 비밀을 분리한다. [T06]

PR 제목과 이슈 본문도 신뢰되지 않은 입력이다. 이를 셸 스크립트에 직접 삽입하면 문장이 명령으로 해석될 수 있다. 환경 변수나 안전한 인수 전달 방식을 사용하고 해당 셸의 인용 규칙을 지킨다. 값이 GitHub에서 왔다는 사실과 프로젝트 관리자가 작성했다는 사실은 다르다.

### 짧은 자격 증명도 조건이 넓으면 위험하다

OIDC를 이용하면 장기 비밀을 저장하는 대신 실행 맥락을 바탕으로 짧은 자격 증명을 받을 수 있다. 그러나 신뢰 정책이 모든 저장소와 브랜치를 허용하면 짧은 유효 기간만으로 충분한 통제가 되지 않는다. 발급 주체와 대상 및 저장소와 참조 또는 환경 조건을 적절히 제한해야 한다.

Environment의 승인과 배포 제한은 운영 자원에 접근할 시점을 통제한다. 승인자가 어떤 커밋과 검사 결과를 승인하는지 알 수 있어야 의미가 있다. 사람이 버튼을 눌렀다는 사실만 남고 검토 대상이 불분명하면 통제의 실질적인 가치가 낮아진다.

### 비밀 유출의 우선순위는 자격 증명의 무효화다

비밀이 커밋되었다면 먼저 노출된 자격 증명을 폐기하거나 교체한다. 파일을 삭제하거나 이력을 재작성해도 이미 복제된 값까지 회수할 수는 없다. 이후 영향 범위를 조사하고 필요한 이력 정리와 캐시 및 아티팩트 확인을 수행한다. 처리 순서는 비밀의 악용 가능성을 줄이는 목적에 맞춰 정한다.

사고 기록에는 발견 시점과 노출 범위 및 수행한 조치와 확인 결과를 남긴다. 개인을 비난하는 문서보다 재발 조건을 설명하는 문서가 유용하다. 어떤 권한과 검토 경계가 실패했는지 알아야 도구 설정과 작업 절차를 개선할 수 있다.

## 토큰을 받을 수 있다는 사실과 배포할 수 있다는 사실은 다르다

OIDC에서 GitHub Actions는 실행의 주체와 문맥을 주장하는 토큰을 제공한다. 클라우드 제공자는 발급자와 대상 및 주체 등의 조건을 확인해 자신의 권한으로 교환할지를 판단한다. id-token 권한은 토큰 요청을 허용하는 조건이며 그 자체가 모든 클라우드 자원에 대한 권한을 뜻하지 않는다. 양쪽의 정책이 연결되어야 한다. [T25]

신뢰 조건을 저장소 이름 하나로만 넓게 정하면 의도하지 않은 브랜치나 실행 문맥에서도 권한을 받을 수 있다. 배포 대상과 브랜치 및 환경을 어떻게 제한할지 검토한다. 짧은 수명의 토큰도 허용 범위가 넓으면 그 시간 동안 큰 영향을 줄 수 있다. 만료 시간과 권한 범위는 각각의 통제 수단이다.

2026년 10월 확인한 문서에는 2026년 7월 15일 이후 생성된 저장소와 이후의 이름 변경 및 이전에 대해 소유자와 저장소의 불변 ID를 포함하는 주체 형식이 설명되어 있다. 이전 저장소에는 기존 형식이 남을 수 있으며 GitHub Enterprise Server에는 같은 전제를 적용하지 않는다. 예제 문자열을 그대로 복사하기보다 대상 저장소의 실제 주체 형식을 확인해 클라우드 정책과 맞춘다. [T25]

## 인증 실패를 해결할 때 조건을 하나씩 대조한다

배포 인증이 실패하면 토큰 발급과 교환 및 자원 접근을 나누어 조사한다. 토큰을 받지 못한 문제와 클라우드가 주체 조건을 거부한 문제 및 교환 후 자원 권한이 부족한 문제는 조치가 다르다. 모든 권한을 넓히는 방식은 원인을 확인하기 어렵게 만들고 불필요한 접근을 허용할 수 있다.

로그에는 판정에 필요한 비밀이 아닌 식별 정보와 단계별 오류를 남긴다. 토큰 원문을 복사해 공유할 필요는 없다. 저장소 이전 직후 실패했다면 변경된 주체와 기존 신뢰 조건을 대조한다. 원인을 설명할 수 있는 최소 변경을 적용하고 필요한 작업만 허용되는지 다시 확인한다.

상태 변화 / 외부 PR 코드와 배포 권한을 분리한다

외부 PR 코드와 배포 권한을 분리한다

1 / 4 외부 기여 코드는 신뢰되지 않은 입력이다

입력 출처

외부 PR

검사 환경

격리 검사

비밀 접근

없음

허용 행동

테스트

단계 사이의 자료도 검증해야 한다. 외부 실행이 만든 파일을 내려받아 높은 권한으로 실행하면 경계가 무너질 수 있다.

외부 PR 코드와 배포 권한을 분리한다

4 / 4 운영 비밀은 필요한 단계에만 전달한다

입력 출처

운영 승인

검사 환경

배포 환경

비밀 접근

필요 범위

허용 행동

게시

단계 사이의 자료도 검증해야 한다. 외부 실행이 만든 파일을 내려받아 높은 권한으로 실행하면 경계가 무너질 수 있다.

단계 사이의 자료도 검증해야 한다. 외부 실행이 만든 파일을 내려받아 높은 권한으로 실행하면 경계가 무너질 수 있다.

- 1. 외부 기여 코드는 신뢰되지 않은 입력이다
- 2. 검사 결과가 권한 상승을 자동 허용하지 않는다
- 3. 신뢰된 소스와 빌드 조건을 확인한다
- 4. 운영 비밀은 필요한 단계에만 전달한다

13.1 Secrets와 GITHUB_TOKEN과 안전한 실행

관련 근거 [R42] [R43]

133

## 자동화는 코드 실행 권한이다

GITHUB_TOKEN은 Workflow 실행에 제공되는 토큰이며 권한을 최소화해야 한다. contents: read만 필요한 검사에 write-all을 주지 않는다. permissions에서 일부 권한을 지정하면 지정하지 않은 권한은 기본적으로 none이 되므로 필요한 Job에서만 추가한다.

항목	원칙
저장소 읽기	contents: read
배포 쓰기	필요한 Job에만 별도 권한
외부 서비스 비밀	Secrets로 주입
외부 PR	비신뢰 코드로 취급
Action 참조	공식 원본의 전체 SHA 검토 후 고정

## 외부 입력을 셸 코드에 직접 붙이지 않는다

PR 제목 같은 입력을 run 문자열 안에 바로 삽입하면 셸 명령으로 해석될 수 있다. 환경 변수로 전달하고 인용해 데이터로 다룬다. 마스킹은 로그에서 비밀의 모든 변형을 완벽히 숨긴다는 뜻이 아니다.

```
- name: Show title safely
 env:
 PR_TITLE: ${GITHUB_EVENT_PULL_REQUEST_TITLE}
 run: printf '%s\n' "$PR_TITLE"
```

pull_request_target은 일반 pull_request와 신뢰 경계가 다르다. 외부 PR의 코드를 체크아웃해 권한 있는 환경에서 실행하는 설계를 피한다. 플랫폼 보호 기능만 믿고 임의 코드 실행을 허용하지 않는다.

## 비밀과 외부 입력이 만나는 지점 찾기

### 같은 문자열도 들어온 경로에 따라 신뢰가 다르다

저장소 관리자가 검토해 넣은 실행 명령과 외부 작성자가 적은 PR 제목은 같은 수준의 입력이 아니다. 제목을 셸 코드에 직접 끼워 넣으면 단순 문장이 명령의 일부로 해석될 수 있다. 입력은 데이터로 전달하고 적절히 인용한다.

검토 질문	찾아볼 지점
누가 입력하는가	제목과 본문과 브랜치 이름
어디서 실행하는가	run과 스크립트 인자
무엇을 읽을 수 있는가	Secrets와 토큰과 내부 파일
무엇을 바꿀 수 있는가	저장소와 배포와 패키지 권한
무엇이 남는가	로그와 캐시와 Artifact

### 최소 권한은 Job별로 판단한다

단순 읽기 검사에 저장소 쓰기 권한을 줄 이유가 없다. 배포에 쓰기 권한이 필요하다면 그 Job에 좁혀 부여한다. 저장소 기본값에 의존하기보다 Workflow에서 의도를 드러내고 조직 정책과의 관계를 확인한다.

### 비밀이 없으면 안전한가

Runner가 내부망이나 별도 파일에 접근할 수 있다면 환경 자체가 중요한 권한일 수 있다. 특히 오래 살아 있는 자체 Runner에서는 앞선 작업의 흔적과 자격 증명과 네트워크 접근도 고려한다.

### 로그에 남기는 정보

문제 해결에 필요한 오류와 경로는 기록하되 토큰 값은 출력하지 않는다. 마스킹이 모든 변형을 가린다고 가정하지 않는다. 비밀을 Base64로 바꾸거나 잘라 출력하는 것도 노출 경로가 될 수 있다.

권한 높은 이벤트에서 비신뢰 PR의 코드를 실행하는 조합을 먼저 찾는다. 편의를 위해 만든 우회 Workflow가 원래 보호를 무너뜨리지 않는지 검토한다.

## 실습 13.1 | Workflow 권한 리뷰

실제 비밀 값을 만들 필요는 없다. 12.1절 Workflow와 조직에서 쓰는 Workflow 하나를 읽고 아래 표를 채운다.

질문	기록할 내용
누가 이벤트를 만들 수 있는가	내부 작성자 또는 외부 기여자

질문	기록할 내용
어떤 코드를 실행하는가	기준 브랜치 또는 PR 코드
어떤 토큰이 있는가	GITHUB_TOKEN과 외부 비밀
무엇을 쓸 수 있는가	코드와 배포와 패키지
실패하면 무엇이 남는가	로그와 Artifact와 캐시

## Action SHA 고정 절차

1. 사용할 Action의 공식 소유자와 저장소를 확인한다.
2. 원하는 버전의 태그와 릴리스 설명을 읽는다.
3. 공식 저장소에서 그 버전의 전체 커밋 ID를 확인한다.
4. uses의 태그를 검증한 SHA로 바꾸고 주석에 버전을 남긴다.
5. Dependabot 등으로 갱신 PR을 받고 변경 사항을 검토한다.

## 자체 Runner의 추가 위험

Job이 끝나도 머신의 파일과 자격 증명이 남을 수 있다. 외부 PR을 내부망에 접근하는 상시 Runner에서 실행하지 않는다. 격리와 수명과 접근 경로와 로그 정리를 설계한다.

확인 문제 Secret이 로그에서 마스킹되면 출력해도 괜찮을까?  
출력하지 않는 것이 원칙이다. 인코딩이나 분할이나 파생 값은 기대처럼 가려지지 않을 수 있다. 로그와 결과물에 비밀이 들어가지 않게 설계한다.

## 13.2 Environment와 OIDC와 배포 통제

관련 근거 [R44] [R45]

### 검증과 배포를 다른 권한으로 실행한다

검사는 읽기 권한으로 돌리고 운영 배포는 별도 Job에서 실행한다. Environment에 배포 대상과 필요한 승인과 비밀을 연결할 수 있다. 환경 보호 기능의 제공 범위는 저장소 공개 여부와 요금제를 확인한다.

OIDC는 GitHub의 실행 신원을 클라우드가 검증한 뒤 짧은 수명의 자격을 발급하는 방식이다. 저장소에 장기 클라우드 키를 두는 일을 줄인다. GitHub 권한만 켜면 끝나는 것이 아니라 클라우드의 신뢰 정책을 정확히 제한해야 한다.

조건	제한할 대상의 예
Repository	지정 조직의 지정 저장소
Branch 또는 Environment	main 또는 production
Audience	대상 클라우드가 기대하는 값
Cloud role	필요한 리소스의 최소 권한

### 동시에 두 버전을 배포하지 않는다

concurrency로 같은 환경의 실행을 제어한다. 검사에는 오래된 실행 취소가 유용할 수 있다. 운영 배포는 중간 취소의 영향을 고려해야 한다. 성공 판정은 명령 종료뿐 아니라 실제 서비스의 상태 확인까지 포함한다.

id-token: write는 OIDC ID 토큰을 요청할 권한이다. 그 자체가 저장소 코드 쓰기 권한이나 클라우드 관리자 권한을 뜻하지 않는다. 클라우드 정책이 허용할 권한을 따로 결정한다.

## OIDC에서 GitHub와 클라우드의 역할 나누기

### GitHub는 실행 신원을 제시하고 클라우드는 허용을 판단한다

Workflow의 id-token: write는 실행 신원을 담은 ID 토큰을 요청할 수 있게 한다. 실제 자원에 접근할 자격을 발급할지는 클라우드가 설정한 신뢰 정책과 역할 권한에 달려 있다. 토큰 요청과 자원 쓰기를 한 설정처럼 생각하지 않는다.

경계	확인할 내용
GitHub 이벤트	어떤 브랜치와 사건이 시작할 수 있는가
Environment	배포 대상과 승인과 비밀 범위
OIDC 신뢰	어느 저장소와 실행 조건을 신뢰하는가
클라우드 역할	어느 자원에 무엇을 할 수 있는가
배포 검증	어떤 산출물이 실제로 적용됐는가

## 신뢰 조건을 넓게 쓰면

조직 이름만 맞으면 모든 저장소의 모든 실행을 믿는 식으로 설계할 경우 원래 의도보다 넓은 경로가 열릴 수 있다. 공급자의 공식 안내에 맞춰 저장소와 브랜치 또는 Environment와 audience 등을 좁힌다. 값의 문법은 공급자마다 확인한다.

## Environment 승인 이후의 코드

승인을 받는 코드와 실제 실행하는 코드가 연결되어야 한다. 승인 이후 임의 브랜치의 최신 내용을 다시 가져오는 설계는 검토한 대상을 바꿀 수 있다. 어떤 커밋의 어떤 결과물을 쓰는지 기록한다.

## 배포를 취소하거나 되돌릴 때

코드 배포와 데이터 변경이 함께 있었는지 확인한다. 중간 취소가 어떤 자원을 남기는지도 본다. 같은 환경에 두 실행이 겹치지 않게 제어하되 오래된 배포를 취소할지 끝낼지는 서비스의 복구 특성에 맞춰 정한다.

장기 키가 없어졌다는 사실만으로 운영 위험이 모두 사라지지는 않는다. 신뢰 조건과 자원 권한과 배포 대상의 연결을 계속 검토한다.

## 설계 실습 13.2 | 운영 배포 Job 검토

아래는 구조 설명용 조각이다. 클라우드 로그인과 배포 명령은 서비스별 공식 설정이 필요하므로 이대로 실제 배포가 완성되는 예제가 아니다.

```
deploy:
 needs: validate
 if: github.ref == 'refs/heads/main'
 runs-on: ubuntu-latest
 environment: production
 concurrency:
 group: production-deploy
 cancel-in-progress: false
 permissions:
 contents: read
 id-token: write
 steps:
 - uses: actions/checkout@v7
 # 여기에 공급자 공식 OIDC 로그인 추가
 - 검증된 산출물 배포와 상태 확인 추가
```

## 운영자가 답해야 할 질문

어느 커밋의 어떤 산출물을 배포하는가. 승인은 누가 하는가. 환경 비밀은 언제 접근 가능한가. 실패하면 이전 버전으로 어떻게 돌리는가. 데이터 구조가 바뀌었다면 코드만 되돌려도 되는가. 각 질문의 답을 릴리스 운영 문서에 남긴다.

확인 문제 production이라는 환경 이름만 만들면 외부 PR의 배포가 자동으로 안전해질까? 아니다. 이벤트와 if 조건과 보호 규칙과 비밀 범위와 클라우드 신뢰 정책이 함께 맞아야 한다.

## 13.3 비밀 유출과 장애 대응

관련 근거 [R49] [R48]

### 파일 삭제보다 자격 증명 회수가 먼저다

API 키나 토큰을 올렸다면 먼저 해당 서비스에서 폐기하거나 교체한다. 새 키가 정상 동작하는지 확인하고 비밀을 안전한 경로로 다시 설정한다. GitHub 파일을 지우는 커밋과 Revert는 과거 이력의 비밀을 없애지 않는다.

순서	행동	목적
1	노출 키 폐기와 교체	추가 악용 차단
2	권한과 사용 기록 확인	영향 범위 파악
3	실행 로그와 배포 파일 점검	다른 노출 지점 확인
4	필요한 이력 정리 협의	잔존 내용 축소
5	재발 방지와 기록	같은 경로 차단

### 이력 정리의 영향

git-filter-repo 같은 도구로 이력을 재작성하면 커밋 ID와 서명과 참조와 동료 사본에 영향을 준다. Fork와 PR 참조와 캐시까지 한 번의 Push로 모두 회수할 수 있다고 생각하지 않는다. 공식 민감 정보 제거 절차를 읽고 필요한 경우 GitHub 지원 절차를 따른다.

유출 키를 사고 보고서에 다시 붙여 넣지 않는다. 키의 이름과 발생 시각과 관련 커밋을 기록하고 실제 값은 제외한다. 개인정보 사고는 조직의 대응 담당에게 연결한다.

## 유출 발견부터 복구 확인까지 시간순으로 정리

### 키가 유효한 시간을 먼저 줄인다

실제 키를 올렸다면 저장소 정리보다 해당 서비스의 키 폐기나 교체를 먼저 진행한다. 새 키로 필요한 서비스가 작동하는지 확인하고 옛 키가 더 이상 허용되지 않는지 검토한다. 이 과정에서 새 키를 다시 파일에 적어 올리지는 않는다.

시점	남길 기록
발견	발견 시각과 관련 저장소와 커밋
차단	회수한 키의 이름과 완료 시각
영향 조사	사용 기록과 접근 범위와 비용
복구	교체 후 확인한 기능과 결과
정리	이력과 로그와 결과물의 후속 조치

### 파일을 지웠는데 왜 끝나지 않을까

과거 커밋과 Fork와 Clone과 실행 로그와 배포 파일에 사본이 남을 수 있다. 다른 사람이 옛 Clone에서 다시 Push하면 정리한 정보가 재유입될 수도 있다. 이력 재작성은 협업자에게 필요한 정리 절차까지 함께 설계한다.

### 보고할 때 비밀을 다시 노출하지 않는다

실제 키 문자열 대신 자격 증명의 이름과 발급 주체와 관련 커밋을 적는다. 사고 기록의 접근 범위도 정한다. 외부 공유용 설명에는 복구 사실과 영향과 후속 조치처럼 필요한 정보만 담는다.

### 재발 방지를 구체화한다

원인이 잘못된 add였다면 선택 검토와 ignore와 검사 체계를 고친다. 애초에 실제 키를 소스 안에 두는 구조였다면 비밀 주입 방식부터 바꾼다. 사람에게 조심하라는 말만 남기면 같은 경로가 유지된다.

저장소를 Private으로 바꾼 사실과 유출 자격 증명을 무효화한 사실을 같은 조치로 기록하지 않는다.

## 모의 훈련 30 | 가짜 키로 대응 순서 연습

실제 비밀을 만들거나 공개하지 않는다. EXAMPLE-NOT-A-REAL-KEY라는 가짜 문자열을 사용한 상황 기록만 작성한다.

1. 발견 시각과 저장소와 브랜치와 커밋 ID를 적는다.
2. 키를 회수할 서비스와 담당자를 정한다.
3. 확인할 사용 기록과 비용 기록의 위치를 적는다.
4. 교체 후 정상 동작을 확인할 방법을 정한다.
5. 동료에게 기존 사본의 재업로드 금지를 알릴 계획을 적는다.
6. .gitignore와 비밀 주입 방식과 검사 방식을 고친다.

### 사고 기록 양식

## 발생과 발견

발견 시각:

영향 저장소와 커밋:

## 차단

회수한 자격 증명의 이름:

회수 완료 시각:

## 영향과 복구

확인한 사용 기록:

복구 검증:

## 재발 방지

원인과 후속 작업:

확인 문제 저장소를 바로 Private으로 바꾸면 유출 사고가 해결될까?

추가 노출을 줄일 수는 있어도 이미 복제되거나 수집된 키는 유효할 수 있다. 접근 범위를 바꾸는 것과 자격 증명을 무효화하는 것은 별개다.

## 14장 보안 검사와 출처 검증

보안 알림의 실제 영향과 수정 우선순위를 판단한다. 서명과 해시와 출처 증거가 각각 무엇을 확인하는지 구분한다.

### 의존성은 외부 코드를 실행하는 관계다

라이브러리와 Action 및 컨테이너 이미지는 모두 외부에서 들어오는 실행 요소다. 프로젝트가 직접 작성하지 않았다는 이유로 검토 책임이 사라지지 않는다. 버전 선택과 갱신 주기 및 취약점 대응과 출처 확인을 운영 과정에 포함한다.

Action을 전체 커밋 SHA로 고정하면 이동 가능한 태그가 바뀌는 위험을 줄일 수 있다. 선택한 SHA가 공식 저장소의 의도한 판본인지 확인해야 하며 고정 이후에도 보안 갱신이 필요하다. 고정은 영원히 바꾸지 않는 정책이 아니라 변경을 명시적으로 검토하는 정책이다. [T07]

Dependabot이나 코드 스캐닝의 결과는 검토할 신호다. 경고가 없다는 사실은 알려지지 않은 취약점과 업무 권한 오류가 없다는 증거가 아니다. 반대로 경고가 있다는 사실만으로 현재 실행 경로가 반드시 취약하다고 단정할 수도 없다. 도달 가능성과 영향 및 수정 비용을 검토하고 판단 근거를 남긴다.

### 서명과 증명은 무엇을 주장하는지부터 읽는다

커밋 서명은 특정 키가 해당 객체에 서명했다는 검증에 사용된다. 키와 사람 또는 자동화 주체의 연결을 신뢰할 수 있는지도 별도 문제다. 서명이 유효해도 코드에 결함이 있을 수 있다. 출처 증명은 어떤 입력과 빌드 과정으로 결과가 만들어졌는지를 확인하는 데 도움을 주지만 그 결과가 모든 요구를 만족한다는 보장은 아니다.

SLSA는 소프트웨어 공급망의 무결성을 다루는 요구와 용어를 제공한다. 적용할 때는 실제 빌더와 증명 자료 및 검증 정책이 어떤 수준을 충족하는지 확인해야 한다. 도구를 하나 설치했다고 조직 전체가 특정 보증 수준을 달성했다고 표현하지 않는다. [T08]

### 검사 도구의 차이도 연구 대상이다

Fares 등의 2026년 프리프린트는 GitHub Actions 보안 스캐너 9개를 비교하고 596개 워크플로에서 탐지 범위와 결과의 차이를 분석했다. 프리프린트라는 출판 상태와 연구가 정한 표본 및 분류 체계를 함께 읽어야 한다. 이 책은 특정 도구의 순위를 제시하는 대신 검사 도구의 적용 범위를 확인해야 한다는 문제에 연결한다. [S16]

lesson-lab의 공급망 점검에서도 외부 Action의 출처와 권한 및 입력을 직접 읽는다. 스캐너가 탐지하기 어려운 업무 맥락은 사람이 판단해야 한다. 여러 도구의 결과를 합쳤다는 사실보다 각각 어떤 실패를 탐지하고 어떤 실패를 놓칠 수 있는지 설명하는 편이 전문적인 검토에 가깝다.

## 증명이 존재하는 것과 기대한 증명인 것은 다르다

artifact attestation은 어떤 워크플로와 소스 문맥에서 산출물이 만들어졌는지 확인하는 데 사용된다. 검증자는 서명 자체뿐 아니라 산출물의 다이제스트와 허용한 저장소 및 워크플로 같은 정책 조건을 확인해야 한다. 예상하지 않은 제작자가 만든 유효한 증명을 받아들이면 검증의 목적을 달성하지 못한다. [T27]

같은 파일 이름의 두 PDF가 있다고 하자. 이름이 같아도 바이트가 다르면 서로 다른 산출물이다. 증명이 가리키는 다이제스트와 사용자가 받은 파일의 다이제스트를 연결해야 한다. 소스 저장소의 이름만 확인하고 실제 파일을 대조하지 않으면 중간 교체를 놓칠 수 있다.

증명은 빌드의 출처를 설명하며 프로그램의 모든 성질을 증명하지 않는다. 신뢰한 워크플로가 취약한 의존성을 포함할 수도 있다. 출처 검증과 의존성 분석 및 기능 시험을 결합해야 한다. 각각의 도구가 어떤 주장을 뒷받침하는지 표로 정리하면 보안 기능을 컷다는 포괄적인 표현보다 정확하다.

## SBOM은 성분 목록과 실제 영향 분석 사이에 놓인다

소프트웨어 자재 명세서는 포함된 구성 요소를 파악하는 데 도움이 된다. 어떤 패키지가 있다는 사실은 취약한 코드 경로가 실제로 실행된다는 사실과 같지 않다. 반대로 직접 선언한 패키지 목록만으로 전이 의존성을 모두 파악했다고 할 수도 없다. 구성 목록과 실행 경로 및 배포 환경을 함께 확인해야 영향을 평가할 수 있다.

새 취약점이 알려졌을 때는 영향받는 버전과 산출물을 식별하고 수정 후 검증 범위를 정한다. 이름이 비슷한 패키지나 다른 환경의 결과를 혼동하지 않는다. 업데이트를 적용했다는 기록에는 실제 배포 판본과 남은 예외도 연결한다. 보안 작업의 완료는 알림을 달는 행위보다 실행 중인 시스템의 상태로 판단한다.

## 14.1 Dependabot과 Code scanning과 공급망

관련 근거 [R46] [R47] [R48]

## 외부 코드도 프로젝트의 일부다

내가 작성한 소스만 안전해도 의존성이나 빌드 Action이 취약하면 문제가 생긴다. Dependabot의 업데이트 PR과 취약점 알림과 Code scanning과 Secret scanning은 서로 다른 위험을 다룬다.

도구	주로 찾는 것	대신하지 못하는 것
Dependabot 업데이트	새 의존성 버전	모든 기능 검증
취약점 알림	알려진 취약 의존성	미공개 취약점 탐지
Code scanning	코드의 위험 패턴	모든 런타임 결함
Secret scanning	인식 가능한 비밀 노출	모든 개인정보 탐지

## 업데이트도 검토가 필요하다

버전 번호가 작게 올라가도 동작이 달라질 수 있다. 변경 설명과 테스트와 배포 영향을 읽는다. 보안 수정을 영원히 미루지 않도록 담당자와 기한을 정한다. 자동 병합은 검증 범위와 위험 수준에 맞게 제한한다.

## 공급망 증거

잠금 파일은 설치 버전을 재현하는 데 도움을 준다. SBOM은 포함된 구성요소 목록을 정리한다. Artifact attestation은 산출물의 출처와 빌드 맥락을 검증하는 데 쓴다. 서명이 있다는 사실과 코드가 안전하다는 사실은 다르다.

보안 기능은 저장소 종류와 조직 요금제와 추가 제품 활성화 여부에 따라 달라진다. 메뉴가 없으면 현재 계정의 지원 조건과 권한을 확인한다.

## 보안 알림의 우선순위를 정하는 근거

### 알림 이름보다 실제 노출 경로를 읽는다

취약한 의존성이 있다는 사실과 내 서비스에서 그 취약한 기능을 외부 입력으로 사용할 수 있다는 사실은 다를 수 있다. 그렇다고 사용하지 않을 것 같다는 추측만으로 닫지는 않는다. 포함 경로와 실행 환경과 영향 기능을 확인한다.

분류	조사할 질문
직접 의존성	내가 지정한 패키지인가
간접 의존성	어느 상위 패키지가 포함했는가
개발 도구	빌드나 CI에서도 공격 경로가 있는가
실행 의존성	운영 환경에서 어떤 기능에 연결되는가
수정 버전	호환성을 유지하며 올릴 수 있는가

### 자동 갱신 PR을 읽는 순서

버전 범위와 잠금 파일의 차이를 읽는다. 릴리스 설명에서 호환성 변화를 확인한다. 프로젝트의 검사 결과와 실제 사용하는 기능을 확인한다. 보안 수정이 지연되면 가능한 완화와 담당 시점을 기록한다.

### 도구의 탐지 범위를 구분한다

Secret scanning이 조용하다고 개인정보가 없다고 판단할 수는 없다. Code scanning이 통과했다고 모든 실행 경로가 안전하다는 뜻도 아니다. 도구가 확인한 범위와 사람이 추가로 검토할 범위를 문서에 나누어 적는다.

### 자동 병합을 적용한다면

수정 종류와 의존성 중요도와 테스트 수준을 기준으로 범위를 정한다. 모든 보안 PR을 영원히 수동 대기시키는 것과 모든 갱신을 즉시 자동 병합하는 것 사이에서 실제 검토 용량과 위험을 함께 조절한다.

알림을 담은 이유도 자산이다. 미사용과 오탐과 수정 완료를 구분하고 나중에 조건이 바뀌면 다시 확인할 수 있게 남긴다.

## 실습 14.1 | GitHub Actions 갱신 PR 받기

아래 파일은 Action 버전 갱신 감시를 위한 완성된 예제다. npm 프로젝트라면 package.json 등이 있는 디렉터리에 별도 npm 항목을 추가할 수 있다.

```
.github/dependabot.yml
version: 2
updates:
 - package-ecosystem: github-actions
 directory: /
```

```
schedule:
 interval: weekly
open-pull-requests-limit: 5
```

## 알림 처리 기록

1. 영향받는 저장소와 버전과 경로를 확인한다.
2. 실제 사용하는 기능인지와 외부 입력 경로를 검토한다.
3. 업데이트 또는 완화 변경을 PR로 만든다.
4. 테스트와 배포 후 알림 상태를 확인한다.
5. 오탐이나 미사용으로 닫는다면 근거를 남긴다.

기록 항목	적용 내용
영향	어떤 사용자와 환경이 노출되는가
조치	올릴 버전 또는 차단 방법
검증	수정 후 무엇을 확인했는가
책임	담당자와 다음 확인 시점

확인 문제 Dependabot PR이므로 리뷰 없이 모두 병합해도 될까?  
신뢰할 수 있는 갱신 제안이어도 프로젝트와의 호환성은 확인해야 한다. 변경 규모와 테스트 수준에 맞는 검토 정책을 정한다.

## 14.2 서명과 출처 검증과 신뢰

관련 근거 [R60] [R61]

### 누구의 변경인지와 무엇이 안전한지는 다르다

GitHub의 Verified 표시는 지원되는 방식의 커밋 또는 태그 서명을 검증한 결과다. 해당 코드에 취약점이 없다는 보증은 아니다. 서명 키의 신원 연결과 보관과 만료와 회수 정책을 함께 관리한다.

증거	보여 주는 것	보장하지 않는 것
커밋 서명	연결된 키로 서명한 기록	기능 정확성
필수 리뷰	정한 검토 절차 통과	모든 결함 발견
CI 성공	구성한 검사 통과	미검사 영역의 안전
Artifact attestation	산출물 출처의 검증 근거	소스 자체의 무결점

## SSH 서명과 SSH 접속은 목적이 다르다

지원되는 Git 버전에서 SSH 키로 커밋 서명을 설정할 수 있다. GitHub에 등록할 때 Authentication key와 Signing key의 용도를 구분한다. 같은 키를 재사용하는 선택과 별도 키를 쓰는 선택은 조직의 보안 정책에 맞춘다.

## 산출물의 연속성

검사한 소스와 배포한 파일이 연결돼야 한다. 한 번 빌드한 산출물을 검증하고 같은 산출물을 배포하는 방식은 단계마다 다시 빌드해 내용이 달라지는 위험을 줄인다. 서명과 해시와 빌드 출처를 이 연결에 사용한다.

서명 강제 규칙을 켜기 전에 웹 편집과 봇과 자동 업데이트와 비상 복구의 서명 방식도 확인한다. 정상 자동화가 모두 막히는지 연습 저장소에서 시험한다.

## 서명과 해시와 출처 증거의 다른 역할

### 파일이 같다는 것과 누가 만들었는지는 다르다

해시는 내용을 식별하는 데 도움을 준다. 서명은 어떤 키가 해당 기록을 서명했는지 검증하는 근거다. 출처 정보는 어떤 소스와 빌드 과정에서 산출물이 나왔는지 연결한다. 이 셋이 각각 답하는 질문을 구분한다.

근거	답하는 질문
콘텐츠 해시	이 파일의 바이트가 같은가
커밋 서명	해당 기록이 어떤 키와 연결되는가
리뷰 기록	어떤 검토 절차를 거쳤는가

## 근거

## 답하는 질문

CI 결과

무엇을 어떤 환경에서 검사했는가

Attestation

어떤 빌드 출처 주장을 검증하는가

### Verified만 보고 멈추지 않는다

서명 검증은 코드에 취약점이 없다는 검사와 다르다. 올바른 사람이 실수한 코드도 서명할 수 있다. 검토와 테스트와 권한과 출처를 서로 보완하는 근거로 사용한다.

### 같은 산출물을 이어서 사용

검사한 뒤 배포 단계에서 전혀 새롭게 빌드하면 환경 차이로 결과가 달라질 수 있다. 검증한 산출물의 해시와 출처를 남기고 배포가 그 결과물을 사용하는지 확인한다. 배포 시점에 어떤 파일이 실제 적용됐는지도 기록한다.

### 키가 바뀌거나 담당자가 바뀌면

서명 키 등록과 회수와 새 담당자의 사용 절차를 정한다. 키를 개인 메신저로 넘기는 방식보다 새 주체에 맞는 자격을 마련한다. 과거 서명의 표시와 현재 키 접근 권한은 문맥이 다를 수 있으므로 공식 검증 안내를 함께 읽는다.

신뢰할 주체와 검증 도구의 기준을 적는다. 서명이 있다는 문장만으로는 무엇을 신뢰한 것인지 충분히 설명하지 못한다.

### 설계 실습 14.2 | 서명 적용 점검

실제 키 경로와 등록을 마친 뒤에만 아래 설정을 적용한다. .pub는 공개 키 경로다. 키 파일을 새로 생성하거나 기존 키를 덮어쓰는 절차는 아니다.

```
git config --local gpg.format ssh
git config --local user.signingkey /PATH/TO/SIGNING-KEY.pub
git config --local commit.gpgsign true
git config --local --get user.signingkey
```

새 연습 커밋을 만든 뒤 GitHub의 서명 표시를 확인한다. 로컬 검증을 하려면 allowed signers 같은 신뢰 설정도 필요하다. GitHub에서 보이는 검증과 로컬 신뢰 설정을 같다고 생각하지 않는다.

### 배포 파일 출처 확인 질문

어느 저장소와 커밋에서 빌드했는가. 어떤 Workflow와 Runner에서 만들었는가. 파일 해시는 무엇인가. 검증 도구는 무엇을 신뢰하는가. 내려받은 뒤에도 이 근거를 확인할 수 있는가. 각 질문을 릴리스 기록과 연결한다.

확인 문제 Verified가 붙지 않은 커밋은 모두 악성일까?

서명을 하지 않았거나 검증 조건이 맞지 않을 수 있다. 표시는 하나의 신원 근거다. 조직 정책과 변경 내용과 검토 증거를 함께 판단한다.

## 15장 백업과 비용과 운영 인수인계

운영 자원과 담당 역할과 비용을 기록한다. Bundle의 포함 범위를 이해하고 새 환경에서 복원해 백업이 실제로 쓸 수 있는지 확인한다.

### Git 이력만으로 GitHub 프로젝트 전체가 복원되지는 않는다

Git 저장소에는 커밋과 tree 및 blob과 참조가 들어 있다. GitHub의 이슈와 PR 대화 및 조직 권한과 환경 설정은 별도의 서비스 데이터다. 따라서 저장소를 mirror clone하거나 bundle로 보관하는 작업은 중요하지만 프로젝트 운영 상태 전체의 백업과 같지는 않다.

lesson-lab에 LFS 파일과 Release 첨부 파일 및 Pages 설정이 있다면 각각의 보존 방법을 확인해야 한다. 복원 목표를 먼저 정한 뒤 필요한 자산 목록을 만든다. 소스 이력만 되살리면 되는 학습 저장소와 사용자에게 같은 서비스를 다시 제공해야 하는 운영 저장소는 요구가 다르다.

### 복원 시험은 사본의 품질을 측정한다

bundle verify는 번들의 구조와 필요한 전제 등을 확인하는 데 도움이 된다. 그러나 애플리케이션이 실제로 실행되는지는 별도의 검사다. 분리된 폴더에서 복원하고 필요한 참조와 파일 및 테스트를 확인한다. 외부 의존성과 비밀 설정이 빠졌다면 소스 복원에는 성공해도 서비스 복구에는 실패할 수 있다.

백업 주기와 보존 기간은 변경 빈도와 허용 손실 및 복구 비용에 따라 정한다. 자동 백업 작업의 성공 여부를 감시하고 정기적으로 복원해 본다. 오래된 사본이 계속 쌓이는 것과 복구 가능한 사본이 유지되는 것은 다르다.

### 비용은 자원 사용과 정책의 결과다

Actions 실행 시간과 저장 공간 및 개발 환경의 사용량은 프로젝트 구조에 영향을 받는다. 불필요한 Matrix 조합과 중복 실행 및 오래된 아티팩트를 줄일 수 있다. 그러나 검사를 줄이는 결정은 어떤 위험이 증가하는지 함께 평가해야 한다. 비용 절감과 검증 범위를 별개의 문제로 취급하지 않는다.

인수인계에는 소유자와 결재 및 자격 증명과 복구 담당자를 포함한다. 한 사람의 개인 계정에만 중요한 권한이 남아 있으면 기술 문서가 완벽해도 운영이 중단될 수 있다. 조직의 지속 가능성은 코드와 사람 및 권한 구조가 함께 결정한다.

## bundle의 범위와 전제 객체를 이해한다

git bundle은 참조와 관련 객체를 파일로 전달하는 방법이다. 모든 필요한 객체를 포함한 bundle과 이전 객체를 전제로 하는 증분 bundle은 복원 조건이 다르다. old..new 같은 범위로 만들었다면 수신 쪽에 old와 필요한 이력이 있어야 할 수 있다. bundle verify는 이런 조건을 확인하는 데 사용한다. [T24]

```
git bundle create lesson-lab.bundle --all
git bundle verify lesson-lab.bundle
git clone lesson-lab.bundle recovered-lab
```

이 예시는 Git 참조에서 도달하는 이력을 파일로 보존하고 별도 폴더에 복원하는 흐름이다. 추적되지 않은 파일과 로컬 환경 변수 및 GitHub의 이슈와 권한 설정까지 모두 담는 방식은 아니다. LFS의 실제 객체와 submodule의 별도 저장소도 보존 범위를 확인해야 한다. 백업 파일이 하나라는 편리함과 프로젝트 전체가 포함되었다는 판단을 구분한다.

## 복원 목표는 시간과 자료 손실의 허용 범위로 표현한다

복구 시간 목표는 장애 이후 어느 정도 안에 필요한 기능을 되살릴지에 관한 기준이다. 복구 시점 목표는 어느 시점의 자료까지 되살릴 수 있어야 하는지에 관한 기준이다. 매일 백업하는 서비스와 매시간 백업하는 서비스는 같은 장애에서도 잃을 수 있는 자료의 범위가 다르다. 두 목표는 같은 숫자로 표현할 필요가 없다.

교육 자료 저장소는 하루 전 판본으로 복원해도 수업을 이어갈 수 있지만 실시간 제출 시스템은 최근 답안 손실이 큰 영향을 줄 수 있다. 자산의 성격에 따라 보존 주기와 복원 우선순위를 정한다. 모든 자료에 가장 높은 수준을 적용하면 비용과 관리 부담이 커지므로 실제 요구에 연결해야 한다.

복원 연습에서는 파일을 꺼내는 데 걸린 시간뿐 아니라 실행 환경 준비와 권한 연결 및 결과 검증에 걸린 시간을 측정한다. 압축 해제에 성공한 시점이 서비스 복구의 완료 시점은 아니다. 다른 사람이 문서를 따라 복원할 수 있는지도 시험하면 개인의 기억에 의존한 절차를 발견할 수 있다.

상태 변화 / 백업은 복원으로 확인한다

사본

생성됨

분리 환경

자료 검사

실행 검사

미준비

미실행

미실행

1 / 4 파일 존재는 복원 성공의 증거가 아니다

Git bundle의 복원과 GitHub 서비스 메타데이터의 복원은 범위가 다르다. LFS와 Release 파일도 별도로 확인한다.

사본

보관됨

분리 환경

자료 검사

실행 검사

복원 완료

통과

테스트 통과

4 / 4 프로그램과 운영 절차까지 시험한다

Git bundle의 복원과 GitHub 서비스 메타데이터의 복원은 범위가 다르다. LFS와 Release 파일도 별도로 확인한다.

Git bundle의 복원과 GitHub 서비스 메타데이터의 복원은 범위가 다르다. LFS와 Release 파일도 별도로 확인한다.

- 1. 파일 존재는 복원 성공의 증거가 아니다
- 2. 원본과 분리된 위치에서 복원을 시작한다
- 3. 필요한 이력과 파일 및 외부 자료를 확인한다
- 4. 프로그램과 운영 절차까지 시험한다

15.1 백업과 비용과 인수인계

관련 근거 [R62] [R35] [R63]

저장소 복제만으로 서비스 전체가 백업되지는 않는다

Git Clone은 코드와 이력을 다룬다. Issue와 PR 대화와 Projects와 Releases 첨부과 Secrets와 권한과 외부 배포 설정을 모두 동일하게 복구하는 것은 아니다. 무엇을 보관하고 어떻게 복원할지 자원별로 정한다.

대상	보관과 복구 계획
Git 이력	Mirror 또는 검증한 백업
LFS 파일	실제 객체 별도 확보
Issue와 PR 정보	API와 내보내기 도구 검토
Release 첨부	파일과 버전 연결 보관
권한과 규칙	설정 기록과 재적용 절차
비밀과 도메인	안전한 별도 인수인계

비용은 실행과 저장을 나눠 본다

Actions 실행과 Artifact와 캐시와 Codespaces와 LFS와 AI 기능의 사용량을 따로 확인한다. 포함량과 단가를 책의 고정 숫자로 외우지 않는다. 실행 머신 종류와 계정과 요금제에 따라 달라질 수 있다. 예산 경보가 항상 사용 종단을 뜻하지는 않으므로 실제 종단 동작도 확인한다.

유지 종료와 아카이브

더 이상 관리하지 않는 저장소에는 마지막 지원 버전과 후속 경로를 적는다. Archive는 저장소를 읽기 위주로 전환하는 운영 선택이다. 외부 서비스와 토큰과 청구를 자동으로 모두 종료하는 절차가 아니다.

백업 성공 메시지보다 복구 시험이 중요하다. 새 위치에서 파일과 이력과 필요한 메타데이터가 실제로 돌아오는지 정기적으로 확인한다.

## 백업 목록보다 복원 시나리오를 먼저 작성

### 어떤 사고에서 무엇을 되살릴지 정한다

로컬 PC 분실과 저장소 삭제와 계정 접근 상실과 외부 배포 장애는 필요한 복구가 다르다. 코드의 Mirror 사본이 있어도 도메인 계정과 비밀번호와 데이터가 없으면 서비스 전체를 다시 실행하지 못할 수 있다.

사고 상황	필요한 복구 근거
로컬 디스크 손실	원격 커밋과 미커밋 파일 별도 보관
저장소 접근 문제	소유 조직과 관리자와 복구 경로
코드 이력 손실	검증한 Git 백업과 참조
배포 장애	실행 버전과 배포 설정과 비밀 주입
담당자 교체	역할과 결제와 도메인과 운영 절차

### 복원 시험을 실제로 한다

새 풀더나 새 환경에서 저장소를 받고 실행 안내를 따라가 본다. 필요한 파일과 버전과 권한이 빠지지 않았는지 확인한다. 백업 도구의 성공 로그와 사람이 서비스를 다시 쓰는 성공은 다른 단계다.

### 비용의 책임 주체

개인 계정의 실행인지 조직 결제인지 확인한다. Codespaces와 Actions와 저장량과 패키지 등 각 자원의 사용량을 읽는다. 경보를 설정했다면 경보만 보내는지 사용을 막는 조건도 있는지 실제 정책을 확인한다.

### 종료할 때 남은 자원

저장소를 Archive해도 외부 클라우드와 도메인과 토큰과 청구가 모두 자동으로 정리되지는 않는다. 더 쓰지 않는 연결과 권한과 자원을 목록으로 확인하고 필요한 기록을 보관한다.

인수인계의 완료 기준은 파일 전달이 아니다. 새 담당자가 정상 작업과 복구 작업을 직접 재현할 수 있는지 확인한다.

## 실습 15.1 | 코드 백업의 범위 확인

아래는 새 백업 폴더를 만드는 예제다. 쓰기 대상 원격에 mirror Push하는 명령은 넣지 않는다. 잘못 쓰면 대상 참조를 덮어쓸 수 있기 때문이다.

```
git clone --mirror https://github.com/YOUR-ID/lesson-lab.git less
on-lab-backup.git
git -C lesson-lab-backup.git fsck --full
git clone lesson-lab-backup.git lesson-restore-check
git -C lesson-restore-check log -5 --oneline
```

LFS를 쓴다면 Git 이력 외에 실제 LFS 객체 확보 여부를 별도로 확인한다. fsck는 Git 객체 검사를 도와주지만 서비스 전체 복원 성공을 증명하지는 않는다.

### 인수인계 체크

소유 조직과 관리자를 확인한다. 결제 담당과 도메인 소유자와 외부 배포 연결을 적는다. 토큰은 개인 값 전달 대신 새 소유 주체의 자격 증명으로 바꾼다. 정상 배포와 장애 복구를 새 담당자가 직접 해 본다.

확인 문제 GitHub 계정만 넘겨주면 운영 인수인계가 끝날까?  
계정을 공유하지 않는다. 조직 권한과 배포와 결제와 도메인과 비밀과 복구 절차를 각각 새 담당자에게 이전하고 확인한다.

## 15.2 Bundle로 오프라인 이력 보관과 복원

관련 근거 [R83] [R62]

### 단일 파일에 Git 객체와 참조를 묶는다

Git bundle은 네트워크 없이 Git 이력을 전달하거나 보관할 때 쓴다. ZIP과 달리 Git 객체와 참조를 담는다. 모든 커밋을 자동으로 담는다는 뜻은 아니며 생성할 때 지정한 참조와 도달 가능한 객체가 범위다.

```
git status
git bundle create ../lesson-lab.bundle --all
git bundle verify ../lesson-lab.bundle
git bundle list-heads ../lesson-lab.bundle
```

--all은 로컬에서 보이는 참조를 기준으로 한다. GitHub에만 있고 아직 가져오지 않은 브랜치를 자동으로 모두 수집하는 명령이 아니다. 미커밋 파일과 LFS의 실제 파일과 Issue와 Secret은 별도 보관 대상이다.

범위	Bundle만으로 충분한가
포함한 참조의 Git 커밋	복원 시험으로 확인 가능
저장하지 않은 편집	포함하지 않음
GitHub Issue와 PR 대화	포함하지 않음
LFS 실제 객체	별도 확보 필요
계정과 배포 권한	별도 인수인계 필요

Bundle에는 과거의 민감한 내용도 들어갈 수 있다. 단일 파일이 되었다고 일반 첨부로 자유롭게 공유하지 않는다. 보관 위치와 암호화와 접근 범위를 정한다.

## Bundle을 복원 가능한 백업으로 검증

### 파일이 만들어졌다는 사실만으로 충분하지 않다

Bundle은 선택한 Git 객체와 참조를 옮기는 수단이다. 작업 폴더의 모든 자료를 자동으로 넣는 압축 파일과 다르다. 커밋하지 않은 편집과 무시한 파일과 외부 자원을 따로 파악한다.

대상	별도 확인할 점
Git 커밋과 참조	원하는 범위를 Bundle에 넣었는가
미커밋 편집	별도로 보관했는가
Git LFS 객체	외부 저장 객체를 확보했는가
Submodule	별도 저장소도 확보했는가
Issue와 설정과 비밀	Git 백업 외 절차가 있는가

### 독립 복원과 증분 전달

모든 이력을 담은 Bundle과 이전 객체를 전제로 하는 증분 Bundle은 복원 조건이 다르다. 파일 이름만 보고 독립 복원 가능성을 판단하지 않는다. git bundle verify의 결과와 필요한 선행 객체를 확인한다.

## 새 폴더에서 직접 복원

원래 작업 폴더와 다른 위치에 Clone한다. 브랜치와 태그와 최신 커밋을 확인한다. 파일을 열고 실행 안내를 따라가 본다. 백업을 만들었던 PC의 전역 설정과 인증과 설치 도구가 없어도 필요한 절차를 찾을 수 있는지 확인한다.

## 보관 기록에 적을 정보

생성 시점과 대상 저장소와 포함 범위와 파일 해시와 복원 시험 결과를 남긴다. 암호화와 접근 권한과 보관 위치도 자료의 민감도에 맞춘다. 백업이 있는지보다 누가 어떤 조건에서 읽고 복원할 수 있는지 설명할 수 있어야 한다.

Bundle 사본에도 비공개 코드와 과거에 기록한 비밀이 포함될 수 있다. 파일을 외부로 전달하기 전에 포함 범위를 검토한다.

## 실습 15.2 | 새 폴더에서 복원 확인

새 복원 폴더 이름을 정해 기존 작업과 겹치지 않게 한다. 아래는 Bundle 파일이 현재 폴더의 한 단계 위에 있는 상황이다.

```
git clone ../lesson-lab.bundle ../lesson-restored
git -C ../lesson-restored log -5 --oneline
git -C ../lesson-restored branch -a
git -C ../lesson-restored fsck --full
```

복원한 폴더의 origin은 GitHub가 아니라 Bundle 경로일 수 있다. 실서비스 저장소로 Push하려면 대상과 목적을 새로 확인한다. 복원 시험을 했다는 이유로 원래 백업 파일을 바로 지우지 않는다.

## 증분 Bundle의 추가 조건

특정 기준 이후의 변경만 묶으면 파일은 작아지지만 받는 쪽이 기준 객체를 이미 가져야 한다. 전체 백업과 증분 백업의 순서와 기준을 함께 보관해야 한다. 이번 입문 실습에서는 --all로 만든 전체 참조 묶음만 사용한다.

확인 문제 Bundle을 만들었으니 아직 저장하지 않은 HTML 편집도 보관됐을까?  
Git 객체로 기록되지 않은 작업 파일은 포함되지 않는다. 필요한 변경을 커밋하거나 별도 파일 백업으로 보존한다.

## 16장 이력 재작성과 원인 조사

개인 이력을 정리하고 공유된 기준이 바뀌는 영향을 검토한다. Worktree와 Bisect와 range-diff를 조사 목적에 맞게 활용한다.

### 재작성의 목적을 먼저 제한한다

Interactive rebase는 커밋을 묶거나 순서를 바꾸고 메시지를 다듬는 데 사용할 수 있다. 목적은 검토 가능한 논리적 이력을 만드는 것이다. 이미 공유된 이력을 바꿀 때는 다른 작업자의 참조와 자동화에 미치는 영향을 확인한다. 정리된 그래프의 미관만으로 협업 비용을 정당화할 수는 없다.

force-with-lease는 원격 참조가 예상 상태일 때 갱신하도록 조건을 둔다. 다만 기대값을 명시하지 않는 형태는 로컬 원격 추적 참조에 의존하므로 백그라운드 fetch가 그 기준을 바꿀 수 있다. 특정 원격 커밋을 읽고 그 값으로 조건을 명시하는 방법을 이해하면 보호 장치가 무엇을 비교하는지 더 분명해진다. 이 옵션도 팀 정책과 권한 제한을 대체하지 않는다. [T09]

재작성 전후에는 단순히 커밋 개수만 보지 않는다. 최종 파일 상태와 패치의 의미가 유지되는지 확인한다. range-diff는 두 커밋 범위의 패치 시리즈를 비교하는 데 유용하다. 테스트와 함께 사용하면 정리 과정에서 의도하지 않은 변경이 들어갔는지 살피는 근거가 된다.

### bisect는 판정 가능한 질문이 있어야 작동한다

git bisect는 알려진 좋은 상태와 나쁜 상태 사이에서 문제를 도입한 커밋을 좁힌다. 후보를 나누어 검사하기 때문에 선형으로 모든 커밋을 보는 것보다 효율적일 수 있다. 그러나 성공과 실패를 판정하는 검사가 안정적이어야 한다. 같은 커밋이 실행할 때마다 다른 결과를 내면 탐색의 의미가 약해진다.

자동 bisect 스크립트의 종료 코드는 계약이다. 성공과 실패 및 검사 불가능을 구분해야 한다. 해당 커밋이 옛 런타임을 요구해 실행하지 못하는 경우를 기능 실패로 판정하면 잘못된 원인을 찾을 수 있다. 테스트 환경의 변화와 실제 회귀를 분리하는 과정이 필요하다.

## worktree는 조사와 개발의 상태를 분리한다

worktree를 사용하면 같은 저장소 객체를 공유하면서 다른 작업 트리를 둘 수 있다. 현재 작업을 억지로 치우지 않고 과거 판본을 조사하거나 긴 테스트를 돌릴 수 있다. 하지만 작업 디렉터리의 의존성 파일과 외부 데이터베이스까지 자동으로 분리되는 것은 아니다. 각 작업 트리의 실행 설정을 확인한다.

원인 커밋을 찾았다고 분석이 끝나는 것은 아니다. 왜 그 변경이 기존 테스트를 통과했는지와 요구가 어디에서 누락되었는지 확인한다. 커밋은 원인이 드러난 위치를 가리키며 조직적 원인이나 설계의 빈틈까지 모두 설명하지는 않는다.

## 커밋의 새 식별자와 변경의 새 의미를 구분한다

rebase 뒤 커밋 식별자가 달라지는 것은 예상되는 현상이다. 중요한 질문은 변경의 의도가 유지되었는지다. git range-diff는 두 패치 계열을 비교해 재작성 전후의 관계를 읽는 데 도움을 준다. 일반 diff가 최종 스냅샷의 차이를 보는 데 쓰인다면 range-diff는 변경 묶음 사이의 대응을 이해하는 도구다. [T23]

충돌 해결을 포함한 rebase에서는 자동 적용이 끝났다고 의미가 보존되었다고 가정하지 않는다. 이전 계열과 새 계열을 비교하고 기존 검사와 상호작용 검사를 다시 수행한다. 커밋이 합쳐지거나 순서가 바뀌면 중간 상태의 실행 가능성이 달라질 수 있다. 팀이 커밋 단위로 bisect하는지에 따라 중간 상태의 품질도 중요해진다.

## diff는 유일한 객관적 서술이 아니다

두 파일 사이의 변화는 여러 방식으로 줄을 대응시켜 표현할 수 있다. 반복되는 문장과 이동한 코드가 많으면 선택한 diff 알고리즘에 따라 변경 묶음의 모양이 달라질 수 있다. Nugroho와 동료들의 연구는 Git diff 알고리즘 선택이 코드 변경량과 결함 유입 추정 같은 분석에 영향을 줄 수 있음을 보고했다. 연구의 대상 프로젝트와 분석 절차를 현재의 모든 저장소로 일반화하지 않되 측정 도구의 선택도 결과의 일부임을 기억한다. [S23]

리뷰에서 차이가 이해되지 않으면 다른 알고리즘과 공백 표시 옵션을 비교할 수 있다. 다만 보기 좋은 결과를 얻기 위해 중요한 공백 변화나 생성 규칙의 변화를 숨겨서는 안 된다. 분석의 목적과 옵션을 기록해야 다른 사람이 같은 결과를 재현할 수 있다.

bisect의 판정 스크립트도 측정 도구다. 테스트가 간헐적으로 실패하거나 외부 서비스 상태에 따라 달라지면 탐색 결과가 흔들린다. 정상과 실패를 안정적으로 구분할 수 있는 최소 조건을 먼저 마련한다. 찾은 커밋은 원인 후보이며 그 변경이 실패를 일으키는 기제를 설명하고 검증하는 단계가 뒤따른다.

상태 변화 / rebase는 새로운 부모로 새 커밋을 만든다

**rebase는 새로운 부모로 새 커밋을 만든다**

1 / 4   작업 커밋 C는 A를 부모로 가진다

<b>기준 main</b> A	<b>원래 커밋</b> C의 부모 A	<b>재적용</b> 미시작	<b>작업 참조</b> C
---------------------	-------------------------	-------------------	-------------------

C를 그대로 이동시키는 것이 아니다. 원래 커밋 객체와 새 커밋 객체를 구분해야 공유 이력의 재작성 위험을 이해할 수 있다.

● ● ● ●

**rebase는 새로운 부모로 새 커밋을 만든다**

4 / 4   부모가 달라진 새 객체 C2를 가리킨다

<b>기준 main</b> B	<b>원래 커밋</b> 원래 C 유지	<b>재적용</b> 새 C2의 부모 B	<b>작업 참조</b> C2
---------------------	-------------------------	--------------------------	--------------------

C를 그대로 이동시키는 것이 아니다. 원래 커밋 객체와 새 커밋 객체를 구분해야 공유 이력의 재작성 위험을 이해할 수 있다.

● ● ● ●

C를 그대로 이동시키는 것이 아니다. 원래 커밋 객체와 새 커밋 객체를 구분해야 공유 이력의 재작성 위험을 이해할 수 있다.

- 1. 작업 커밋 C는 A를 부모로 가진다
- 2. 기준 브랜치가 B까지 진행한다
- 3. 변경을 B 위에 다시 적용한다
- 4. 부모가 달라진 새 객체 C2를 가리킨다

16.1 Interactive rebase와 안전한 이력 재작성

관련 근거 [R12] [R11]

## 내 브랜치의 커밋을 검토 가능한 단위로 정리한다

Interactive rebase는 커밋을 합치거나 순서를 조정하거나 메시지를 고친다. 아직 공유하지 않은 개인 브랜치에서 먼저 익힌다. 파일 변경이 같더라도 새 커밋이 생기므로 협업 중인 이력에는 영향을 알리고 진행한다.

```
git status
git branch backup/before-interactive
git rebase -i HEAD~3
```

위 명령에는 최소 세 개의 대상 커밋과 그 앞 기준이 필요하다. 편집기의 목록은 오래된 커밋부터 나열된다. 첫 줄은 pick으로 두고 이어지는 작은 수정 커밋을 squash나 fixup으로 바꿀 수 있다.

명령	의미
pick	커밋 사용
reword	메시지 수정
edit	멈춰서 내용 수정
squash	앞 커밋에 합치고 메시지 편집
fixup	앞 커밋에 합치고 이 메시지는 버림
drop	커밋 제거

## --force-with-lease도 조건을 이해한다

그냥 --force보다 원격의 예상 상태를 검사할 수 있다. 하지만 기본 lease는 로컬 원격 추적 참조에 기대므로 백그라운드 fetch의 영향을 받을 수 있다. 중요한 재작성에서는 예상 원격 커밋을 명시하고 동료 변경 여부를 확인한다.

이 장의 절차는 본인 작업 브랜치에만 사용한다. 보호 규칙을 끄거나 남의 커밋을 덮어쓰는 용도가 아니다. 브랜치 백업 외에 미커밋 파일도 따로 보존한다.

## Interactive rebase의 편집 목록과 Lease 읽기

### 편집 목록은 오래된 커밋부터 이어진다

개인 커밋 세 개를 정리할 때 어떤 기록의 뒤에 어떤 변경이 붙는지 먼저 읽는다. 예를 들어 시작 변경과 그 수정과 설명 보장을 하나로 묶으려면 앞 기록을 기준으로 뒤 기록을 합친다. 첫 줄부터 무조건 fixup으로 바꾸는 식으로 사용하지 않는다.

```
pick a111111 Add answer control
fixup b222222 Fix button label
reword c333333 Add usage guide
```

위는 편집 목록 모양을 보여 주는 예시다. 실제 ID는 자신의 목록에 나온 값을 유지한다. fixup은 앞 기록에 변경을 합치고 해당 메시지는 보통 버린다. reword는 커밋 메시지를 다시 쓰는 선택이다. 실제 기능 변경이 섞이는지 파일 차이도 확인한다.

### expected_oid를 왜 저장하는가

명시적 lease는 원격 브랜치가 내가 확인한 커밋일 때만 재작성을 허용하도록 조건을 준다. expected_oid는 그 예상 값을 별도로 기억한 변수다. 재작성 중 다른 사람이 원격을 갱신했다면 거절을 읽고 그 변경부터 확인한다.

### 거절 후 값을 새로 받기만 하면

기술적으로 새 상태를 기대값으로 삼을 수는 있어도 동료의 변경을 이해하지 않은 덮어쓰기 위험은 남는다. 무엇이 달라졌는지와 어떻게 통합할지 판단한 뒤 다음 절차를 정한다.

### 정리 뒤 두 가지 비교

최종 파일 상태의 비교와 커밋 열의 비교를 나누어 본다. 메시지 정리만 했다면 파일 diff는 비어있어야 할 수 있다. 패치의 묶음과 순서를 바꿨다면 Range-diff 같은 도구로 의도한 변화인지 검토한다.

Lease는 협업 동의를 대신하지 않는다. 원격 검사에 통과해도 동료의 로컬 작업 기준이 달라질 수 있다.

## 실습 16.1 | 명시적 lease로 개인 브랜치 갱신

다른 사람이 수정하지 않는 연습 브랜치 feature/history-cleanup을 준비한다. 적어도 세 개의 개인 커밋을 Push해 두고 작업 폴더를 깨끗하게 만든다. 아래는 Git Bash용이다.

```
git switch feature/history-cleanup
git fetch origin
```

```
git status
git branch backup/before-push-rewrite
expected_oid=$(git rev-parse origin/feature/history-cleanup)
git rebase -i HEAD~3
git diff backup/before-push-rewrite HEAD
git log -5 --oneline
git push \
 --force-with-lease=refs/heads/feature/history-cleanup:$expected
 _oid \
 origin HEAD:refs/heads/feature/history-cleanup
```

커밋 메시지와 묶음만 정리했다면 마지막 파일 상태의 diff는 비어 있어야 한다. 내용도 고쳤다면 예상한 차이만 남아야 한다. 실패한 lease는 원격이 바뀌었다는 신호일 수 있다. 새 OID만 다시 받아 무작정 재시도하지 말고 원격 변경을 읽는다.

## 정리 후 확인

PR의 파일 차이가 의도대로인지 본다. 새 커밋 ID와 CI와 승인 상태를 확인한다. 다른 사람이 이미 그 브랜치를 사용했다면 새 기준으로 정리할 절차를 공유한다.

확인 문제 --force-with-lease가 있으면 팀과 합의할 필요가 있을까?  
원격 상태 검사와 협업 합의는 다른 문제다. 원격을 덮어쓰지 않더라도 동료의 로컬 작업 기준이 달라질 수 있다.

## 16.2 Worktree와 Bisect로 작업과 조사 분리

관련 근거 [R55] [R56]

### 하나의 이력에서 여러 작업 폴더를 쓴다

Worktree는 같은 저장소의 객체를 공유하면서 별도 작업 폴더와 체크아웃 상태를 만든다. 기능 개발을 유지한 채 긴급 수정이나 과거 버전 검증할 때 편하다. 같은 브랜치를 여러 Worktree에 동시에 체크아웃하는 것은 기본적으로 제한된다.

```
git fetch origin
git worktree add ../lesson-hotfix -b fix/urgent origin/main
git worktree list
```

머킹팅 변경은 각 작업 폴더에 남는다. 파일이 별도라고 저장소의 모든 설정과 참조가 독립적인 것은 아니다. 다른 Worktree에서 브랜치를 삭제하거나 이력을 바꾸는 동작의 영향을 이해한다.

## Bisect는 처음 깨진 지점을 좁힌다

정상 커밋과 오류 커밋을 지정하고 중간 상태를 반복 검사한다. 각 커밋이 같은 절차로 검사 가능해야 한다. 환경이 바뀌거나 외부 서비스가 불안정하면 잘못된 판정이 생길 수 있다.

판정	의미
good	그 상태에서는 문제 없음
bad	그 상태에서 문제 재현
skip	검사할 수 없음
reset	조사 종료 후 원래 위치 복귀

Bisect는 깨끗한 별도 폴더에서 한다. 과거 코드가 어떤 설치 스크립트를 실행하는지도 검토한다. 조사 도중 현재의 비밀과 운영 환경을 그대로 연결하지 않는다.

## Worktree의 독립 범위와 공유 범위

### 폴더가 두 개라도 모든 것이 두 벌은 아니다

Worktree마다 작업 파일과 체크아웃 상태가 분리된다. 같은 저장소의 객체와 일부 참조와 설정은 공유할 수 있다. 한쪽에서 한 Git 작업의 영향이 다른 쪽과 전혀 무관하다고 생각하지 않는다.

독립적으로 볼 것	함께 관리할 것
각 폴더의 미커밋 편집	공유하는 브랜치와 객체
각 작업의 실행 파일	원격 주소와 접근 경로
현재 체크아웃 상태	브랜치 정리와 이력 재작성

## 급한 수정을 시작할 때

진행 중인 기능 폴더를 억지로 정리하지 않고 별도 Worktree에서 최신 기준의 수정 브랜치를 준비할 수 있다. 그 폴더에 필요한 설치와 환경 설정은 프로젝트 특성에 맞게 확인한다. 작업 폴더가 새로 생겼다고 실행 환경까지 모두 복제된 것은 아니다.

## Bisect의 good과 bad는 조사하는 문제 기준이다

한 커밋에 다른 경고가 있어도 조사하는 기능이 정상이라면 그 기준을 유지한다. 반대로 설치 실패로 기능을 확인하지 못했다면 곧바로 bad로 분류하지 않는다. 같은 절차로 판정할 수 있는 조건을 먼저 마련한다.

### 조사 후 정리

bisect reset으로 조사를 끝낸 뒤 필요한 수정이나 기록을 보존한다. Worktree를 제거하기 전 그 폴더의 status를 확인한다. 탐색기에서 폴더만 먼저 지웠다면 Git이 알고 있는 Worktree 목록과 상태도 정리해야 할 수 있다.

최초 불량 커밋을 찾은 결과는 원인 조사 출발점이다. 실제 회귀 원인과 수정 방법은 코드와 맥락으로 검증한다.

## 실습 16.2 | 별도 폴더에서 회귀 조사

GOOD-ID는 문제가 없었던 커밋으로 바꾼다. 처음에는 수동으로 브라우저 동작을 확인해 good 또는 bad를 판단한다.

```
git worktree add --detach ../lesson-investigation HEAD
cd ../lesson-investigation
git bisect start
git bisect bad HEAD
git bisect good GOOD-ID
현재 파일을 실행해 보고 하나를 선택
git bisect good
다음 상태를 실행해 보고 good 또는 bad 반복
git bisect reset
```

위 good은 예시 판정이다. 실제로 실패한 상태에는 bad를 사용한다. 결과로 나온 최초 불량 커밋의 diff와 관련 PR을 읽고 원인을 확인한다. 자동화하려면 성공 때 0과 실패 때 비영 종료로 반환하는 신뢰할 수 있는 검사 스크립트를 준비한다.

### Worktree 정리

작업을 커밋하거나 별도 보존한 뒤 원래 저장소 폴더에서 git worktree remove 경로를 실행한다. 미커밋 변경이 있는 Worktree를 --force로 지우지 않는다. 폴더를 탐색기에서 먼저 지웠다면 list와 prune의 의미를 확인한다.

확인 문제 처음 불량 커밋을 찾았으면 작성자의 실수가 확정될까?  
문제의 최초 관찰 지점일 뿐이다. 환경과 이전 잠재 결함과 통합 맥락을 함께 읽어 실제 원인을 검증해야 한다.

## 16.3 Bisect 자동화와 Rebase 전후 검토

관련 근거 [R56] [R82]

### 검사를 자동으로 판정할 수 있어야 한다

git bisect run은 스크립트의 종료 코드로 정상과 오류를 판정한다. 보통 0은 good이며 1부터 127 중 125를 제외한 값은 bad다. 125는 검사 불가능한 상태를 건너뛰게 한다. 테스트 도구 설치 실패를 기능 오류와 섞으면 잘못된 커밋을 가리킬 수 있다.

```
별도 조사 Worktree 안에서 실행
git bisect start
git bisect bad BAD-ID
git bisect good GOOD-ID
git bisect run python /ABSOLUTE/PATH/check_regression.py
git bisect reset
```

check_regression.py는 조사 중인 저장소 밖에 보관하면 과거 커밋 이동으로 사라지지 않는다. 검사 대상 파일은 현재 Worktree에서 읽는다. 외부 네트워크와 현재 시각에 의존하는 검사는 먼저 안정화한다.

### Range-diff는 변경 묶음의 변화를 본다

Rebase 전후 커밋 열을 비교할 때 최종 파일 diff만으로 부족할 수 있다. Range-diff는 두 커밋 범위의 패치를 대응시켜 변경 의도가 어떻게 달라졌는지 읽는 데 도움을 준다. 완전한 의미 동등성 증명은 아니다.

자동 Bisect 전에 알려진 정상과 오류 상태에 검사 스크립트를 각각 실행해 판정이 맞는지 확인한다.

### Bisect와 range-diff를 조사 목적에 맞게 사용

#### 두 도구는 다른 질문에 답한다

Bisect는 좋은 상태와 나쁜 상태 사이에서 문제를 도입한 변경을 좁힌다. range-diff는 두 커밋 열의 패치가 어떻게 달라졌는지 검토한다. 하나는 원인 후보 탐색이며 다른 하나는 재작성 전후 비교다.

상황	먼저 준비할 것
Bisect 수동 판정	반복 가능한 재현 절차
Bisect 자동 판정	종료 코드가 분명한 검사 스크립트

## 상황

## 먼저 준비할 것

실행 불가 커밋

skip 기준과 조사 기록

range-diff

비교할 두 커밋 범위

### 종료 코드가 조사 결과를 좌우

bisect run에서 0은 좋은 상태로 판정한다. 1부터 127까지는 125를 제외하고 나쁜 상태로 판정한다. 125는 해당 커밋을 건너뛴다. 네트워크 장애나 설치 실패를 기능 버그와 같은 실패로 처리하면 엉뚱한 변경을 원인으로 지목할 수 있다.

### 범위를 줄이기 전에 같은 검사를 유지

처음의 good와 bad를 동일한 조건에서 재현한다. 환경 버전과 입력과 의존성을 기록한다. 커밋마다 검사 기준이 달라지면 이분 탐색의 전제가 흔들린다. 간헐 오류라면 먼저 안정적으로 판정할 방법을 마련한다.

### 후보를 찾은 뒤 검증

문제로 지목된 커밋과 그 직전 상태에서 같은 검사를 다시 수행한다. 변경 내용을 읽고 재현 결과와 연결한다. 최초 실패 지점은 조사 근거이며 누가 잘못했는지 단정하는 근거로만 사용하지 않는다. 조사 후에는 bisect reset으로 원래 작업 흐름으로 돌아간다.

range-diff 결과는 기능 검사를 대신하지 않는다. 패치가 비슷하게 보이더라도 새 기준 코드와 합쳐졌을 때의 동작을 확인한다.

### 실습 16.3 | 커밋 열의 전후 비교

OLD-BASE와 OLD-TIP은 재작성 전 범위다. NEW-BASE와 NEW-TIP은 재작성 후 범위다. Rebase 전에 백업 브랜치를 만들어 이전 끝을 보존한다.

```
git branch backup/before-range-review
개인 브랜치의 Rebase 수행 후
git range-diff OLD-BASE..OLD-TIP NEW-BASE..NEW-TIP
git diff OLD-TIP NEW-TIP
```

Range-diff는 커밋별 패치의 대응을 읽고 diff는 두 끝의 파일 상태 차이를 읽는다. 두 출력을 함께 검토하면 메시지 정리와 기능 변경이 섞였는지 알아보기 쉽다.

### 회귀 검사 스크립트의 계약

상태	반환할 결과
기능이 정상	0
조사하는 오류가 재현	1 등 명확한 실패 코드
그 커밋은 실행 불가	125
검사 자체가 잘못 구성됨	조사 중단 후 수정

테스트가 한 번 실패했다고 모두 bad로 분류하면 안 된다. 불안정한 네트워크와 버전 설치 실패는 기능의 회귀가 아닐 수 있다. 최초 불량 커밋을 찾은 뒤 관련 코드와 PR을 읽어 원인을 검증한다.

확인 문제 Range-diff에 큰 차이가 없으면 배포 검증을 생략할 수 있을까?

패치 비교는 검토 근거 중 하나다. 실제 실행 환경과 의존성과 관련한 문제는 별도 검증이 필요하다.

## 17장 CLI와 API를 활용한 업무 연동

명령과 요청의 대상을 명시하고 필요한 결과를 수집한다. REST와 GraphQL과 Webhook의 응답과 페이지 이동과 서명을 검증한다.

### API는 화면의 복사본이 아니라 명시적인 계약이다

REST는 자원과 요청 방식 및 응답 상태를 중심으로 읽는다. GraphQL은 필요한 필드를 질의로 표현한다. 어느 방식이든 페이지네이션과 권한 및 오류와 제한을 처리해야 한다. 첫 응답에 자료가 100개 있다고 전체가 100개라고 결론 내리면 데이터가 잘릴 수 있다.

GraphQL의 cursor는 다음 페이지를 읽기 위한 불투명한 값으로 다룬다. 내부 형식을 해석하거나 번호처럼 계산하지 않는다. hasNextPage와 endCursor를 사용해 반복하고 중간 실패 뒤 어디서 재개할지 정한다. 수집 중 원본 자료가 바뀔 수 있으므로 결과가 어느 시점의 완전한 스냅샷인지도 검토해야 한다.

Webhook은 사건을 전달하는 신호다. 중복 전달과 순서 변화 및 일시적 실패를 고려해 수신자를 설계한다. 서명 검증은 원본 바이트를 기준으로 수행하고 처리한 전달 ID를 기록해 중복 효과를 줄인다. Webhook이 도착했다고 외부 자료의 모든 최신 상태를 이미 안다고 가정하지 않는다.

### 활동 데이터와 성과를 동일시하지 않는다

Kalliamvakou 등의 GitHub 마이닝 연구는 저장소의 목적과 활동 상태 및 PR 병합 기록의 해석에서 주의할 점을 제시했다. 저장소 데이터가 풍부하더라도 연구 질문에 맞는 표본과 측정 정의가 필요하다는 내용이다. 2014년 자료에서 관찰한 비율을 현재 플랫폼 전체의 비율로 재사용해서는 안 된다. [S14]

팀의 커밋 수를 비교할 때도 같은 주의가 필요하다. Squash 정책을 쓰는 팀과 작은 커밋을 유지하는 팀은 같은 작업을 다른 개수로 기록할 수 있다. 자동 생성 커밋과 봇의 활동도 섞인다. 활동량은 관찰 가능한 행동의 흔적이지 가치와 품질의 완전한 대리 지표가 아니다.

### 자동화의 완료 조건을 부분 실패까지 확장한다

이슈 목록을 내보내는 자동화가 중간 페이지에서 실패하면 이미 저장한 자료와 남은 자료를 구분해야 한다. 중복 없이 재개할 수 있는 키와 진행 기록을 남긴다. 삭제된 자원과 권한 변경으로 더 이상 읽을 수 없는 자원을 단순한 빈 값으로 처리하면 의미가 사라질 수 있다.

자동화는 필요한 권한만 가진 자격 증명으로 실행하고 결과에 수집 범위와 시점을 적는다. 정확한 수치처럼 보이는 표일수록 누락과 필터 및 페이지네이션의 조건을 함께 설명해야 한다. 전문적인 보고서는 숫자의 정밀도보다 그 숫자가 무엇을 대표하는지 분명하게 한다.

## 조회의 반복과 변경의 반복은 다른 비용을 가진다

API 자동화는 응답 코드와 페이지네이션 및 제한 헤더를 계약으로 다뤄야 한다. GitHub는 불필요한 polling을 줄이고 조건부 요청을 활용하며 제한 오류의 안내에 맞춰 재시도하도록 설명한다. 변경 요청을 무분별하게 병렬 실행하는 방식도 제한과 순서 문제를 일으킬 수 있다. 최대 호출 수를 고정값으로 외우기보다 실제 응답과 인증 문맥을 확인한다. [T28]

전체 이슈 목록을 매번 처음부터 내려받는 자동화는 자료가 많아질수록 비용이 커진다. 변경된 범위를 식별하고 중간 진행 상태를 남기는 방법을 검토한다. 다만 마지막 시각 하나만 저장하면 동일 시각의 경계 자료나 지연된 갱신을 놓칠 수 있다. 겹치는 조회 범위와 안정적인 식별자 기반 중복 제거를 함께 사용할 수 있다.

페이지를 읽는 동안 원격 자료가 바뀔 수 있다는 점도 고려한다. 첫 페이지와 마지막 페이지가 한 시점의 완전한 스냅샷이라고 보장되는지 API 계약을 확인한다. 보장이 없다면 수집 시각과 조건을 남기고 정확한 스냅샷이 필요한 분석에는 별도 전략을 선택한다. 수집 성공과 측정 타당성은 같은 판단이 아니다.

## 이벤트 전달과 업무 처리의 완료를 분리한다

Webhook을 받는 서버는 서명을 확인하고 필요한 이벤트와 action을 구분해야 한다. GitHub의 전달 식별자는 중복 처리 방지와 재전달 추적에 사용할 수 있다. 실패한 전달이 자동으로 모두 재전송된다고 가정하지 않으며 누락된 전달을 확인하고 다시 처리하는 경로를 준비한다. [T29]

서버가 성공 응답을 보냈다는 사실은 업무 처리가 끝났다는 뜻이 아닐 수 있다. 오래 걸리는 작업을 큐에 넣는다면 응답은 안전하게 접수했다는 의미로 정의할 수 있다. 이 경우 큐 기록과 실제 처리 결과를 별도로 관리해야 한다. 응답을 먼저 보내고 아무 기록도 남기지 못한 채 프로세스가 종료되면 사건이 사라질 수 있다.

같은 전달을 두 번 받았을 때 이슈 댓글을 두 번 작성하지 않게 하려면 식별자와 처리 결과를 연결한다. 단순히 처리 중이라는 표시만 남긴 뒤 실패하면 영원히 재처리되지 않는 문제도 생길 수 있다. 접수와 처리 중 및 완료와 재시도 가능 실패 같은 상태를 정의하고 상태 전이를 기록한다. 자동화의 신뢰성은 성공 경로보다 중단 이후의 행동에서 드러난다.

상태 변화 / 페이지네이션은 마지막 페이지까지 이어진다

페이지네이션은 마지막 페이지까지 이어진다

1 / 4 첫 응답은 전체 결과일 필요가 없다

요청

첫 페이지

받은 항목

1-100

다음 cursor

c1

누적 상태

100개

숫자는 설명용 가상 데이터다. cursor는 불투명한 값으로 취급하며 중간 실패 후 재개와 중복 처리도 설계한다.

페이지네이션은 마지막 페이지까지 이어진다

4 / 4 수집 시점과 권한 및 필터도 기록한다

요청

종료

받은 항목

범위 확인

다음 cursor

종료

누적 상태

중복과 누락 검사

숫자는 설명용 가상 데이터다. cursor는 불투명한 값으로 취급하며 중간 실패 후 재개와 중복 처리도 설계한다.

숫자는 설명용 가상 데이터다. cursor는 불투명한 값으로 취급하며 중간 실패 후 재개와 중복 처리도 설계한다.

- 1. 첫 응답은 전체 결과일 필요가 없다
- 2. 서버가 제공한 다음 위치를 사용한다
- 3. 다음 페이지가 없는지 확인한다
- 4. 수집 시점과 권한 및 필터도 기록한다

17.1 GitHub CLI와 검색을 업무에 활용

관련 근거 [R09] [R50] [R51]

## Git은 이력을 다루고 gh는 GitHub를 다룬다

git commit은 로컬 이력을 만든다. gh pr create는 GitHub의 PR을 만든다. gh는 Issue와 PR과 Actions와 Release를 명령으로 관리한다. 반복 업무에서 창 이동을 줄이지만 변경 대상과 인증 계정을 항상 확인한다.

```
gh auth status
gh repo view
gh issue list --state open
gh pr list --state open
gh run list --limit 10
```

검색 입력	찾는 대상
repo:YOUR-ID/lesson-lab is:issue is:open	열린 Issue
repo:YOUR-ID/lesson-lab is:pr review-requested:@me	내 검토 요청
repo:YOUR-ID/lesson-lab is:pr is:merged	병합한 PR
repo:YOUR-ID/lesson-lab label:bug	오류로 분류한 작업

## 구조화된 결과로 반복 작업을 줄인다

명령의 --json 옵션으로 선택 필드를 받을 수 있다. 화면의 표를 복사해 분석하기보다 필요한 필드만 추출한다. 자동화가 쓰기 작업을 한다면 dry run과 대상 수 제한과 실패 기록을 설계한다.

터미널 기록과 CI 로그에 토큰을 직접 입력하지 않는다. gh auth token처럼 비밀을 출력하는 명령은 일반 확인용으로 쓰지 않는다.

## GitHub CLI의 대상과 출력 형식을 명시

### 현재 폴더가 기본 대상에 영향을 준다

gh가 어느 저장소를 보고 있는지 확인하지 않고 이슈나 PR을 만들면 다른 곳에 기록할 수 있다. git remote -v와 gh repo view로 대상을 읽는다. 반복 자동화에서는 -R OWNER/REPO 같은 대상 지정도 검토한다.

행동	로컬과 원격의 변화
git commit	로컬 Git 이력 변경
git push	원격 Git 참조 갱신
gh issue list	GitHub 작업 목록 조회
gh pr create	GitHub PR 생성
gh run view	Actions 실행 기록 조회

## 본문 파일을 쓰는 이유

여러 줄의 설명을 셀 인자 안에 길게 넣으면 줄바꿈과 따옴표 해석이 꼬일 수 있다. 파일에 원하는 본문을 쓰고 --body-file로 전달하면 실제 내용을 검토하기 쉽다. 임시 본문 파일을 소스에 포함할지 여부도 정한다.

## --json과 --jq를 구분한다

--json은 필요한 필드를 구조화해 받는다. --jq는 그 구조에서 골라 표시하거나 변형하는데 쓴다. 프로그램이 사용할 결과를 화면 표에서 위치로 잘라 읽기보다 명시적인 필드를 사용하는 편이 안정적이다.

## 쓰기 자동화로 확장한다면

처음에는 목록을 읽어 대상이 맞는지 확인한다. 바꿀 항목 수와 조건을 출력하는 검토 단계를 둔다. 일부 실패와 중복 실행을 어떻게 처리할지도 설계한다. 조회 명령의 성공을 대량 변경 허가로 해석하지 않는다.

gh auth status는 인증 확인에 쓰지만 토큰을 출력하는 다른 명령은 비밀을 노출할 수 있다. 문제 해결에 필요한 최소 정보만 공유한다.

## 실습 17.1 | 파일을 이용해 PR 본문 만들기

편집기로 pr-body.md를 만들고 문제와 결과와 검증을 적는다. 작업 브랜치를 Push한 뒤 실행한다. 공개 활동을 만드는 명령이므로 실제 대상 저장소를 먼저 확인한다.

```
git remote -v
git branch --show-current
gh repo view
git push -u origin HEAD
gh pr create --base main --title 'Improve answer access' \
```

```
--body-file pr-body.md --draft
gh pr view --web
```

## 검토와 검사 조회

```
gh pr checks
gh pr diff
gh run list --limit 5
gh run view RUN-ID --log-failed
gh issue list --state open --json number,title,url
```

RUN-ID는 run list에서 확인한 실제 실행 ID로 바꾼다. PR 본문 파일이 불필요하게 커밋되지 않도록 추적 여부를 확인한다. CLI 출력의 심표와 따옴표는 프로그램 문법이므로 그대로 사용한다.

확인 문제 gh repo view가 다른 저장소를 보여 주면 그대로 PR을 만들어도 될까?  
현재 폴더의 원격 연결과 명령의 대상 지정부터 바로잡는다. gh의 -R OWNER/REPO로 대상을 명시할 수도 있다.

## 17.2 REST와 GraphQL과 Webhook 연동

관련 근거 [R52] [R53] [R54] [R71]

### 조회와 변경과 이벤트 수신을 구분한다

REST는 경로별 자원을 다루고 GraphQL은 필요한 필드를 질의한다. Webhook은 GitHub의 사건을 내 서버로 전달한다. 통합이 여러 저장소에 오래 접근해야 한다면 개인 PAT보다 설치 범위와 권한과 수명을 관리하기 쉬운 GitHub App을 검토한다.

설계 문제	원칙
결과가 많음	페이지네이션 처리
요청 제한	헤더와 재시도 시각 확인
일시 오류	지수 백오프와 재시도 한도
중복 Webhook	delivery ID로 중복 처리 방지
외부 입력	서명 검증 후 스키마 검증

```
gh api --paginate \
 'repos/YOUR-ID/lesson-lab/issues?state=open&per_page=100' \
 --jq '.[].number'
```

Issue 목록 API에는 PR 항목도 섞일 수 있다. 분석 전에 pull_request 필드가 있는 항목을 구분한다. 페이지 하나를 읽고 전체라고 보고하지 않는다.

## Webhook의 신뢰 경계

X-Hub-Signature-256의 HMAC 서명을 원본 요청 바이트로 검증한다. JSON을 다시 직렬화한 값으로 계산하면 안 된다. 서명이 맞아도 허용한 이벤트와 저장소인지 검사하고 본처리는 큐로 넘기는 편이 운영에 유리하다.

서명 확인은 중복 방지나 재생 공격 방지를 자동으로 완성하지 않는다. delivery ID와 처리 기록과 재시도 정책을 별도로 둔다.

## REST 조회와 Webhook 수신의 실패를 다르게 다루기

### 목록의 일부를 전체 결과로 보고하지 않는다

REST 응답이 페이지로 나뉘면 다음 페이지 표시와 끝 조건을 따라가야 한다. 첫 응답이 정상이라는 사실과 모든 데이터를 받았다는 사실은 다르다. 중간 요청이 실패했다면 불완전한 결과로 표시하거나 다시 이어갈 기준을 남긴다.

### Webhook은 중복 도착을 고려한다

서명 검증을 통과한 전달이라도 같은 사건이 다시 도착할 수 있다. 전달 식별자와 처리 기록으로 같은 일을 두 번 실행하지 않게 한다. 처리 시작과 완료를 어떻게 기록할지와 실패 후 재시도 경로를 함께 정한다.

검사	확인하는 질문
HMAC 서명	예상 비밀로 검증되는 원본인가
이벤트 종류	이 서버가 처리할 사건인가
저장소와 브랜치	허용한 자원에서 왔는가
중복 기록	이미 처리한 전달인가
동작 권한	요청한 후속 행동을 허용하는가

## 서명 전에 JSON을 다시 만들면 안 되는 이유

HMAC은 바이트를 대상으로 계산한다. 공백과 줄바꿈과 문자 표현이 바뀌면 JSON의 의미가 같아도 바이트가 달라진다. 원본 요청 본문을 먼저 검증한 뒤 필요한 파싱과 스키마 검사를 진행한다.

## 긴 작업은 수신과 분리한다

수신 요청 안에서 모든 빌드와 배포를 오래 수행하면 시간 제한과 중복 재시도가 얹히기 쉽다. 검증 후 큐에 넣고 후속 작업이 처리 상태를 관리하는 구조를 검토한다. 큐를 쓴다고 자동으로 중복 문제가 사라지는 것은 아니다.

전달의 진위를 확인하는 일과 배포를 허용하는 일은 서로 다른 판단이다.

## 실습 17.2 | 서명 확인 함수 로컬 검증

아래 함수는 웹 서버 전체가 아니다. 요청의 raw body와 서명 헤더와 안전하게 보관한 공유 비밀을 전달하는 검증 부분이다. 실제 서버는 HTTPS와 요청 크기 제한과 이벤트 검증도 필요하다.

```
import hashlib
import hmac

def valid_signature(raw_body, signature, secret):
 if not signature or not signature.startswith('sha256='):
 return False
 digest = hmac.new(
 secret.encode('utf-8'),
 raw_body,
 hashlib.sha256
).hexdigest()
 expected = 'sha256=' + digest
 return hmac.compare_digest(expected, signature)

payload = b'{'
demo_secret = 'EXAMPLE-ONLY'
signed = 'sha256=' + hmac.new(
 demo_secret.encode(), payload, hashlib.sha256
).hexdigest()
assert valid_signature(payload, signed, demo_secret)
assert not valid_signature(b'{}', signed, demo_secret)
```

두 번째 검사는 JSON 의미가 같아도 바이트가 다르면 서명이 맞지 않음을 보여 준다. 데모 비밀을 실제 서비스에서 사용하지 않는다.

확인 문제 서명이 맞는 Webhook은 바로 운영 배포를 실행해도 될까?  
허용 저장소와 이벤트와 브랜치와 승인 정책을 추가로 검사한다. 진짜 전달인지와 그 동작을 허용할지는 별개의 판단이다.

## 17.3 GraphQL로 필요한 데이터만 가져오기

관련 근거 [R71] [R52] [R87]

### 필드 선택과 페이지 진행을 함께 설계한다

GraphQL은 필요한 필드를 고르는 API다. 적은 필드를 요청한다고 무제한 호출이 가능한 것은 아니다. 연결의 first와 after와 pageInfo를 사용해 다음 페이지를 진행하고 요청 비용과 제한 상태를 함께 확인한다.

```
query($owner: String!, $name: String!, $cursor: String) {
 repository(owner: $owner, name: $name) {
 issues(first: 50, after: $cursor, states: OPEN) {
 nodes { number title url }
 pageInfo { hasNextPage endCursor }
 }
 }
 rateLimit { remaining resetAt }
}
```

위 질의를 query.graphql로 저장한다. 이름이 같은 필드라도 REST와 반환 구조가 다르다. GraphQL 응답에 errors가 있는지와 data가 일부만 있는지도 읽어야 한다. HTTP 상태만 성공이라고 전체 조회가 완성됐다고 판단하지 않는다.

필드	읽는 뜻
nodes	이번 페이지의 작업 목록
hasNextPage	다음 페이지가 있는가
endCursor	다음 요청에서 사용할 위치
remaining	남은 API 사용 여유
resetAt	제한 상태가 재설정될 시각

## GraphQL 응답을 데이터 구조로 읽기

### 필요한 필드를 정해 요청한다

GraphQL은 필요한 필드를 요청하고 그 구조에 맞는 응답을 읽는다. REST보다 항상 간단하거나 저렴한 것은 아니다. 연결된 데이터와 페이지 이동과 조회 비용을 함께 설계한다.

응답 요소	읽는 의미
data	요청한 데이터의 결과
errors	처리 과정의 오류 정보
nodes 또는 edges	연결 안의 항목
pageInfo.hasNextPage	다음 페이지 존재 여부
pageInfo.endCursor	다음 요청의 기준 커서

### 성공 상태만으로 끝내지 않는다

HTTP 상태가 성공이어도 errors가 있는지 확인한다. 일부 필드만 실패할 수 있으므로 data가 있다는 이유로 모든 결과가 완전하다고 판단하지 않는다. null 값도 실제 부재인지와 접근 권한이나 오류 영향인지 문맥을 읽는다.

### 페이지 이동을 이어서 한다

첫 페이지의 endCursor를 다음 요청의 after에 전달한다. hasNextPage가 false가 될 때까지 반복한다. 매번 같은 커서를 다시 쓰지 않도록 기록한다. 조사 도중 데이터가 바뀔 수 있으므로 장시간 수집에서는 중복과 누락을 어떻게 다룰지도 정한다.

### 파일에서 쿼리를 읽는 이유

복잡한 쿼리를 셸 문자열 안에 모두 넣으면 따옴표와 변수 확장 문제가 섞일 수 있다. 별도 query.graphql 파일로 관리하면 요청 구조를 검토하기 쉽다. 본문의 gh api graphql 예제는 파일 내용을 읽는 -F 형식을 사용한다.

응답과 로그를 보관할 때 비공개 저장소 이름과 사용자 정보가 포함되는지 확인한다. 조회 권한이 있다고 모든 결과를 공개할 권한까지 생기지는 않는다.

## 실습 17.3 | gh로 GraphQL 질의 실행

아래는 파일의 질의를 gh의 query 필드로 전달한다. 실제 저장소를 읽을 수 있는 인증이 필요하다. 질의 변수 owner는 본인 계정으로 바꾼다.

```
gh api graphql \
 -F owner=YOUR-ID \
 -F name=lesson-lab \
 -F query=@query.graphql
```

첫 응답의 hasNextPage가 true라면 endCursor를 다음 호출의 cursor 변수로 전달해야 한다. gh api의 파일 필드 입력은 @파일이름 문법을 지원한다. 환경에 따라 질의를 셸 문자열로 넣는 대신 --input JSON 등 공식 방식도 사용할 수 있다.

### 분석 결과를 과장하지 않는다

조회 시각과 저장소와 필터와 페이지 범위를 기록한다. PR 숫자를 작업 성과와 동일시하지 않는다. 텍스트와 URL을 외부 문서로 옮길 때 비공개 저장소의 내용이 새 문서의 접근 범위에 맞는지 확인한다.

### 처리 안정성

응답의 오류와 제한을 기록하고 재시도 한도를 둔다. 일부 페이지를 읽은 뒤 실패하면 결과가 불완전하다고 표시한다. 숫자만 저장하기보다 어떤 조건에서 얻었는지 함께 남긴다.

확인 문제 첫 페이지의 목록이 짧아 보이면 전체라고 보고해도 될까?

API의 다음 페이지 표시와 조회 조건을 확인해야 한다. 인상이나 화면 길이로 전체 조회 여부를 판단하지 않는다.

## 18장 종합 프로젝트와 교육 자료 운영

개인 작업과 팀 검토를 Issue부터 배포까지 이어 간다. 교육 자료의 수식과 도식과 판본 및 접근 범위를 관리한다.

### 기능 완료와 운영 준비를 함께 판단한다

lesson-lab의 종합 프로젝트는 저장소 생성에서 끝나지 않는다. 변경 요구를 이슈로 설명하고 브랜치에서 구현하며 PR로 검토하고 CI로 검사한 뒤 판본을 식별해 배포한다. 문제가 생겼을 때 이전 결과와 데이터를 복구할 수 있는지도 확인한다. 각각의 단계는 별도 기능이면서 앞 단계의 주장에 대한 증거를 제공한다.

예를 들어 자료 검색을 개선했다는 주장은 변경된 코드와 테스트 및 배포 결과가 함께 뒷받침해야 한다. 로컬 테스트만 통과했다면 배포 경로의 문제는 아직 확인하지 않은 것이다. CI가 통과했어도 실제 사용자의 키보드 조작과 내용의 정확성은 별도 검토가 필요할 수 있다.

### 재현 가능한 산출물에는 실행 조건이 남는다

교육 자료와 연구 코드를 GitHub로 공유할 때는 결과만 올리지 않는다. 입력 데이터의 출처와 처리 과정 및 실행 환경을 설명한다. 공개할 수 없는 자료는 가짜 데이터나 접근 절차로 대체하되 그로 인해 재현 범위가 달라진다는 점을 표시한다. 계산 결과가 같다는 의미의 재현성과 연구 해석이 타당하다는 의미의 검증은 구분한다.

수식과 도식은 버전 관리할 원본과 배포할 결과를 함께 생각한다. 소스와 생성 이미지를 모두 보관한다면 어느 쪽이 권위 있는 원본인지 적는다. 자동 생성 결과를 사람이 직접 수정하면 다음 빌드에서 사라질 수 있다. 문서 생성 명령과 확인 절차를 기록하면 유지보수자가 같은 결과를 다시 만들 수 있다.

### 최종 검토는 질문에 증거로 답하는 과정이다

저장소의 목적을 처음 보는 사람이 이해할 수 있는가. 필요한 권한과 비밀이 문서화되어 있는가. 현재 배포가 어느 커밋에서 나왔는가. 중요한 실패를 어떤 검사로 발견하는가. 담당자가 바뀌어도 복원할 수 있는가. 이 질문에 실제 파일과 실행 결과 및 설정으로 답할 수 있어야 한다.

학술 자료는 이러한 판단의 배경을 제공한다. 하지만 논문을 인용했다고 특정 저장소가 자동으로 좋은 설계를 갖는 것은 아니다. 연구에서 제시한 개념을 자신의 환경에

적용하고 관찰한 결과와 남은 한계를 설명해야 한다. 이 책의 마지막 점검표는 그 과정을 반복 가능하게 만드는 도구다.

## 주장과 증거와 남은 가정을 연결한다

프로젝트를 마칠 때 모든 기능이 완성되었다는 문장 하나로 보고하면 검토자는 의미를 확인하기 어렵다. 요구별로 구현한 판본과 수행한 검사 및 관찰한 결과를 연결한다. 같은 자료에서 다른 사람이 검사를 반복할 수 있는지도 본다. 증거가 있는 주장과 아직 가정에 의존하는 부분을 구분하면 다음 수정의 우선순위가 드러난다.

주장	적절한 증거	그 증거가 보장하지 않는 범위
검색 규칙이 맞다	명세와 경계 입력의 실제 결과	모든 브라우저의 표시 품질
배포 판본을 안다	커밋과 산출물 다이제스트 및 배포 기록	소스의 모든 보안 속성
복원할 수 있다	분리 환경에서의 복원 실행 기록	아직 보존하지 않은 외부 자산
협업 절차가 적용된다	실제 PR의 승인과 검사 및 거부 결과	모든 예외 경로의 적절성

이 표는 서류를 늘리기 위한 양식이 아니다. 검사 결과가 어떤 결정을 허용하는지 읽는 방법이다. 검색 함수 검사만으로 운영 계정의 안전성을 주장할 수 없고 서명 검증만으로 교육 내용의 정확성을 주장할 수 없다. 서로 다른 품질 속성에는 서로 다른 증거가 필요하다.

## 전문성은 변경을 예측하고 한계를 설명하는 능력이다

초보자는 현재 명령의 성공 여부에 집중하기 쉽다. 경험이 쌓이면 그 명령이 어느 상태를 바꾸며 다른 사람의 작업과 어떤 관계를 갖는지 생각한다. 더 나아가 실패했을 때 남은 증거와 복구 경로 및 운영 비용을 함께 설계한다. 이 발전은 사용하는 명령의 개수보다 판단하는 범위의 확장으로 볼 수 있다.

마지막으로 작은 변경을 다른 사람에게 맡겨 본다. 자료 한 개를 추가하고 검사하며 PR을 만들고 배포 판본을 확인하게 한다. 설명이 막히는 곳은 프로젝트의 지식이

개인에게 묶여 있는 지점일 수 있다. 그 지점을 코드와 문서 및 자동화의 계약으로 옮기는 작업이 장기적인 유지보수 능력을 만든다.

## 18.1 개인 종합 실습

관련 근거 [R01] [R30] [R37]

### HTML 수업 도구를 변경 기록과 함께 배포한다

한 저장소에서 기초 흐름을 끝까지 연결한다. 작은 수학 퀴즈나 안내 페이지를 만들되 실제 학생 데이터는 받지 않는다. 출발 기능은 10.1절의 정답 버튼이면 충분하다. 새 기능을 많이 만드는 것보다 변경의 근거가 이어지는지가 중요하다.

단계	만들어야 할 증거
요구	완료 조건이 있는 Issue
구현	작업 브랜치와 목적별 커밋
검토	설명과 검증이 있는 PR
검사	성공과 실패가 확인된 CI
공개	직접 동작 확인한 Pages 주소
판본	태그와 Release 설명

### 실습 과제

정답을 열고 다시 닫는 버튼을 만든다. 버튼의 상태가 화면과 접근성 정보에 맞게 바뀌어야 한다. 작은 화면에서도 내용을 읽을 수 있어야 한다. 실행 방법과 알려진 한계를 README에 적는다.

저장소 공개와 Pages 공개 전에 코드와 이력과 그림과 문서에서 비밀과 개인정보와 저작권 문제를 확인한다. 연습용 가상 내용만 사용하면 부담을 줄일 수 있다.

## 종합 실습에서 증거를 이어 붙이는 방법

### 기능 하나를 끝까지 연결한다

정답 열기와 닫기 기능을 대상으로 Issue부터 Release까지 진행한다. 기능 수를 늘리는 것보다 같은 요구가 코드와 검사와 배포에서 이어지는지 확인하는 것이 이 실습의 목적이다.

단계	남길 구체적인 기록
Issue	작은 화면과 키보드의 완료 조건
구현	작업 브랜치와 목적별 커밋
PR	전후 결과와 실제 검증 방법
CI	성공한 검사와 실패 실험의 원인
배포	커밋과 주소와 실제 버튼 동작
Release	판본과 알려진 한계

### 자기 검토의 순서

요구를 다시 읽는다. Files changed에서 의도 밖의 파일을 찾는다. 실행 조건을 바꿔 작은 화면과 키보드 조작을 확인한다. 자동 검사가 확인하지 못한 범위도 적는다. 합친 뒤에는 배포 주소에서 한 번 더 같은 기능을 확인한다.

### 소스와 사이트가 다르면

어느 커밋이 배포됐는지부터 찾는다. 다른 브랜치나 옛 배포를 보고 있을 수 있다. 결과 폴더와 경로와 캐시도 확인한다. 코드가 맞다는 이유로 배포 문제를 사용자의 조작 실수라고 단정하지 않는다.

### 완성한 뒤 다음 작업 고르기

이번 과정에서 실제로 막혔던 한 가지를 고른다. 실행 안내 부족인지 검사 범위인지 검토 대기인지 분류한다. 그 원인을 줄이는 다음 Issue를 만들면 학습이 실제 개선으로 연결된다.

제출 주소만 모으지 않는다. 요구와 변경과 검증과 실행 버전의 연결을 모으면 나중에 다시 설명하고 수정하기 쉽다.

### 종합 실습 18.1 | 제출물이 아닌 작업 증거 완성

1. lesson-lab에 정답 다시 닫기 Issue를 만든다.
2. feature/answer-toggle 브랜치에서 구현한다.
3. 마우스와 키보드와 작은 화면에서 확인한다.
4. 검사 스크립트를 기능에 맞게 보완한다.
5. PR의 Files changed와 검사 로그를 읽는다.
6. 병합 후 Pages에서 같은 기능을 다시 확인한다.
7. 새 버전 태그와 릴리스 설명을 작성한다.

### 자기 점검 질문

질문	통과 근거
무엇을 해결했는가	Issue의 기대 결과
무엇이 바뀌었는가	PR의 diff
정상인가	실행 방법과 실제 결과
어디서 쓰는가	배포 주소와 버전
실패하면 어떻게 하는가	Revert와 재배포 계획

### 복구 훈련

연습 브랜치에서 안내 문장을 잘못 고친 커밋을 만든다. Revert로 취소하고 새 기록이 남는지 확인한다. 운영 배포를 망가뜨리는 대신 로컬 또는 별도 실습 브랜치에서 절차를 익힌다.

확인 문제 소스와 배포 화면이 다르면 무엇부터 볼까?  
배포한 커밋과 브랜치와 빌드 경로를 확인한다. 최신 Push의 배포 완료 여부와 브라우저 캐시도 본다. 소스 수정 성공과 배포 성공을 나눠 진단한다.

## 18.2 팀 종합 실습과 전문가의 판단

관련 근거 [R28] [R43] [R62]

전문가 수준은 위험과 근거를 함께 설명하는 능력이다

명령어를 많이 외우는 것만으로 팀 운영이 안정되지는 않는다. 요구를 작은 변경으로 나누고 검증할 근거를 만들며 실패했을 때 복구할 수 있어야 한다. 자동화에 부여한 권한과 외부 입력의 경로를 설명할 수 있어야 한다.

역할	책임
요청자	사용 상황과 완료 조건
작성자	작은 변경과 구현 검증
검토자	의도와 영향과 실패 조건 확인
배포 담당	버전과 승인과 복구
운영 담당	권한과 비용과 사고 대응

작은 팀에서는 한 사람이 여러 역할을 맡을 수 있다. 그래도 판단의 단계는 구분한다. 사람이 부족하면 형식적인 자기 승인보다 자동 검사와 변경 크기 제한과 동료 교차 검토 가능 시간을 현실적으로 설계한다.

전문가가 확인할 여섯 가지

어떤 자원이 변하는가. 누가 그 변화를 승인할 수 있는가. 입력을 믿어도 되는가. 어떤 검사가 결과를 뒷받침하는가. 실패하면 어디까지 되돌릴 수 있는가. 운영자가 바뀌어도 같은 절차를 재현할 수 있는가.

이 책의 전문가 장은 운영 판단의 출발점이다. 조직의 실제 시스템에는 서비스별 권한 정책과 데이터 복구와 관측과 규정이 추가로 필요하다. 읽기만 한 항목과 직접 수행한 항목을 구분한다.

팀 실습을 역할 교대와 복구까지 확장

작성자가 바뀌어도 진행되는가

첫 사람이 구현하고 둘째 사람이 검토한 뒤 역할을 바꿔 다른 작은 변경을 진행한다. 이전 작성자가 옆에서 모든 클릭을 대신해 주어야 한다면 안내와 권한과 실행 조건 중 빠진 부분을 찾는다.

역할 교대 후 관찰	보완할 가능성
설치 방법을 다시 묻는다	README와 환경 조건
PR 방향을 틀린다	브랜치와 Base 안내
검증 방법이 사람마다 다르다	완료 조건과 실행 절차
배포 권한을 찾지 못한다	역할과 연결 자원 기록
복구를 한 사람만 안다	복원 문서와 실제 시험

### 전문가의 설명을 평가하는 질문

무엇이 바뀌는지 설명할 수 있는가. 변경을 허용하는 권한을 찾을 수 있는가. 어떤 입력을 믿지 않아야 하는가. 실패를 알아차릴 근거가 있는가. 어떤 상태로 되돌아가야 하는가. 담당자가 바뀌어도 같은 결과를 얻는가.

### 형식적인 절차를 줄인다

모든 작은 수정에 동일한 긴 보고를 요구하면 중요한 정보가 묻힐 수 있다. 단순 문서 수정과 인증 변경에 필요한 검증은 다르다. 팀이 감당할 수 있는 검토 단계를 두되 민감한 변경의 근거를 생략하지 않는다.

### 실습 종료 회고

가장 많은 재질문이 생긴 지점과 가장 오래 대기한 지점을 적는다. 도구를 하나 더 도입하기 전에 그 원인이 불명확한 요구인지 권한인지 자동 검사 부족인지 확인한다. 개선 결과를 다음 작은 작업에서 다시 관찰한다.

성숙도를 커밋 수와 자동화 수로만 판단하지 않는다. 변경을 이해하고 검증하고 복구할 수 있는 능력을 함께 본다.

### 종합 실습 18.2 | 두 사람이 한 변경을 끝내기

1. 요청자가 Issue와 완료 조건을 적는다.
2. 작성자가 브랜치에서 구현하고 Draft PR을 연다.
3. 검토자가 재현과 권한과 실패 조건을 확인한다.
4. CI를 통과시키고 보호 규칙에 맞게 병합한다.
5. 배포 담당이 실제 주소에서 버전을 확인한다.

- 6. 운영 담당이 권한과 비용과 복구 문서를 확인한다.
- 7. 역할을 바꿔 다른 사람이 같은 절차를 반복한다.

최종 역량 평가표

수준	직접 보여 줄 행동
입문	웹에서 브랜치와 PR 만들기
초급	Clone과 커밋과 Push
중급	충돌 해결과 Revert와 리뷰
고급	보호 규칙과 CI와 Release
전문가	신뢰 경계와 이력 재작성과 복구 검증

다음 학습을 정하는 방법

막힌 상황을 Issue로 기록한다. 가장 자주 반복되는 수작업을 찾는다. 그 작업의 입력과 권한과 실패 결과를 먼저 정리한다. 자동화는 그 뒤 도입한다. 한꺼번에 모든 기능을 켜는 대신 측정 가능한 개선 하나씩 쌓는다.

확인 문제 자동화가 많을수록 항상 더 성숙한 팀일까?

자동화의 수보다 실패를 발견하는 능력과 권한 통제와 복구 가능성이 중요하다. 사용하지 않는 자동화와 불안정한 검사는 오히려 운영 부담을 늘릴 수 있다.

18.3 수식과 도식과 교육 자료를 GitHub로 관리

관련 근거 [R84] [R85] [R86]

설명 원본과 배포 파일을 연결한다

수업 자료는 PDF만 보관하면 세부 수정 비교가 어려울 수 있다. Markdown이나 HTML 원본과 그림과 배포판을 함께 설계한다. 개인 학습 기록과 학생 제출물은 공개할 수 있는 안내 자료와 접근 범위를 나눈다.

## Markdown에 수식 쓰기

아래 예시는 GitHub Markdown에서 수식을 표시하는 원본 문법이다. 수식 렌더링은 GitHub가 지원하는 문서 문맥에서 확인한다. 임의의 로컬 HTML에 이 코드를 넣는 것만으로 렌더링 엔진이 생기지는 않는다.

한 변의 길이가  $a$ 인 정사각형의 넓이는  $a^2$ 다.

```
$$
V = \frac{1}{3}Bh
$$
```

## Mermaid로 흐름 설명

아래 내용을 Markdown의 mermaid 코드 구역으로 넣는다.

```
flowchart TD
 A[문제 제시] --> B[개인 탐구]
 B --> C[추가 근거가 필요한가]
 C -->|예| B
 C -->|아니오| D[설명 공유]
```

학생 이름과 점수와 연락처를 예제 저장소에 넣지 않는다. 가상 자료로 구조부터 검증한다.

## 교육 자료를 버전과 배포 방식으로 관리

### 원본과 학생이 보는 결과를 연결한다

Markdown 원본과 렌더링 결과와 PDF 출력은 서로 다른 표현이다. 수식과 다이어그램이 웹에서 잘 보여도 다른 변환 도구에서는 결과가 달라질 수 있다. 실제 학생이 사용할 화면과 파일에서 확인한다.

자료 종류	완료 기준
수식	의도한 기호와 줄바꿈이 보이는가
Mermaid	관계와 화살표가 의미와 맞는가
코드 예제	복사해 실행할 조건을 설명했는가
PDF	한글과 그림과 링크가 읽히는가
과제 저장소	접근 범위와 제출 절차가 맞는가

## 판본을 고정해 수업한다

수업에 사용할 커밋이나 Tag를 기록한다. 수업 중 main이 바뀌면 학생이 본 안내와 교사가 보는 안내가 달라질 수 있다. 수정이 필요한 경우 바뀐 부분과 적용 판본을 알리고 기존 제출에 미치는 영향을 판단한다.

## 과제와 개인정보의 경계

실명과 성적과 민감한 답변을 공개 저장소에 넣지 않는다. 과제 저장소의 공개 여부와 접근 주체를 확인한다. GitHub Classroom을 사용하더라도 조직 정책과 학교의 자료 처리 기준을 함께 따른다. 학생 계정과 결제 주체를 임의로 같은 것으로 가정하지 않는다.

## 자료 개선을 작은 변경으로 운영

오탈자와 정답 오류와 실행 오류를 구분해 Issue로 받는다. 수정 PR에는 바뀐 설명과 검증한 결과를 남긴다. 교사는 정답뿐 아니라 학생이 어디에서 오해했는지를 다음 판본에 반영할 수 있다. 학기가 끝나면 접근과 보관과 외부 실행 자원도 점검한다.

교육 자료의 품질은 기능 수만으로 정해지지 않는다. 처음 보는 학생이 같은 조건에서 내용을 읽고 실습하고 결과를 설명할 수 있는지 확인한다.

## 실습 18.3 | 교사가 다시 쓸 수 있는 자료 구조

경로	내용
README.md	목적과 수업 대상과 시작 방법
docs/lesson.md	활동 흐름과 수식과 도식
assets/	공개 가능한 그림
index.html	브라우저 실습 도구
Releases	확정 배포 PDF와 HTML

1. docs/lesson.md에 수식과 Mermaid 예제를 작성한다.
2. GitHub 화면에서 둘 다 표시되는지 확인한다.
3. 배포 자료가 어느 커밋에서 나왔는지 기록한다.
4. 자료를 재사용할 수 있는 범위와 출처를 적는다.

5. 개선 의견은 Issue로 받고 수정은 PR로 검토한다.

### Classroom을 고려할 때

GitHub Classroom은 개인과 그룹 과제와 교사 대시보드 등을 제공하는 별도 교육 도구다. 이 책의 공개 예제 저장소를 그대로 학생 성적 관리 장소로 쓰는 뜻은 아니다. 기관 계정과 학생의 접근 방식과 제출물 공개 범위와 보관 정책을 먼저 설계한다.

### 교사 공동연구의 기록

실험 도구의 변경과 설명 문서 수정과 자료 판본을 연결한다. 참여자 개인정보와 연구 원자료는 공개 코드 저장소에 섞지 않는다. 연구 결과를 재현할 수 있는 공개 예제와 접근 제한 자료를 각각 정한다.

확인 문제 공개 저장소에 교육 자료를 올리면 누구나 자유롭게 수정하고 배포해도 될까?  
공개와 이용 허락은 별개다. 코드와 글과 그림의 권리와 적용 라이선스를 확인하고 재사용 조건을 명확히 안내한다.

# 자주 쓰는 명령 빠른 찾기 1

## 관찰과 일상 작업

명령을 복사하기 전에 현재 저장소와 브랜치를 확인한다. 아래의 FILE과 BRANCH와 COMMIT-ID는 실제 값으로 바꾼다. 경로에 공백이 있으면 인용한다.

목적	명령	연결 절
상태 확인	<code>git status -sb</code>	3.2절
현재 브랜치	<code>git branch --show-current</code>	3.2절
원격 주소	<code>git remote -v</code>	3.3절
미스테이징 차이	<code>git diff</code>	4.2절
다음 커밋의 차이	<code>git diff --staged</code>	4.2절
파일 추가	<code>git add FILE</code>	4.2절
일부 변경만 추가	<code>git add -p</code>	4.2절
커밋 기록	<code>git commit -m '설명'</code>	4.2절
원격 정보 갱신	<code>git fetch origin</code>	4.4절
앞으로만 통합	<code>git pull --ff-only</code>	4.4절
새 브랜치	<code>git switch -c BRANCH</code>	5.1절
브랜치 이동	<code>git switch BRANCH</code>	5.1절
첫 Push	<code>git push -u origin BRANCH</code>	4.4절
커밋 그래프	<code>git log --graph --oneline --all</code>	6.3절

목적	명령	연결 절
특정 기록	git show COMMIT-ID	6.3절

## 작업 시작의 짧은 루틴

```
git status
git branch --show-current
git fetch origin
변경이 없고 main에서 시작할 때
git switch main
git pull --ff-only
git switch -c feature/NEXT-TOPIC
```

이미 진행 중인 작업 브랜치가 있다면 위 루틴으로 새 브랜치를 계속 만들지 않는다. 현재 작업의 상태와 목적에 맞게 시작 위치를 정한다.

# 자주 쓰는 명령 빠른 찾기 2

## 복구와 GitHub 조회

복구 명령은 바꾸는 대상이 다르다. 파일을 버리는 명령과 커밋을 보존하는 명령을 구분한다. 실제 운영 브랜치에서 연습하지 않는다.

목적	명령	연결 절
스테이징 취소	git restore --staged FILE	6.1절
편집 폐기	git restore FILE	6.1절
공유 커밋 취소	git revert COMMIT-ID	6.1절
참조 이동 기록	git reflog	6.1절
복구 이름 만들기	git branch rescue/name COMMIT-ID	6.1절
Merge 취소	git merge --abort	5.1절
Rebase 취소	git rebase --abort	5.2절
Cherry-pick 취소	git cherry-pick --abort	5.2절
무시 규칙 확인	git check-ignore -v FILE	6.2절
인증 상태	gh auth status	4.1절
열린 Issue	gh issue list	17.1절
열린 PR	gh pr list	17.1절
현재 PR 검사	gh pr checks	17.1절
실행 목록	gh run list	17.1절
실패 로그	gh run view RUN-ID --log-failed	17.1절

목적	명령	연결 절
별도 작업 폴더	git worktree list	16.2절

## 병합 전 짧은 루틴

```
git status
git diff --check
git diff --staged
gh pr checks
gh pr diff
```

각 명령은 관찰 대상이 다르다. 로컬 Index가 비어 있어도 PR에는 커밋된 변경이 있을 수 있다. 전체 PR 차이는 PR 화면이나 gh pr diff로 읽는다.

# 문제 해결 사전 1

## 연결과 Push 문제

오류가 나면 무작정 삭제하거나 강제 옵션을 붙이지 않는다. 메시지와 현재 위치와 재현 조건부터 기록한다.

증상	가능한 원인	첫 확인
not a git repository	다른 폴더에서 실행	pwd와 폴더 경로
repository not found	주소 오류 또는 권한 부족	remote와 인증 계정
Authentication failed	잘못된 인증 또는 만료	HTTPS와 SSH 구분
Permission denied publickey	SSH 키 연결 문제	ssh -T와 등록 키
remote origin already exists	원격 별칭 중복	git remote -v
non-fast-forward	원격 새 커밋	fetch와 log
no upstream branch	첫 추적 관계 없음	push -u와 branch -vv
src refspec main does not match	커밋 없음 또는 이름 오류	log와 branch
파일이 너무 큼	현재 또는 과거 큰 파일	6.2절 크기 기준
웹에서 커밋이 안 보임	Push 전 또는 브랜치 차이	웹 브랜치와 log

## 도움 요청에 붙일 정보

하려던 작업:  
실행한 명령:  
오류 문장:  
현재 폴더와 브랜치:  
언제부터 발생했는지:  
이미 확인한 내용:

토큰과 개인 키와 학생 정보는 지운다. 전체 환경 변수를 통째로 출력해 공유하지 않는다. 스크린샷보다 복사 가능한 오류 텍스트가 검색에 도움이 될 때가 많다.

인증 실패를 해결하려고 저장소를 Public으로 바꾸지 않는다. 인증과 권한 문제를 먼저 진단한다.

# 문제 해결 사전 2

## 통합과 자동화와 배포 문제

증상	가능한 원인	첫 확인
충돌 표시가 파일에 남음	충돌 해결 미완료	status와 diff --check
브랜치 전환 거절	덮어쓸 미커밋 변경	commit 또는 stash 검토
detached HEAD	커밋 직접 선택	새 브랜치로 보존
PR 병합 불가	검사와 승인과 충돌	PR 하단 조건
필수 검사가 계속 대기	이벤트 누락 또는 이름 변경	on과 검사 이름
Actions 파일을 못 찾음	경로와 대소문자 문제	checkout과 작업 폴더
Actions 권한 거절	토큰 최소 권한 불일치	permissions와 조직 정책
Pages 404	경로와 배포 소스 문제	index.html과 배포 기록
CSS와 그림만 안 보임	루트 경로와 대소문자	프로젝트 하위 경로
최신 코드인데 옛 화면	배포 지연 또는 캐시	배포 커밋과 실제 주소
.gitignore가 안 먹힘	이미 추적한 파일	git ls-files FILE
PDF와 그림 diff가 없음	바이너리 비교 한계	별도 원본과 비교 도구

## 원인과 해결을 남긴다

증상과 근본 원인을 구분한다. 서버 오류라는 말만 적기보다 어느 요청이 어떤 권한으로 실패했는지 남긴다. 해결 후 재현 절차를 다시 수행하고 재발 방지 항목을 Issue로 만든다.

## 아직 모르는 경우

오류 직전의 정상 상태와 최초 실패 상태를 좁힌다. 관련 PR과 배포 로그와 의존성 변경을 확인한다. 16.2절의 Bisect는 재현 가능한 회귀 오류에 사용한다. 단순 네트워크 장애나 인증 만료를 커밋 문제로 단정하지 않는다.

# 핵심 용어 사전 1

용어	의미	장
Version control	변경 이력을 기록하고 비교하는 방식	01
Repository	파일과 Git 이력을 가진 저장소	01
Working tree	지금 편집하고 있는 파일 상태	10
Index 또는 Stage	다음 커밋에 넣을 상태	10
Commit	파일 상태와 부모 관계를 가리키는 기록	16
Commit ID 또는 OID	Git 객체를 식별하는 값	16
Branch	커밋을 가리키며 이동하는 이름	04
HEAD	현재 작업 기준을 가리키는 참조	16
Detached HEAD	브랜치 대신 커밋을 직접 선택한 상태	16
Remote	원격 저장소에 붙인 로컬 별칭	09
origin	Clone 때 흔히 붙는 원격 별칭	09
upstream	추적 브랜치 또는 원본 원격을 뜻하는 문맥별 표현	11과 19
Clone	이력을 포함해 로컬로 복제	09

용어	의미	장
Fetch	원격 정보를 로컬로 가져오기	11
Pull	가져온 변경을 로컬에 통합하기	11
Push	로컬 커밋과 참조를 원격으로 보내기	11
Fork	GitHub에서 분리해 이어가는 저장소 사본	19
Diff	두 상태 사이의 차이	10
Merge base	두 이력의 공통 조상 기준	16
Conflict	자동으로 결정하지 못한 통합 부분	12

## 헛갈리는 말

upstream이라는 원격 별칭은 사용자가 붙이는 이름이다. 브랜치의 upstream 설정은 기본으로 비교하고 통합할 추적 대상을 뜻한다. 같은 단어가 항상 같은 대상을 가리키지는 않는다.

## 핵심 용어 사전 2

용어	의미	장
Pull request 또는 PR	변경을 검토하고 합칠 제안	04
Base	PR의 변경을 받을 브랜치	04
Head	PR의 변경을 제공할 브랜치	04
Issue	문제와 요구와 작업 기록	17
Project	Issue와 PR의 계획과 진행 보기	17
Ruleset	브랜치와 태그 등의 작업 규칙	20
CODEOWNERS	경로별 검토 책임 정의 파일	21
Tag	특정 이력 지점의 이름	23
Release	태그 기반 배포 안내와 파일	23
CI	변경을 반복 검사하는 통합 방식	25
Workflow	Actions의 자동 실행 정의	25
Job	Runner에서 실행하는 작업 단위	25
Runner	Job을 실행하는 머신 환경	25
Artifact	실행 결과로 보관하는 파일	26
Secret	안전하게 주입해야 하는 비밀 값	27

용어	의미	장
OIDC	실행 신원을 바탕으로 짧은 자격을 받는 연결 방식	28
Rebase	다른 기준 위에 커밋을 다시 적용	13
Revert	반대 변경을 새 커밋으로 추가	14
Reflog	로컬 참조 이동 기록	14
Lease	원격의 예상 상태를 검사하는 조건	31

## 비슷하지만 다른 결과

Revert는 기존 커밋을 지우지 않는다. Reset은 선택한 모드에 따라 참조와 Index와 작업 파일을 바꾼다. Rebase는 커밋을 다시 만든다. 어떤 명령을 골랐는지보다 어떤 상태를 보존해야 하는지 먼저 말할 수 있어야 한다.

# 학습 계획과 역량 점검

## 4주 기본 과정

주차	범위	직접 만들 결과
1주	1-4장	웹 PR과 로컬 커밋
2주	4-8장	충돌 해결과 리뷰와 보드
3주	10-14장	Pages와 CI와 보안 점검
4주	9-18장	고급 실습과 운영 문서

하루 학습 시간과 경험에 따라 나눠 진행한다. 전문가 장을 처음부터 모두 실행할 필요는 없다. 기초에서 막히면 해당 장을 다시 실습하고 나머지는 읽어 보는 방식으로 이어간다.

## 처음 시작하는 사람의 최소 경로

01 → 02 → 03 → 04 → 05 → 06 → 10 → 11 → 14 → 22 → 18.1절 순서로 진행한다. 설치에 부담이 있으면 1-2장과 10.1절의 웹 실습부터 한다.

## 개발 경험이 있는 사람의 점검 경로

16 → 18 → 20 → 25 → 27 → 28 → 30 → 31 → 33 → 36 → 38 → 18.2절 순서로 빈 부분을 찾는다. 명령을 알아도 권한과 복구 설명이 어렵다면 운영 장을 다시 본다.

## 스스로 답해 보기

## 다음 학습의 기록

아는 용어보다 실제로 막힌 상황을 기록한다. 해결한 Issue와 검토한 PR과 실패를 고친 CI 링크를 모으면 자신의 학습 포트폴리오가 된다.

# 참고문헌과 기술 자료

학술 연구와 기술 문서 및 이 책의 해석을 구분해 읽는다. 새로운 학술 자료와 주요 기술 문서의 확인 기준은 2026년 10월 24일이다. GitHub 원판에서 이어 온 R 자료는 원판의 2026년 10월 21일 확인 기록을 유지하며 주요 변경 사항은 T 자료로 보완했다. 논문의 결과는 해당 표본과 과제 및 시점의 범위에서 해석한다.

## 학술 연구와 연구기관 자료

### [S12] Dabbish 외 / 2012

Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. ACM CSCW / 1277-1286 / DOI 10.1145/2145204.2145396.

활용 범위: 작업 가시성과 협업 / 저자의 논문 소개 및 원문 링크.

<https://www.lauradabbish.com/social-coding-1>

### [S13] Bacchelli와 Bird / 2013

Expectations Outcomes and Challenges of Modern Code Review. ICSE / DOI 10.1109/ICSE.2013.6606617.

활용 범위: 코드 리뷰의 이해와 지식 전달.

<https://www.microsoft.com/en-us/research/publication/expectations-outcomes-and-challenges-of-modern-code-review/>

### [S14] Kalliamvakou 외 / 2014

The Promises and Perils of Mining GitHub. MSR / 92-101 / DOI 10.1145/2597073.2597074.

활용 범위: 저장소 데이터와 활동 지표 해석의 타당도.

[https://kblincoe.github.io/publications/2014_MSR_Promises_Perils.pdf](https://kblincoe.github.io/publications/2014_MSR_Promises_Perils.pdf)

### [S15] Vasilescu 외 / 2015

Continuous Integration in a Social-Coding World: Empirical Evidence from GitHub. ICSME 2014 논문의 수정 공개본 / arXiv 1512.01862.

활용 범위: 기여 방식과 자동 빌드 성공의 관찰 연구.

<https://arxiv.org/abs/1512.01862>

### [S16] Fares 외 / 2026

Unpacking Security Scanners for GitHub Actions Workflows. arXiv 프리프린트 / 2601.14455.

활용 범위: 보안 검사 도구별 범위와 탐지 차이.

<https://arxiv.org/abs/2601.14455>

### [S23] Nugroho와 Hata 및 Matsumoto / 2019

How Different Are Different diff Algorithms in Git?. Empirical Software Engineering / DOI 10.1007/s10664-019-09772-z / arXiv 공개본.

활용 범위: 선택한 Java 프로젝트에서 diff 알고리즘과 저장소 측정 결과의 관계.

<https://arxiv.org/abs/1902.02467>

## 주요 기술 근거

### [T01] Git 객체의 논리 구조

<https://git-scm.com/book/en/v2/Git-Internals-Git-Objects>

### [T02] Git 참조와 원격 추적

<https://git-scm.com/book/en/v2/Git-Internals-Git-References>

### [T03] git diff의 비교 기준

<https://git-scm.com/docs/git-diff>

### [T04] git reset의 모드와 경로 동작

<https://git-scm.com/docs/git-reset>

### [T05] reflog와 만료

<https://git-scm.com/docs/git-reflog>

### **[T06] pull_request_target의 신뢰 경계와 현재 정책**

[https://docs.github.com/en/actions/reference/security/securely-using-pull_request_target](https://docs.github.com/en/actions/reference/security/securely-using-pull_request_target)

### **[T07] Actions의 안전한 사용과 SHA 고정**

<https://docs.github.com/en/actions/reference/security/secure-use>

### **[T08] SLSA 1.2 공급망 명세**

<https://slsa.dev/spec/v1.2/>

### **[T09] git push와 force-with-lease의 기대값**

<https://git-scm.com/docs/git-push>

### **[T10] Merge queue의 검사와 이벤트**

<https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/configuring-pull-request-merges/managing-a-merge-queue>

### **[T11] git fetch와 원격 추적 참조**

<https://git-scm.com/docs/git-fetch>

### **[T18] git ls-files와 미병합 인덱스 단계**

<https://git-scm.com/docs/git-ls-files>

### **[T19] git rev-list와 커밋 집합의 비교**

<https://git-scm.com/docs/git-rev-list>

### **[T20] git update-ref와 기대한 이전 값의 확인**

<https://git-scm.com/docs/git-update-ref>

### **[T21] gitattributes와 줄바꿈 정규화**

<https://git-scm.com/docs/gitattributes>

**[T22] git clone의 이력 깊이와 부분 복제**

<https://git-scm.com/docs/git-clone>

**[T23] git range-diff와 패치 계열 비교**

<https://git-scm.com/docs/git-range-diff>

**[T24] git bundle과 독립 및 증분 묶음**

<https://git-scm.com/docs/git-bundle>

**[T25] GitHub Actions OIDC의 클레임과 식별자**

<https://docs.github.com/en/actions/reference/security/oidc>

**[T26] GitHub Actions 의존성 캐시의 범위와 키**

<https://docs.github.com/en/actions/reference/workflows-and-actions/dependency-caching>

**[T27] GitHub 아티팩트 증명의 개념**

<https://docs.github.com/en/actions/concepts/security/artifact-attestations>

**[T28] GitHub REST API의 조건부 요청과 호출 관리**

<https://docs.github.com/en/rest/using-the-rest-api/best-practices-for-using-the-rest-api>

**[T29] GitHub Webhook의 검증과 전달 식별**

<https://docs.github.com/en/webhooks/using-webhooks/best-practices-for-using-webhooks>

**[T38] git cat-file과 객체의 타입 및 내용**

<https://git-scm.com/docs/git-cat-file>

**[T39] git gc와 도달할 수 없는 객체의 정리**

<https://git-scm.com/docs/git-gc>

## **[T40] GitHub ruleset의 적용 범위와 규칙 결합**

<https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-rulesets/about-rulesets>

## **[T41] Dev Container 메타데이터와 실행 단계**

<https://github.com/devcontainers/spec/blob/main/docs/specs/devcontainerjson-reference.md>

## **[T42] 재현 가능한 빌드의 정의**

<https://reproducible-builds.org/docs/definition/>

# **주제별 공식 문서**

## **[R01] GitHub Hello World**

<https://docs.github.com/en/get-started/using-github/hello-world>

## **[R02] Git 설치와 초기 설정**

<https://docs.github.com/en/get-started/git-basics/set-up-git>

## **[R03] 개인 액세스 토큰 관리**

<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens>

## **[R04] 계정과 데이터 보호**

<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure>

## **[R05] 저장소에 파일 추가**

<https://docs.github.com/en/repositories/working-with-files/managing-files/adding-a-file-to-a-repository>

## **[R06] PR 병합 방식**

<https://docs.github.com/en/pull-requests/reference/pull-request-merges>

### [R07] Markdown 기본 문법

<https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax>

### [R08] GitHub Desktop 시작

<https://docs.github.com/en/desktop/overview/getting-started-with-github-desktop>

### [R09] GitHub CLI 인증

[https://cli.github.com/manual/gh_auth_login](https://cli.github.com/manual/gh_auth_login)

### [R10] Git 명령어 공식 참조

<https://git-scm.com/docs>

### [R11] git push와 force-with-lease

<https://git-scm.com/docs/git-push>

### [R12] git rebase

<https://git-scm.com/docs/git-rebase>

### [R13] git stash

<https://git-scm.com/docs/git-stash>

### [R14] git restore

<https://git-scm.com/docs/git-restore>

### [R15] git reset

<https://git-scm.com/docs/git-reset>

### [R16] git revert

<https://git-scm.com/docs/git-revert>

### [R17] git reflog

<https://git-scm.com/docs/git-reflog>

### [R18] gitignore 규칙

<https://git-scm.com/docs/gitignore>

### [R19] GitHub의 큰 파일 제한

<https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-large-files-on-github>

### [R20] Git LFS 개념

<https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-git-large-file-storage>

### [R21] Pro Git의 객체 설명

<https://git-scm.com/book/en/v2/Git-Internals-Git-Objects>

### [R22] Projects 개념

<https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects>

### [R23] PR과 Issue 연결

<https://docs.github.com/en/issues/tracking-your-work-with-issues/using-issues/linking-a-pull-request-to-an-issue>

### [R24] PR 리뷰 개념

<https://docs.github.com/en/pull-requests/reference/pull-request-reviews>

### [R25] Fork 만들기

<https://docs.github.com/en/pull-requests/how-to-work-with-forks/fork-a-repo>

### [R26] 저장소 라이선스

<https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository>

## **[R27] 저장소 접근 관리**

<https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/managing-repository-settings/managing-teams-and-people-with-access-to-your-repository>

## **[R28] Rulesets 개념과 제공 범위**

<https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/managing-rulesets/about-rulesets>

## **[R29] CODEOWNERS**

<https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners>

## **[R30] GitHub Pages 시작**

<https://docs.github.com/en/pages/quickstart>

## **[R31] Pages의 이용 제한**

<https://docs.github.com/en/pages/getting-started-with-github-pages/github-pages-limits>

## **[R32] Releases 개념**

<https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases>

## **[R33] gh release create**

[https://cli.github.com/manual/gh_release_create](https://cli.github.com/manual/gh_release_create)

## **[R34] github.dev 웹 편집 환경**

<https://docs.github.com/en/codespaces/the-githubdev-web-based-editor>

## **[R35] Codespaces 사용량과 결제**

<https://docs.github.com/en/billing/managing-billing-for-github-codespaces/about-billing-for-github-codespaces>

### [R36] Copilot 책임 있는 사용

<https://docs.github.com/en/copilot/responsible-use/copilot-code-completion>

### [R37] Actions Workflow 문법

<https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax>

### [R38] actions/checkout 공식 저장소

<https://github.com/actions/checkout>

### [R39] actions/setup-python 공식 저장소

<https://github.com/actions/setup-python>

### [R40] 의존성 캐시

<https://docs.github.com/en/actions/reference/workflows-and-actions/dependency-caching>

### [R41] Workflow Artifact 보관

<https://docs.github.com/en/actions/tutorials/store-and-share-data>

### [R42] GITHUB_TOKEN 권한

<https://docs.github.com/en/actions/reference/authentication-in-a-workflow>

### [R43] Actions 안전한 사용

<https://docs.github.com/en/actions/reference/security/secure-use>

### [R44] Actions의 OIDC

<https://docs.github.com/en/actions/concepts/security/openid-connect>

### [R45] 배포와 Environment 보호

<https://docs.github.com/en/actions/reference/workflows-and-actions/deployments-and-environments>

#### [R46] Dependabot 설정 참조

<https://docs.github.com/en/code-security/reference/supply-chain-security/dependabot-options-reference>

#### [R47] Code scanning 개념

<https://docs.github.com/en/code-security/concepts/code-scanning/code-scanning>

#### [R48] Secret scanning 개념

<https://docs.github.com/en/code-security/concepts/secret-security/secret-scanning>

#### [R49] 민감 정보 제거와 키 회수

<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/removing-sensitive-data-from-a-repository>

#### [R50] gh pr create

[https://cli.github.com/manual/gh_pr_create](https://cli.github.com/manual/gh_pr_create)

#### [R51] Issue와 PR 검색

<https://docs.github.com/en/search-github/searching-on-github/searching-issues-and-pull-requests>

#### [R52] REST API 페이지네이션

<https://docs.github.com/en/rest/using-the-rest-api/using-pagination-in-the-rest-api>

#### [R53] Webhook 서명 검증

<https://docs.github.com/en/webhooks/using-webhooks/validating-webhook-deliveries>

#### [R54] GitHub App 개념

<https://docs.github.com/en/apps/creating-github-apps/about-creating-github-apps/about-creating-github-apps>

#### [R55] git worktree

<https://git-scm.com/docs/git-worktree>

## **[R56] git bisect**

<https://git-scm.com/docs/git-bisect>

## **[R57] Sparse checkout**

<https://git-scm.com/docs/git-sparse-checkout>

## **[R58] Submodule**

<https://git-scm.com/docs/git-submodule>

## **[R59] Merge queue와 제공 범위**

<https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/configuring-pull-request-merges/managing-a-merge-queue>

## **[R60] 커밋 서명 검증**

<https://docs.github.com/en/authentication/managing-commit-signature-verification/about-commit-signature-verification>

## **[R61] Artifact attestation**

<https://docs.github.com/en/actions/how-to-secure-your-work/use-artifact-attestations/use-artifact-attestations>

## **[R62] 저장소 백업**

<https://docs.github.com/en/repositories/archiving-a-github-repository/backing-up-a-repository>

## **[R63] Actions 사용량과 결제**

<https://docs.github.com/en/billing/managing-billing-for-your-products/managing-billing-for-github-actions/about-billing-for-github-actions>

## **[R64] git cherry-pick**

<https://git-scm.com/docs/git-cherry-pick>

## **[R65] git diff**

<https://git-scm.com/docs/git-diff>

### **[R66] git log**

<https://git-scm.com/docs/git-log>

### **[R67] gitattributes와 줄바꿈**

<https://git-scm.com/docs/gitattributes>

### **[R68] git fetch**

<https://git-scm.com/docs/git-fetch>

### **[R69] git merge**

<https://git-scm.com/docs/git-merge>

### **[R70] git commit**

<https://git-scm.com/docs/git-commit>

### **[R71] GitHub GraphQL API**

<https://docs.github.com/en/graphql/overview/about-the-graphql-api>

### **[R72] git status**

<https://git-scm.com/docs/git-status>

### **[R73] 저장소 만들기 공식 화면**

<https://docs.github.com/en/repositories/creating-and-managing-repositories/quickstart-for-repositories>

### **[R74] GitHub 코드 검색 문법**

<https://docs.github.com/en/search-github/github-code-search/understanding-github-code-search-syntax>

### **[R75] 커밋과 줄의 영구 링크**

<https://docs.github.com/en/repositories/working-with-files/using-files/getting-permanent-links-to-files>

### **[R76] Desktop Clone 공식 화면**

<https://docs.github.com/en/desktop/adding-and-cloning-repositories/cloning-and-for-king-repositories-from-github-desktop>

### **[R77] VS Code Source Control**

<https://code.visualstudio.com/docs/sourcecontrol/overview>

### **[R78] 재사용 Workflow**

<https://docs.github.com/en/actions/how-tos/reuse-automations/reuse-workflows>

### **[R79] Pages 사용자 지정 Workflow**

<https://docs.github.com/en/pages/getting-started-with-github-pages/using-custom-workflows-with-github-pages>

### **[R80] Issue Form 문법**

<https://docs.github.com/en/communities/using-templates-to-encourage-useful-issues-and-pull-requests/syntax-for-issue-forms>

### **[R81] Container registry**

<https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry>

### **[R82] git range-diff**

<https://git-scm.com/docs/git-range-diff>

### **[R83] git bundle**

<https://git-scm.com/docs/git-bundle>

### **[R84] Markdown의 수식**

<https://docs.github.com/en/get-started/writing-on-github/working-with-advanced-formatting/writing-mathematical-expressions>

**[R85] Markdown의 Mermaid 도식**

<https://docs.github.com/en/get-started/writing-on-github/working-with-advanced-formatting/creating-diagrams>

**[R86] GitHub Classroom 개념**

<https://docs.github.com/en/education/manage-coursework-with-github-classroom/get-started-with-github-classroom/about-github-classroom>

**[R87] gh api의 필드와 파일 입력**

[https://cli.github.com/manual/gh_api](https://cli.github.com/manual/gh_api)

# 마지막 점검과 인계

점검의 목적은 확인한 범위를 증거로 설명하는 것이다. 실행하지 않은 항목은 미확인으로 남기고 담당자와 다음 행동을 기록한다. 체크 표시만으로 검증이 완료되었다고 판단하지 않는다.

## 요구와 결과의 일치

첫 사용자가 수행할 행동과 기대 결과를 한 문장으로 설명한다. 실제 입력과 관찰 결과를 연결한다. 화면이 보인다는 사실과 기능이 정확하다는 판단을 구분한다.

## 변경과 판본의 식별

현재 파일과 검사한 판본 및 공개한 결과를 연결한다. 변경 이유와 확인 방법을 남긴다. 되돌릴 기준과 미커밋 자료의 보존 위치를 확인한다.

## 실패와 경계의 검증

빈 자료와 잘못된 입력 및 권한이 없는 요청을 확인한다. 동시 수정과 재시도 및 오래된 자료의 처리 정책을 읽는다. 모형으로 확인한 규칙과 실제 실행 환경에서 확인한 결과를 구분한다.

## 자료와 권한 및 운영

비밀과 개인정보가 소스 및 로그와 배포 파일에 노출되지 않는지 확인한다. 실행 주체의 권한과 외부 의존성 및 비용을 검토한다. 복구 절차를 별도 환경에서 시험한다.

## 다른 사람에게 인계

처음 읽는 사람이 실행하고 작은 자료 한 개를 수정하게 한다. 설명이 필요한 지점은 문서에 반영한다. 남은 문제와 책임자 및 다음 검토 시점을 기록한다.

판정 항목	남길 증거	상태
요구 충족	입력과 기대 및 실제 결과	확인 / 미확인
회귀 확인	변경 전후 검사 기록	확인 / 미확인
권한 경계	허용과 거부의 실제 시험	확인 / 미확인
배포 식별	커밋과 실행 환경	확인 / 미확인
복원 가능성	분리 환경의 복원 결과	확인 / 미확인
인계 가능성	다른 사람의 실행과 수정	확인 / 미확인

마지막으로 현재 결과의 한계를 한 문단으로 적는다. 한계를 정확히 설명할 수 있는 결과는 다음 변경의 출발점이 된다.