M06-01 · GIBALJA VISUAL MANUAL

Git 저장소와 변경 이력 읽기

> **한 문장 목표:** Git 명령을 변경 버튼으로 쓰기 전에 `저장소 위치 → HEAD·ref → working tree·index 상태 → history graph → diff endpoint → file 종류 → 의도·위험 → 증거` 순서로 읽고, 어떤 파일이 왜 바뀌었는지 재현 가능하게 설명합니다.

난이도	개념	실습	셀프 테스트	최종 산출물
Level 1	60분	60분	15분	저장소 지도, 상태·기준점 기록, commit·diff 분석표, 변경 검토 보고서

`git diff`가 비어 있다고 변경이 없는 것은 아닙니다. 변경이 이미 index에 staged되어 있을 수 있습니다. `main`이라고 같은 기준도 아닙니다. branch는 새 commit마다 움직이고, 마지막 fetch 뒤의 `origin/main`은 실제 server의 현재 상태와 다를 수 있습니다. Git을 안전하게 읽으려면 명령과 함께 repository·HEAD·endpoint·pathspec을 기록해야 합니다.

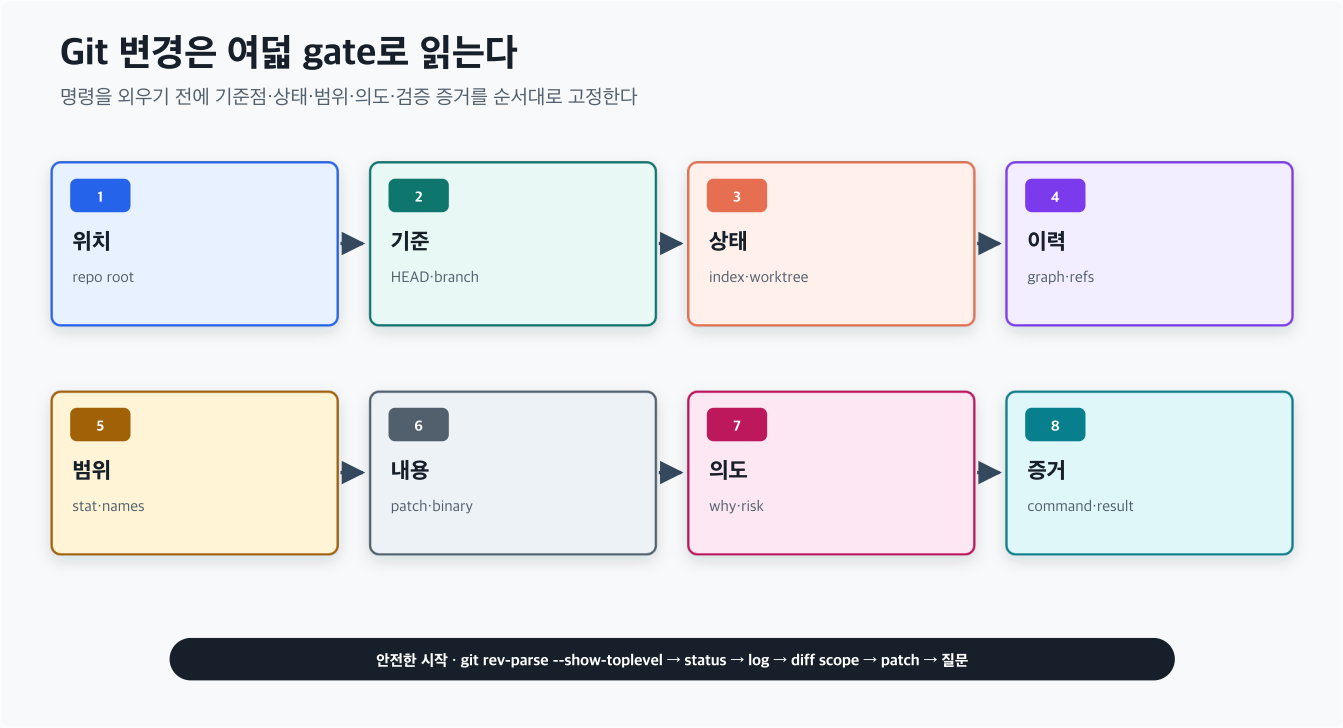
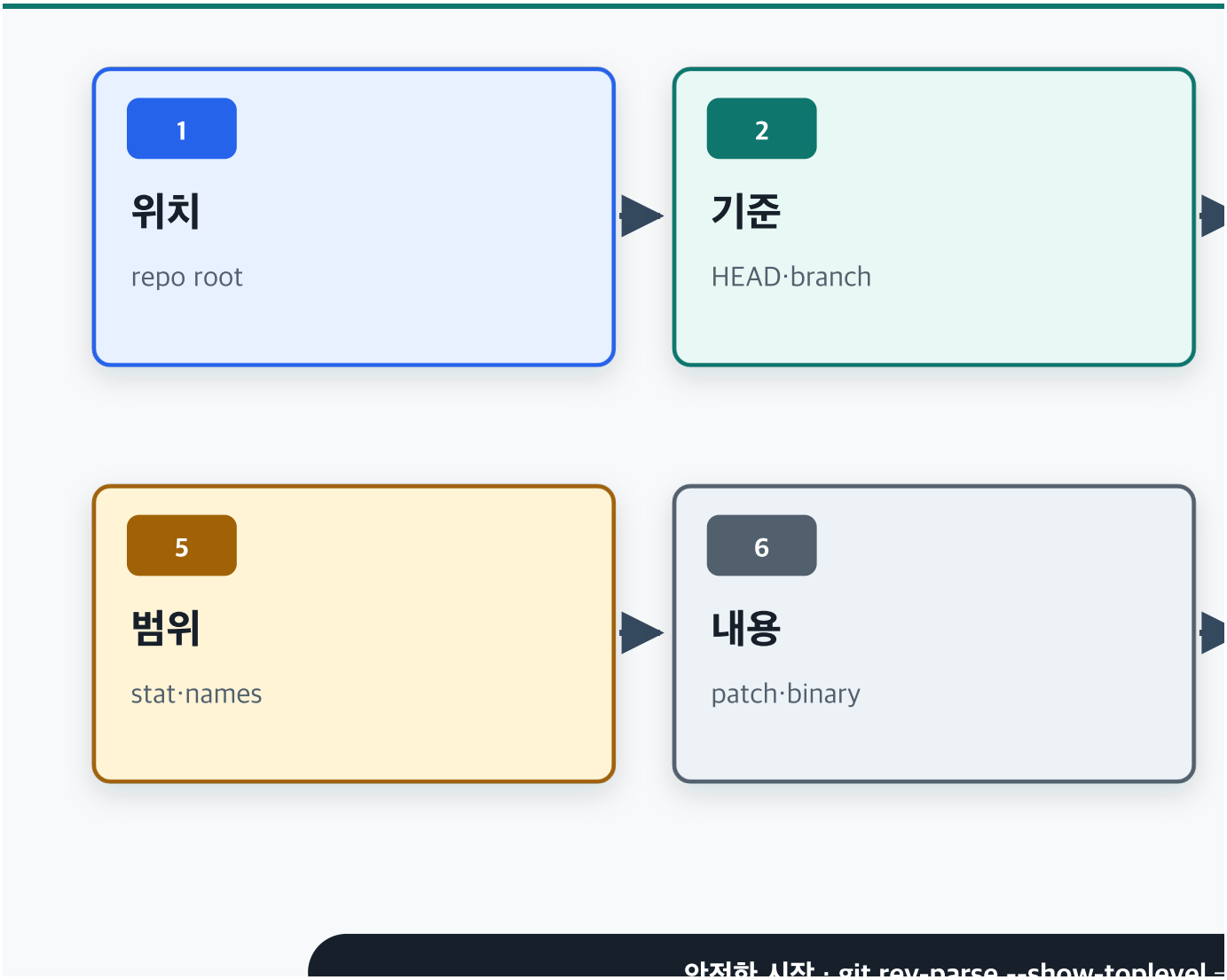
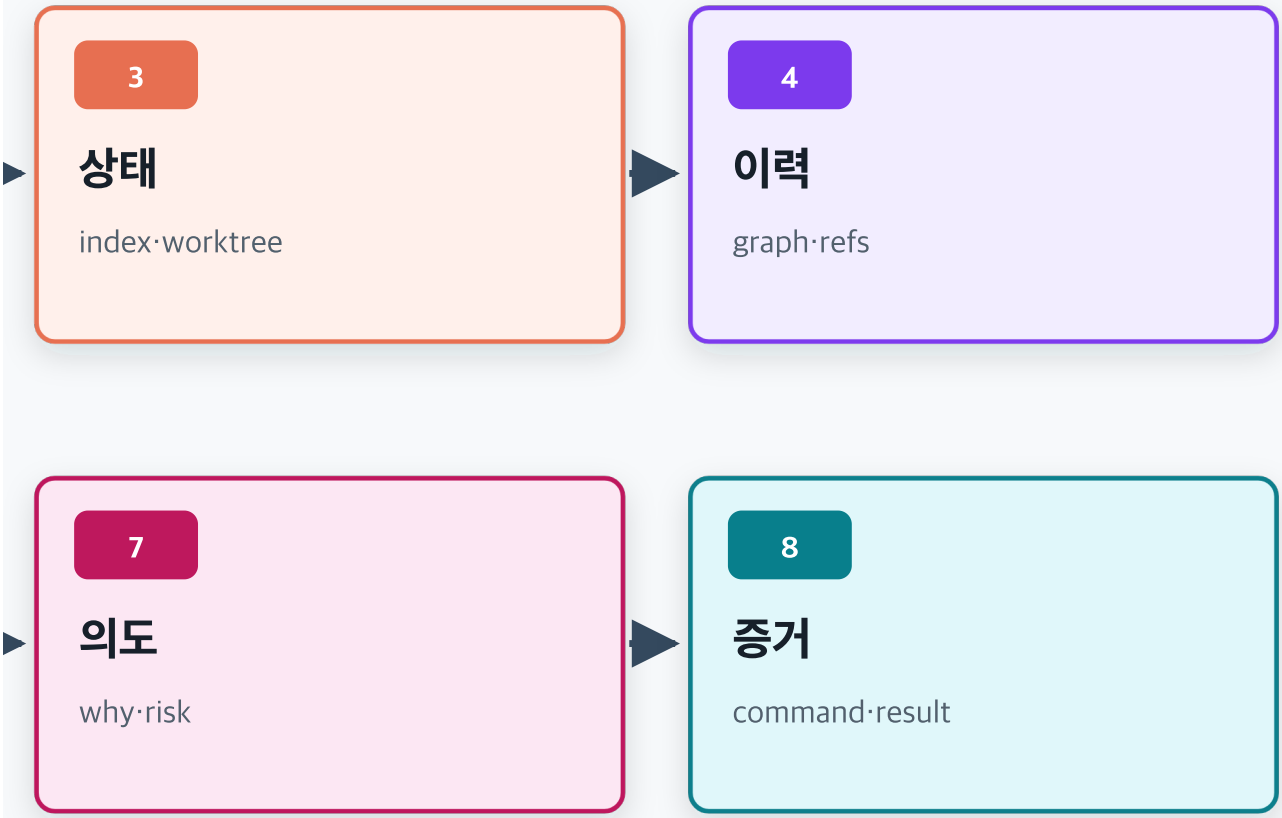


그림 1. Git 변경 검토는 patch에서 시작하지 않습니다. 위치와 기준점을 고정하고 범위를 좁힌 뒤 내용·의도·검증 증거를 연결합니다.


1
위치
repo root
2
기준
HEAD·branch
5
범위
stat·names
6
내용
patch·binary



→ status → log → diff scope → patch → 질문

0. 이 PDF를 공부하는 방법

1회차 · 그림만 읽기 · 20분

그림 1부터 14까지 제목과 맨 아래 결론 띠만 읽습니다. 다음 문장을 소리 내어 말합니다.

```
repository root, branch, HEAD를 먼저 기록한다.
working tree, index, HEAD는 서로 다른 세 상태다.
commit은 patch가 아니라 project snapshot과 parent를 기록한다.
short status의 X는 index, Y는 working tree 차이다.
git diff는 endpoint에 따라 다른 변경을 보여 준다.
history는 날짜 목록이 아니라 parent graph다.
rename, binary, generated, config, test는 검토 증거가 다르다.
.gitignore는 이미 tracked된 secret을 history에서 지우지 않는다.
```

2회차 · 실제 연습 repository 만들기 · 15분

터미널에서 다음 생성기를 실행합니다. 생성기는 기존 폴더를 덮어쓰지 않고, 별도 local bare remote와 practice repository를 만듭니다.

```
./02_Labs/G06_Git/L06-01_create-practice-repo.sh
```

생성된 경로는 실행 결과 마지막 줄에 표시됩니다. Git version, repository path, starting status, history graph를 실습 기록에 옮깁니다.

3회차 · 12개 실제 출력 읽기 · 40분

Git 증거 리더를 열어 12개 증거를 순서대로 봅니다. 해설을 열기 전에 comparison endpoint와 출력의 한 줄을 설명합니다.

4회차 · 내 저장소 분석표 작성 · 60분

Git 저장소·변경 이력 분석표를 채웁니다. 처음에는 read-only 관찰 명령만 사용합니다. `add`, `commit`, `restore`, `reset`, `clean`, `merge`, `rebase`, `push`는 이번 매뉴얼의 완료 조건이 아닙니다.

1. Git은 파일 폴더가 아니라 snapshot graph를 가진 저장소입니다

1.1. repository identity

Git 명령은 현재 directory에서 위쪽으로 `.git`을 찾을 수 있습니다. 그래서 내가 생각한 project가 아닌 상위 repository를 읽을 수 있습니다. 첫 세 명령으로 위치와 기준을 고정합니다.

```
git --version
git rev-parse --show-toplevel
git rev-parse --short HEAD
git branch --show-current
```

항목	역할	예시
Git version	출력·기능 차이 기준	`git version 2.54.0`
top-level	working tree의 root	`/practice/L06-01/repo`
HEAD object ID	현재 commit 기준	`7e0337b`
current branch	HEAD가 가리키는 branch 이름	`main`

`git rev-parse --show-toplevel`이 실패하면 Git working tree 안이 아니거나 bare repository처럼 working tree가 없는 환경일 수 있습니다. 그 상태에서 경로를 추측해 다음 명령을 진행하지 않습니다.

1.2. `.git`의 역할

Git 공식 command overview는 보통 project working directory의 `.git` 안에 압축된 object database, working tree와 history를 연결하는 index, branch head·tag 같은 named pointer가 있다고 설명합니다.

```
repo/
├── .git/              repository metadata·objects·refs·index
├── README.md          working tree file
├── app/
└── config/
```

`.git`은 일반 source folder가 아닙니다. 직접 파일을 수정하지 않습니다. Git command를 통해 조회·변경합니다.

1.3. untrusted repository 경계

Git 공식 보안 안내는 신뢰하지 못한 `.git` directory와 그 주변 working tree에서 Git command를 실행하는 일을 안전하다고 단정하지 않습니다. repository config와 hook은 shell command 실행에 영향을 줄 수 있습니다. 출처가 불명확한 repository에서는 다음을 지킵니다.

- `.git` directory를 zip으로 받아 그대로 실행하지 않습니다.
- build·install·test·hook을 먼저 실행하지 않습니다.
- 별도 계정·container·VM 같은 격리 환경을 검토합니다.
- repository config·submodule·script·dependency를 확인합니다.
- secret이 있는 home directory와 credential helper 접근을 제한합니다.

30초 확인 1
현재 폴더 이름이 `frontend`라는 사실만으로 어떤 Git repository를 읽고 있는지 증명할 수 있습니까? 최소 어떤 세 값을 기록해야 합니까?

2. working tree·index·HEAD·object database를 구분합니다

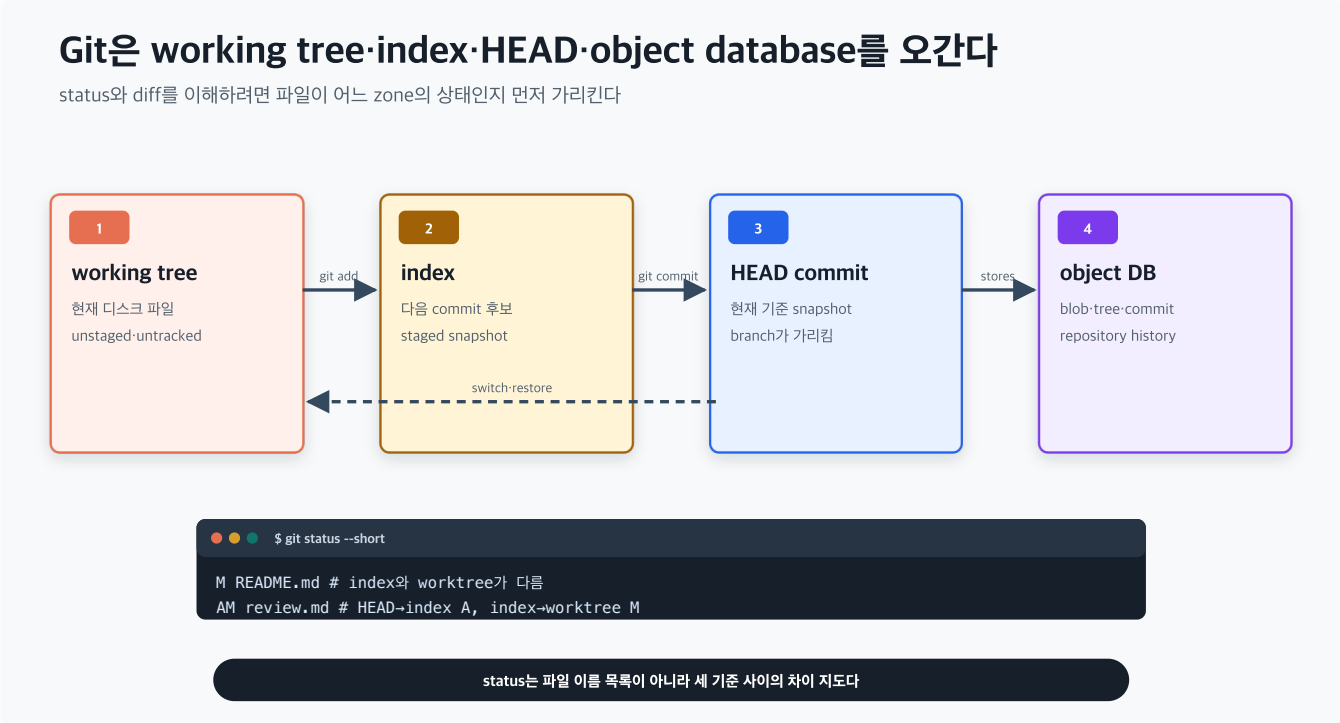
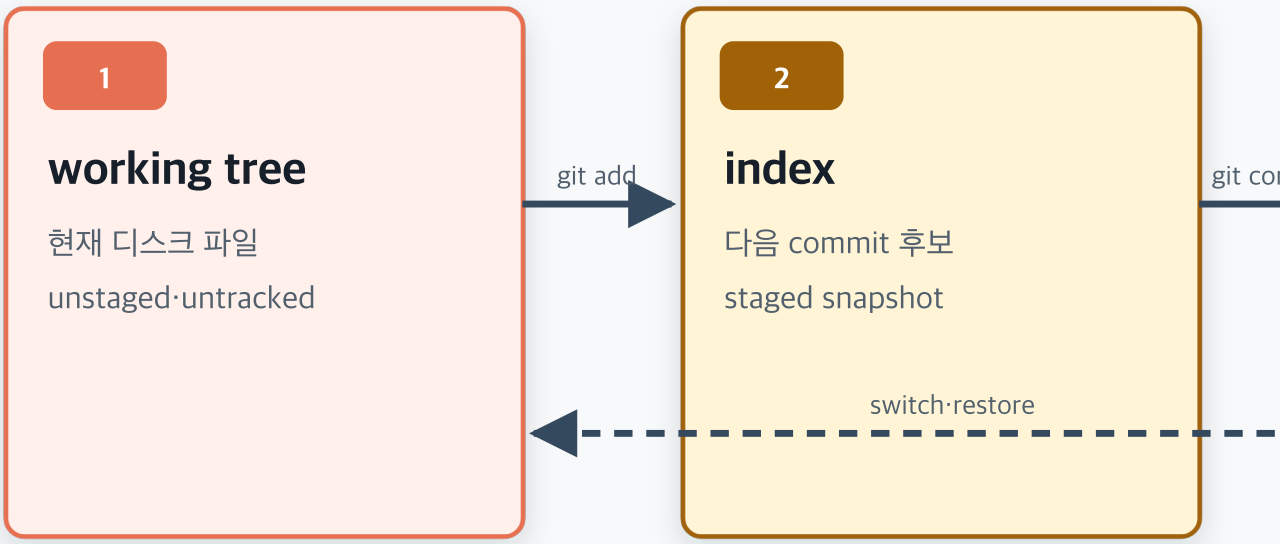
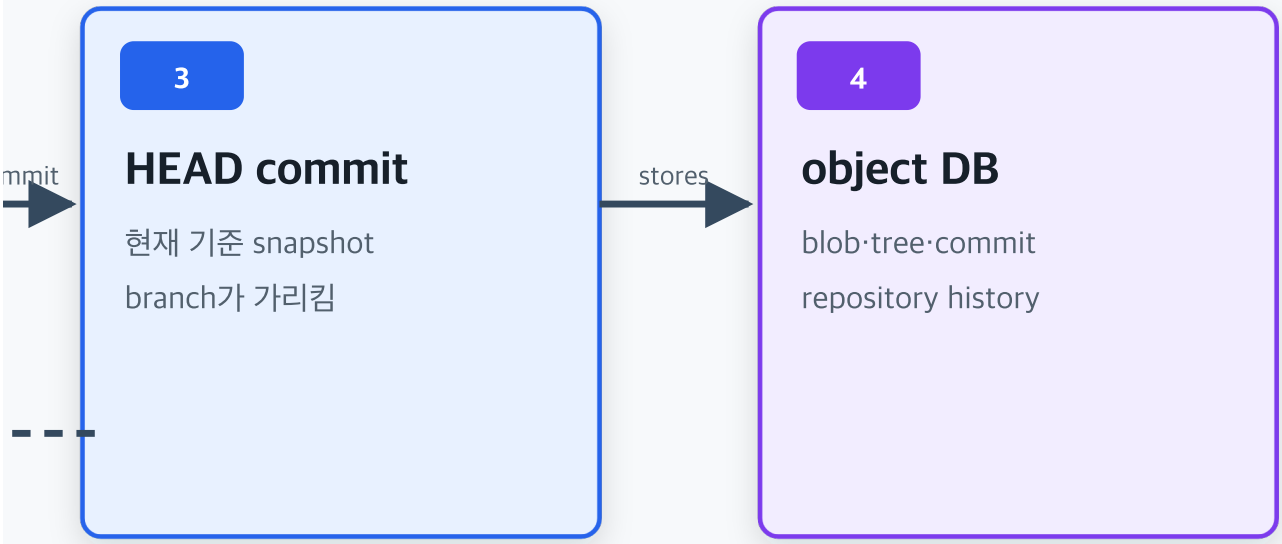


그림 2. Git status와 diff는 세 zone 사이 차이를 읽는 도구입니다. object database는 snapshot과 parent graph의 history를 보관합니다.




```
$ git status --short
M README.md # index와 worktree가 다름
AM review.md # HEAD→index A, index→worktree M
```


mmit
3
HEAD commit
현재 기준 snapshot
branch가 가리킴
stores
4
object DB
blob·tree·commit
repository history

2.1. working tree

working tree는 지금 디스크에서 열어 편집하는 실제 파일입니다. HEAD snapshot의 파일에 local 변경이 더해질 수 있습니다. Git glossary는 working tree를 HEAD commit의 tree 내용과 아직 commit하지 않은 local change가 있는 실제 checked-out file tree로 설명합니다.

2.2. index · staging area

index는 다음 commit에 넣을 후보 snapshot입니다. 단순 파일 목록이 아니라 content와 stat 정보를 가진 저장된 working tree version입니다. 같은 path의 working tree를 다시 수정하면 index와 current file이 달라질 수 있습니다.

2.3. HEAD commit

HEAD는 보통 현재 branch를 상징적으로 가리키고, 그 branch는 현재 기준 commit을 가리킵니다. HEAD commit의 tree가 status와 diff의 기준점이 됩니다.

2.4. object database

Git의 핵심 object를 초급 수준에서 다음처럼 읽습니다.

object	저장하는 것	연결
blob	file 내용	tree가 가리킴
tree	directory·file name·mode·blob/tree	commit이 root tree를 가리킴
commit	tree·parent·author·committer·message	branch·tag가 가리킬 수 있음
annotated tag	target object·tagger·message·signature 정보	`refs/tags/...`가 가리킴

3. commit은 diff가 아니라 snapshot과 parent를 기록합니다

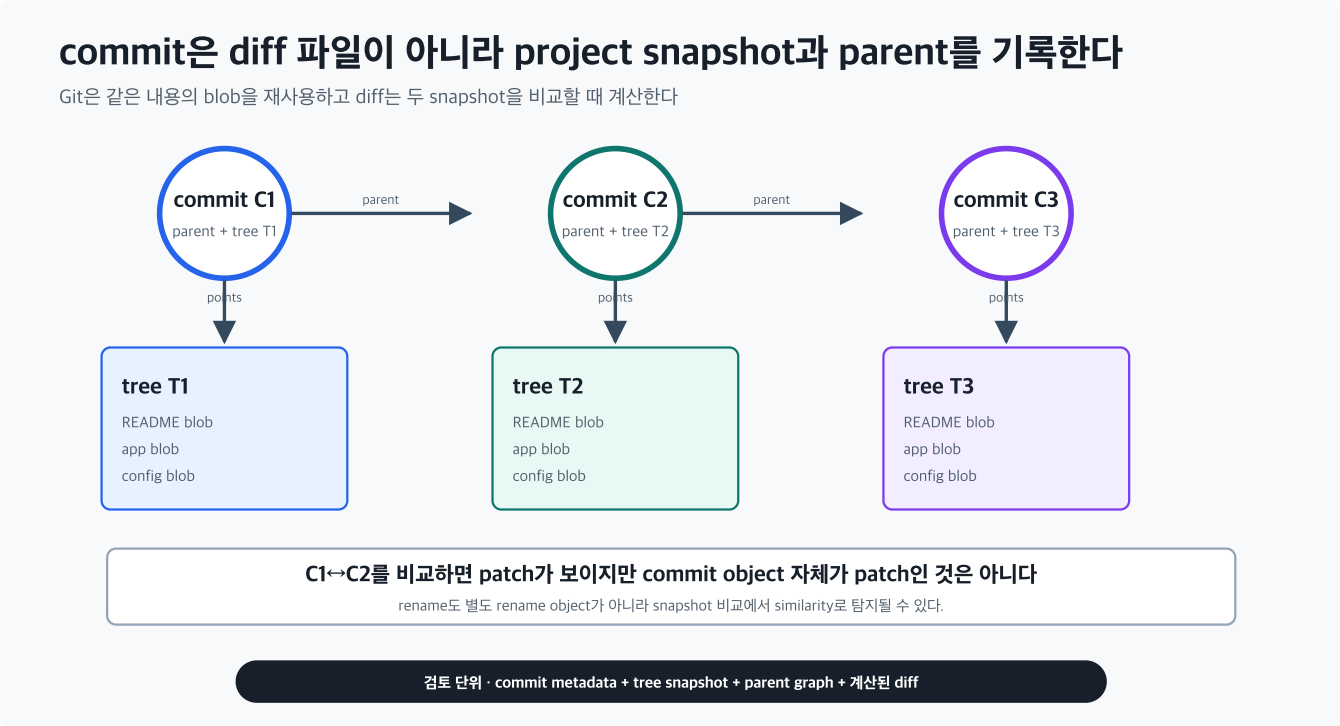
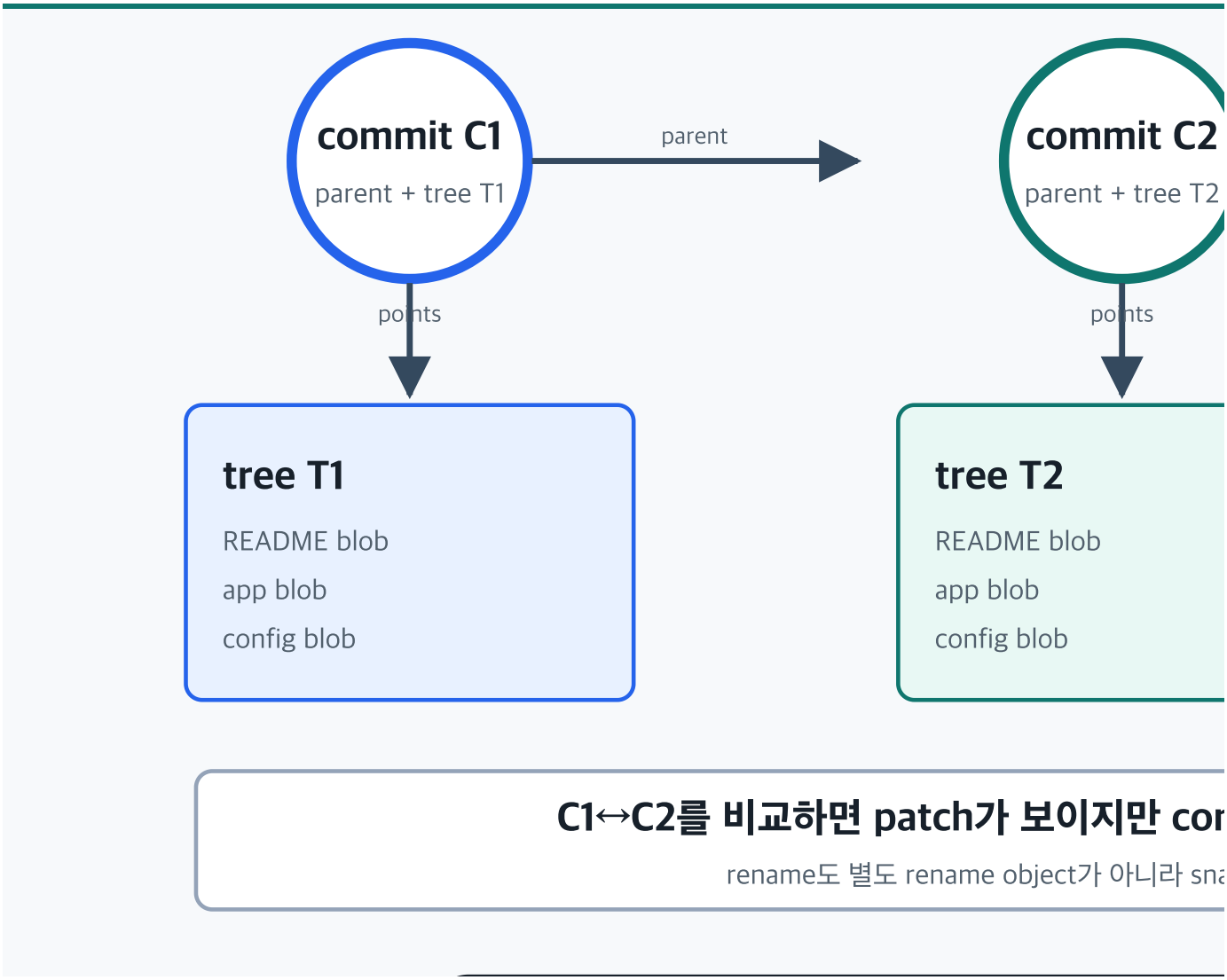
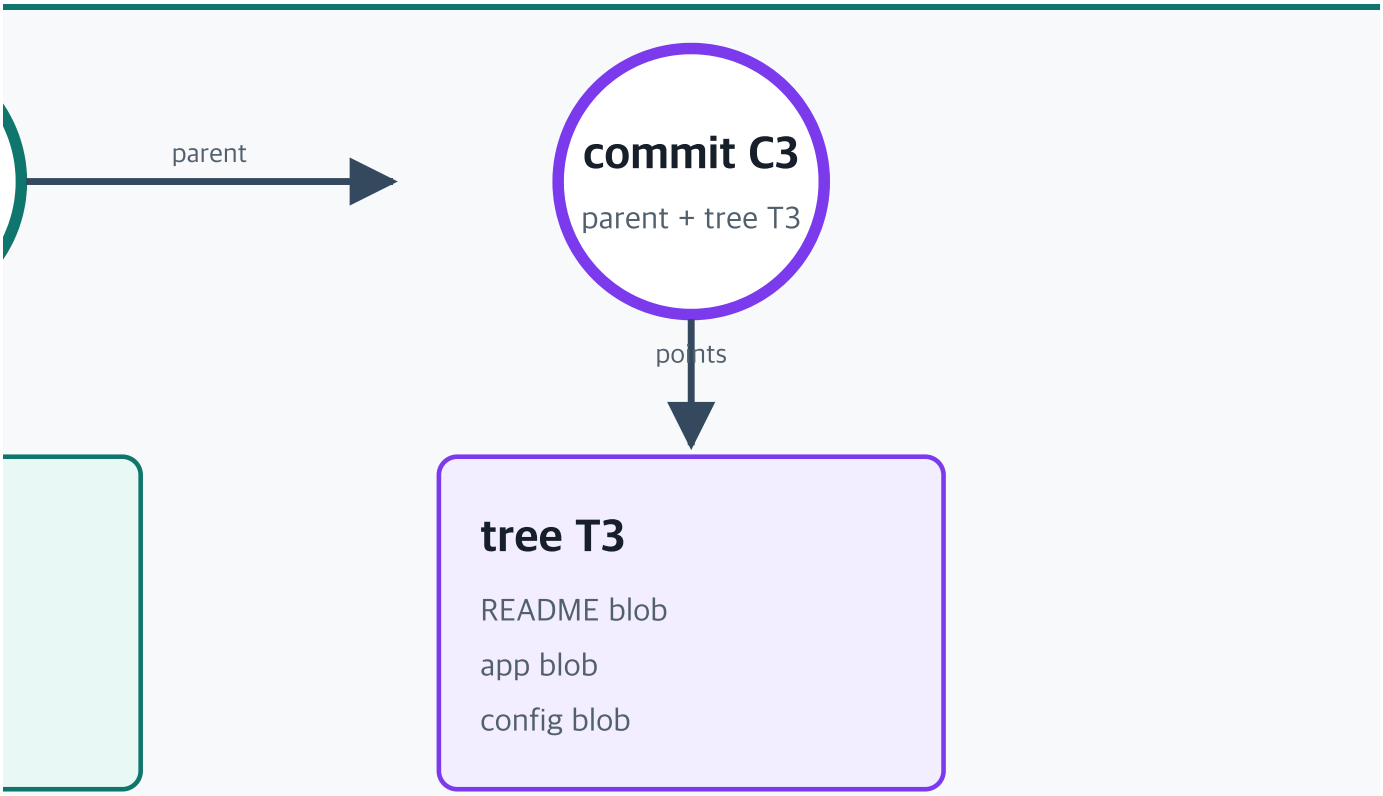


그림 3. commit은 project state의 tree snapshot과 parent를 기록합니다. patch는 두 snapshot을 비교할 때 만들어집니다.


commit C1
parent + tree T1
parent
commit C2
parent + tree T2
points
points
tree T1
README blob
app blob
config blob
tree T2
README blob
app blob
config blob
C1↔C2를 비교하면 patch가 보이지만 cor
rename도 별도 rename object가 아니라 sna



nmit object 자체가 patch인 것은 아니다

apshot 비교에서 similarity로 탐지될 수 있다.

Git glossary는 Git이 changeset을 저장하는 방식으로 생각하기보다 state를 저장한다고 설명합니다. commit object는 하나의 tree와 parent commit을 가리킵니다. unchanged file content는 같은 blob을 재사용할 수 있습니다.

snapshot 관점이 중요한 이유

- commit을 "추가 10줄 파일"로만 보면 삭제·rename·mode·binary를 놓칩니다.
- 두 commit의 patch가 같아 보여도 parent가 다르면 graph에서 다른 commit입니다.
- merge commit은 여러 parent를 가질 수 있습니다.
- rename은 commit object의 영구 file identity가 아니라 두 snapshot의 similarity 비교에서 탐지될 수 있습니다.
- current working tree는 아직 어떤 commit snapshot에도 들어가지 않은 변경을 가질 수 있습니다.

commit identity 읽기

```
git show -s --format=fuller HEAD
```

field	질문
commit ID	정확히 어느 object입니까
Author·AuthorDate	누가 원 변경을 언제 작성했습니까
Committer·CommitDate	누가 이 history에 언제 기록했습니까
parent	어느 snapshot을 기준으로 합니까
message	변경자가 주장하는 의도는 무엇입니까

author와 committer는 같을 수도 다를 수도 있습니다. message는 중요한 맥락이지만 실제 diff·test·issue의 증거를 대신하지 않습니다.

4. HEAD·branch·tag·remote-tracking ref를 구분합니다

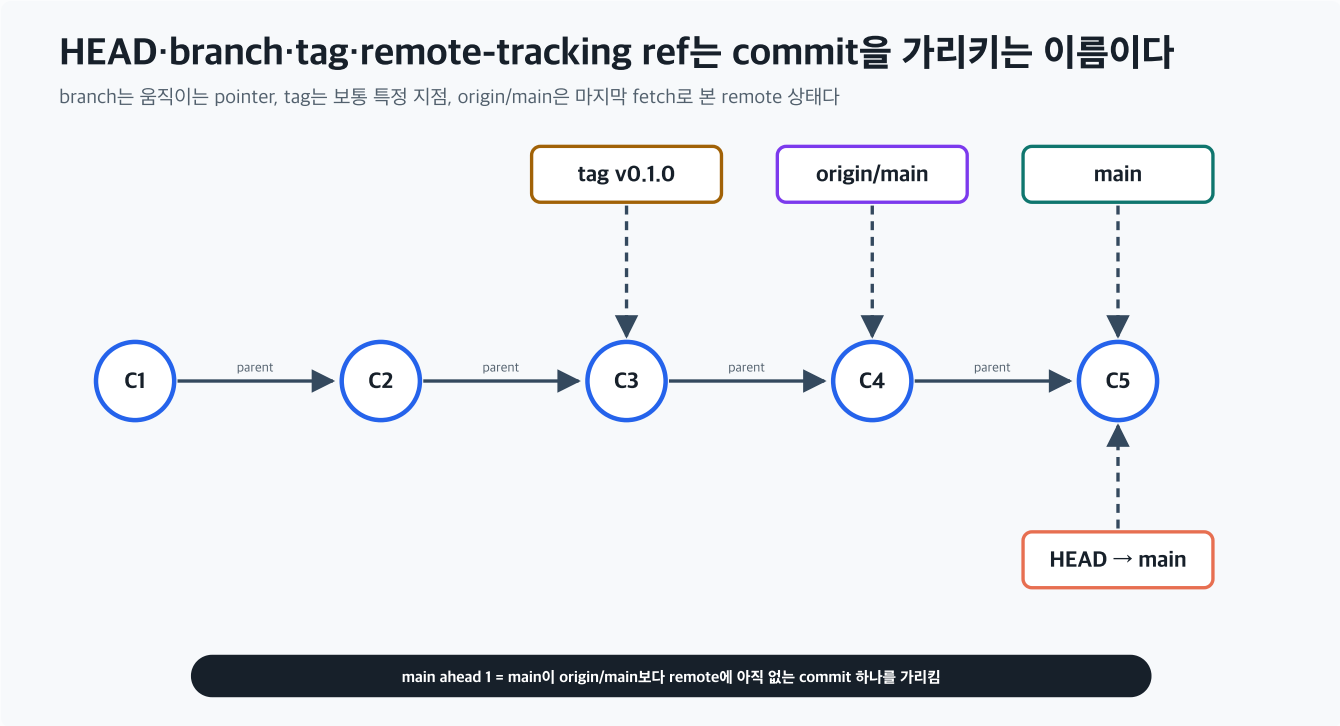
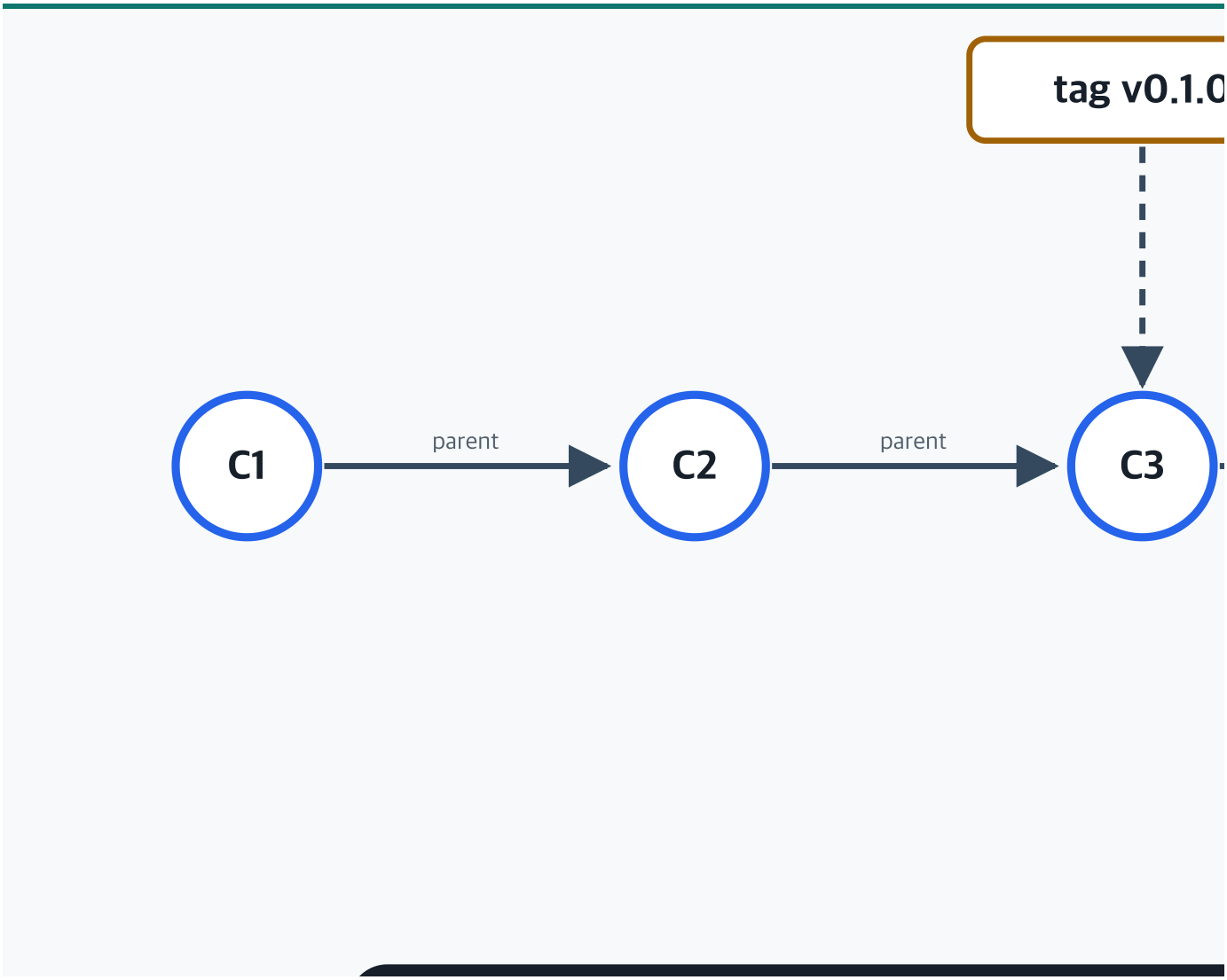
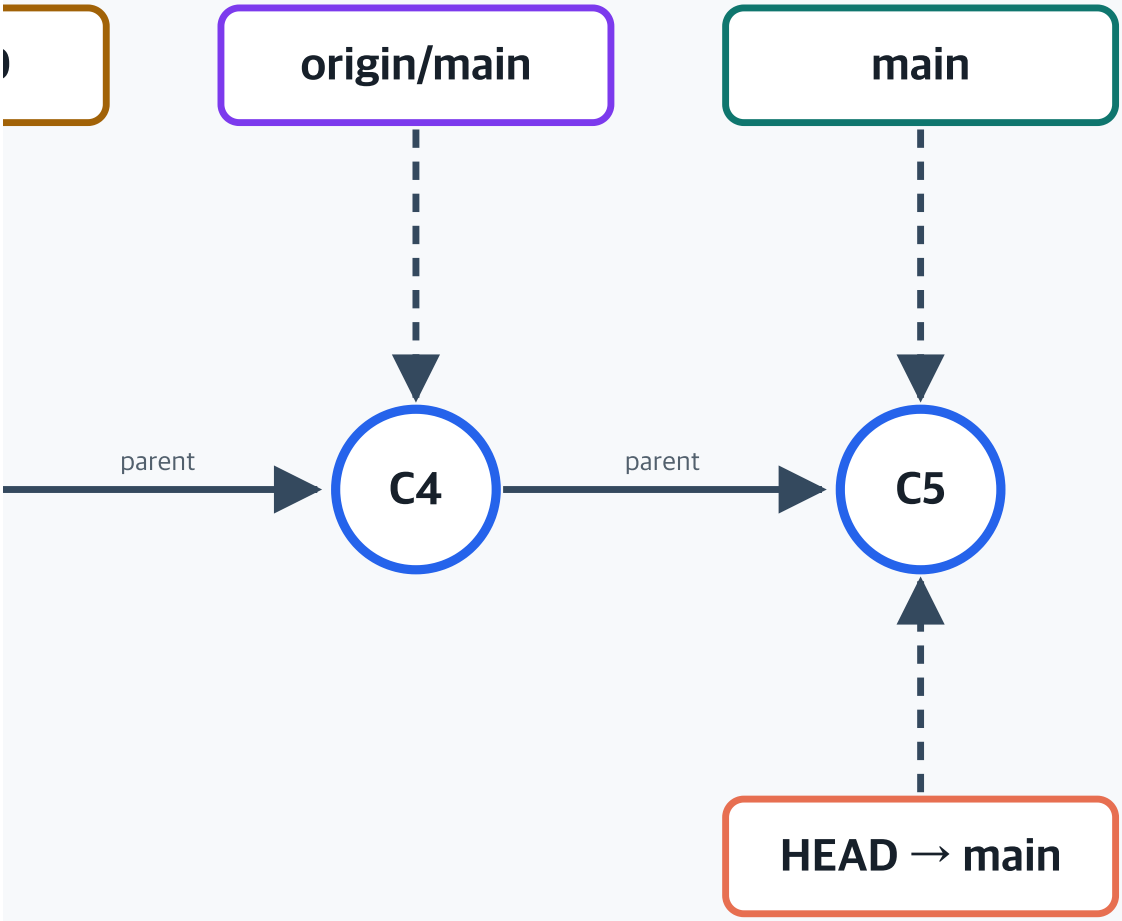


그림 4. branch와 tag는 commit을 복사한 폴더가 아니라 object를 가리키는 ref입니다. HEAD는 현재 작업 기준을 가리킵니다.


tag v0.1.0
C1
parent
C2
parent
C3


origin/main
main
parent
C4
parent
C5
HEAD → main

4.1. branch

branch는 개발 line의 tip을 가리키는 움직이는 ref입니다. current branch에서 새 commit을 만들면 branch ref가 새 commit으로 이동합니다.

```
git branch --show-current
git branch --verbose --verbose
```

4.2. tag

tag는 보통 release·milestone 같은 특정 history 지점을 표시합니다. lightweight tag는 직접 object를 가리키고 annotated tag는 tagger·message·signature를 담는 tag object를 둘 수 있습니다.

```
git tag --list
git show --no-patch --format=fuller v0.1.0
```

4.3. remote와 remote-tracking branch

`origin`은 관례적으로 remote 이름일 뿐 server 자체와 같은 말은 아닙니다. `origin/main`은 local repository가 마지막 fetch에서 관찰한 remote `main`의 상태를 나타내는 remote-tracking ref입니다.

```
git remote --verbose
git rev-parse --abbrev-ref --symbolic-full-name '@{upstream}'
```

`git fetch`는 working tree file을 합치지 않지만 network에 접근하고 local remote-tracking refs와 object database를 갱신합니다. 그러므로 완전한 read-only도 아닙니다. fetch 전·후 기준점과 credential·network 영향을 구분합니다.

ahead·behind

```
## main...origin/main [ahead 1]
```

현재 main에서 reachable하지만 origin/main에서는 reachable하지 않은 commit이 1개라는 뜻입니다. remote server가 지금도 같은 상태라는 보장은 마지막 fetch 시각에 달렸습니다.

5. status는 세 기준 사이의 차이 지도입니다

short status의 X는 index, Y는 working tree 차이다

두 글자를 한 덩어리로 외우지 말고 HEAD→index와 index→worktree를 따로 읽는다

XY path	X = HEAD → index	Y = index → working tree
`M README.md`	index와 HEAD 같음	worktree modified
`M config.json`	index modified	worktree와 index 같음
`AM review.md`	index에 새 파일	stage 뒤 worktree 수정
`?? note.txt`	untracked	아직 index entry 없음
`!! .env`	ignored	--ignored에서만 표시

AM은 staged 완료가 아니다 · 새 파일 snapshot 뒤에도 local 수정이 남아 있다

그림 5. short status의 첫 칸 X는 HEAD→index, 둘째 칸 Y는 index→working tree입니다. `AM`은 staged 새 파일에 unstaged 수정이 더 있다는 뜻입니다.

XY path	X = HEAD → index
`M README.md`	index와 HEAD 같음
`M config.json`	index modified
`AM review.md`	index에 새 파일
`?? note.txt`	untracked
`!! .env`	ignored

AM은 staged 왼쪽과 아니다 : 새 파일 s

```
Y = index → working tree
```

worktree modified

worktree와 index 같음

stage 뒤 worktree 수정

아직 index entry 없음

--ignored에서만 표시

napshot 뒤에도 local 수정이 남아 있다

git status 공식 문서는 HEAD와 index가 다른 path, index와 working tree가 다른 path, untracked path를 보여준다고 설명합니다.

5.1. 사람용 short status

```
git status --short --branch --untracked-files=all
```

연습 저장소의 시작 결과:

```
## main...origin/main [ahead 1]
 M README.md
AM docs/review-checklist.md
?? notes/local-review.txt
```

code	HEAD→index X	index→working tree Y	판정
`M`	같음	modified	unstaged 수정
`M`	modified	같음	staged 수정
`AM`	added	modified	새 파일을 stage한 뒤 다시 수정
`??`	index entry 없음	untracked	add 전 새 path
`!!`	ignore match	ignore match	`--ignored`에서 표시

5.2. script용 porcelain

```
git status --porcelain=v2 --branch
```

porcelain format은 script가 parse하기 위한 안정된 형식입니다. 사람용 문장의 번역·color·설정 영향을 그대로 parse하지 않습니다. version 2는 branch OID, head, upstream, ahead·behind와 path별 자세한 정보를 제공합니다.

5.3. clean의 의미

working tree가 clean이라는 말은 current HEAD에 대응하고 local tracked change가 없다는 뜻입니다. 다음을 자동 증명하지는 않습니다.

- application이 올바르게 동작함

- remote server와 최신으로 동기화됨
- ignored file에 secret·generated output이 없음
- untracked file을 `--untracked-files=no`로 숨기지 않았음
- 다른 branch에 중요한 commit이 없음

30초 확인 2

`git diff`는 비어 있는데 `git status`에 `M config.json`이 보입니다. 변경이 어디에 있으며 어떤 명령으로 봅니까?

6. git diff는 endpoint를 적어야 뜻이 생깁니다

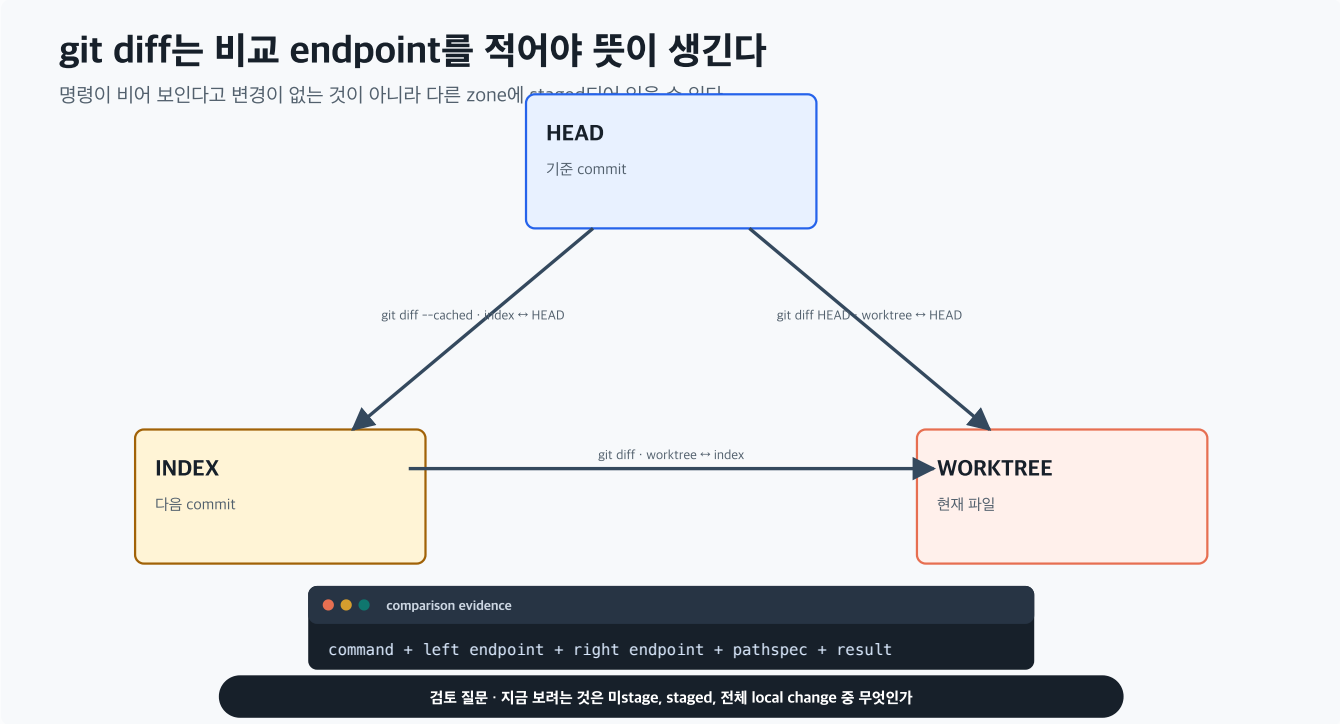
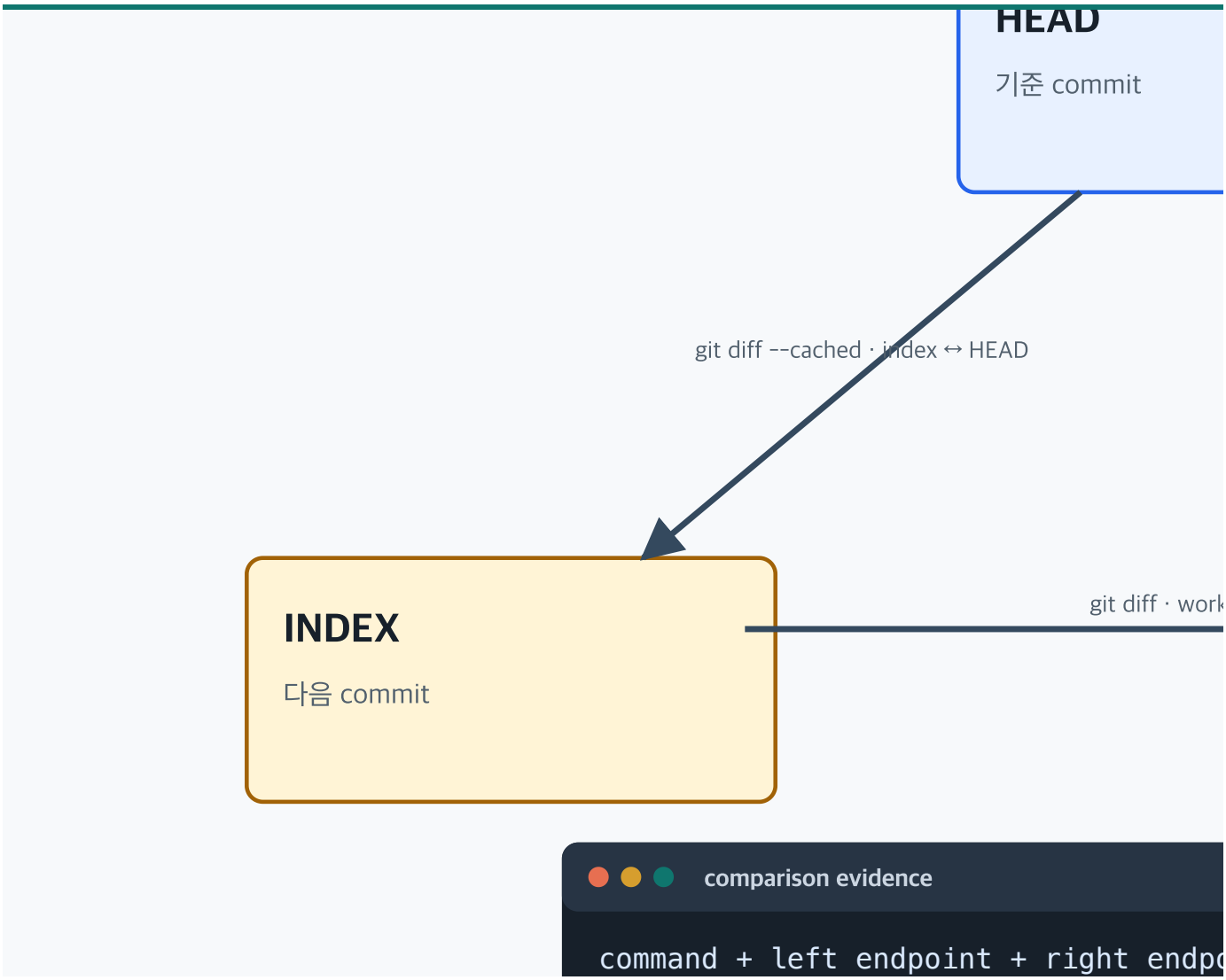
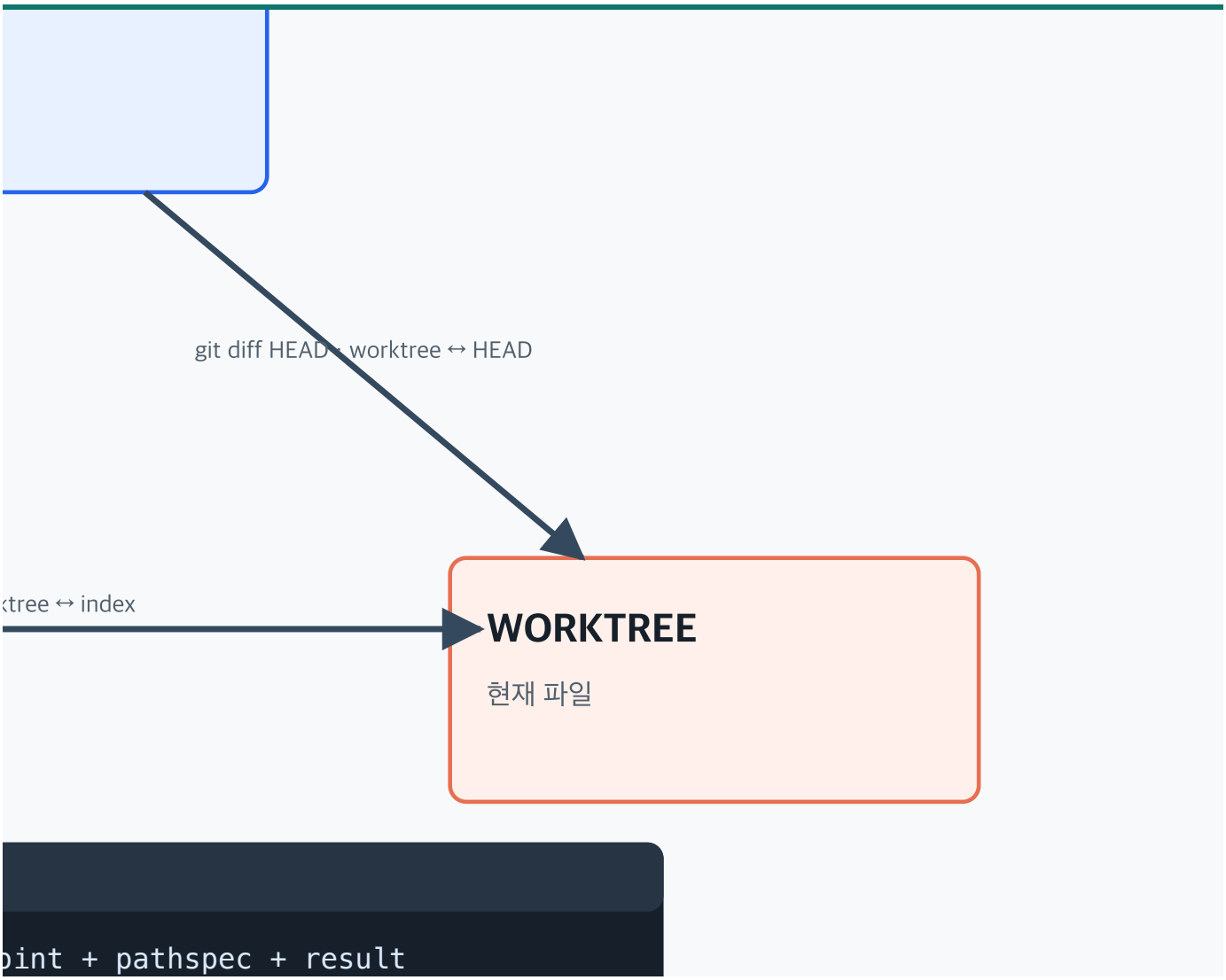


그림 6. 같은 path라도 working tree, index, HEAD 중 어느 두 상태를 비교하는지에 따라 patch가 달라집니다.


HEAD
기준 commit
git diff --cached · index ↔ HEAD
INDEX
다음 commit
git diff · work
comparison evidence
command + left endpoint + right endpo


git diff HEAD : worktree ↔ HEAD
tree ↔ index
WORKTREE
현재 파일
pint + pathspec + result

git diff 공식 문서는 working tree와 index·tree, index와 tree, 두 tree, 두 blob 등 여러 대상을 비교합니다.

6.1. unstaged change

```
git diff -- README.md docs/review-checklist.md
```

기본 `git diff`는 index와 working tree를 비교합니다. `git add`한 뒤 working tree를 다시 수정한 내용도 여기 나타납니다.

6.2. staged change

```
git diff --cached -- docs/review-checklist.md
# --staged는 --cached와 동의
```

HEAD와 index를 비교합니다. 다음 commit에 들어갈 후보 snapshot을 검토하는 핵심 명령입니다.

6.3. HEAD 이후 전체 local change

```
git diff HEAD -- README.md docs/review-checklist.md
```

HEAD snapshot과 working tree를 비교합니다. staged·unstaged를 합친 최종 file 상태 차이를 보지만 어느 부분이 stage됐는지는 status와 두 diff로 나눠 봅니다.

endpoint evidence

command	left	right	질문
`git diff`	index	working tree	아직 stage하지 않은 것은
`git diff --cached`	HEAD	index	다음 commit 후보는
`git diff HEAD`	HEAD	working tree	HEAD 이후 현재 file 전체 차이는
`git diff A B`	A tree	B tree	두 snapshot 최종 차이는
`git diff A...B`	merge-base(A,B)	B tree	갈라진 뒤 B의 변화는

`--` 뒤의 pathspec은 revision과 path를 분리하고 범위를 좁힙니다. 증거에는 명령 전체와 endpoint·pathspec을 함께 남깁니다.

7. patch는 header·hunk·context·삭제·추가로 읽습니다

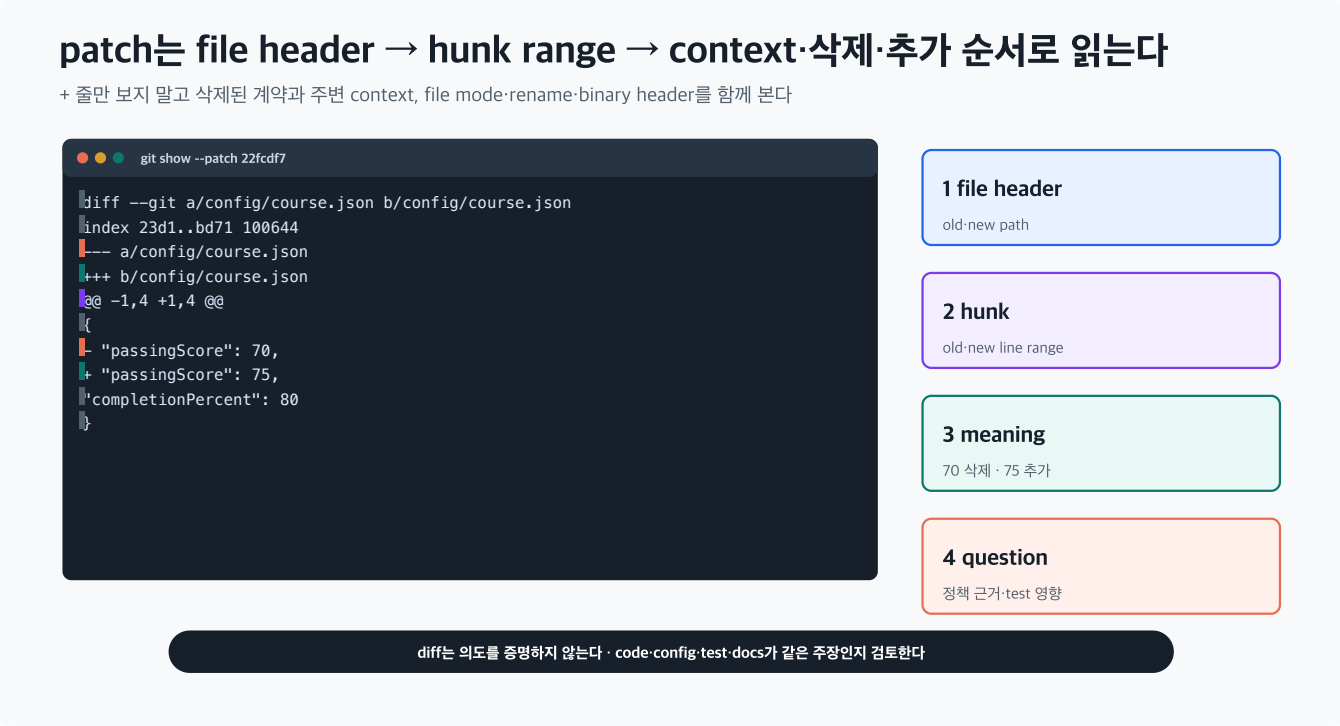


그림 7. `+`만 모아 보지 않습니다. old·new path, mode, hunk 범위, 삭제된 계약, 유지된 context를 함께 읽습니다.

git show --patch 22fcdf7

```
diff --git a/config/course.json b/config/course.json
index 23d1..bd71 100644
--- a/config/course.json
+++ b/config/course.json
@@ -1,4 +1,4 @@
{
- "passingScore": 70,
+ "passingScore": 75,
"completionPercent": 80
}
```

diff는 의도를 증명하지 않는다 · code·co

1 file header

old·new path

2 hunk

old·new line range

3 meaning

70 삭제 · 75 추가

4 question

정책 근거·test 영향

nfig·test·docs가 같은 주장인지 검토한다

```
diff --git a/config/course.json b/config/course.json
index d7a4727..480c210 100644
--- a/config/course.json
+++ b/config/course.json
@@ -1,4 +1,4 @@
 {
-  "passingScore": 70,
+  "passingScore": 75,
   "completionPercent": 80
 }
```

7.1. file header

`a/...`, `b/...`는 비교 양쪽 path를 표시합니다. `new file mode`, `deleted file mode`, file mode change, similarity index, rename from/to, binary 표시도 header에 나타날 수 있습니다.

7.2. hunk header

`@@ -1,4 +1,4 @@`는 old file과 new file에서 이 hunk가 위치하는 line range를 나타냅니다. source file의 영구 line ID는 아닙니다.

7.3. line marker

marker	뜻	검토 질문
space	양쪽에 있는 context	어떤 함수·조건 안입니까
`-`	old에 있고 new에서 제거	기존 계약·fallback이 사라집니까
`+`	new에 추가	입력·오류·권한·test 영향은
`\ No newline...`	파일 끝 newline 차이	formatter·tool 영향은

7.4. patch는 의도가 아닙니다

`passingScore: 75`라는 변화는 보이지만 왜 75인지, 누가 승인했는지, 어디서 소비하는지, 기존 회원에 소급되는지, test가 있는지는 diff만으로 알 수 없습니다. commit message·issue·requirements·test·docs를 교차 확인합니다.

8. 큰 commit은 summary에서 patch로 좁혀 읽습니다

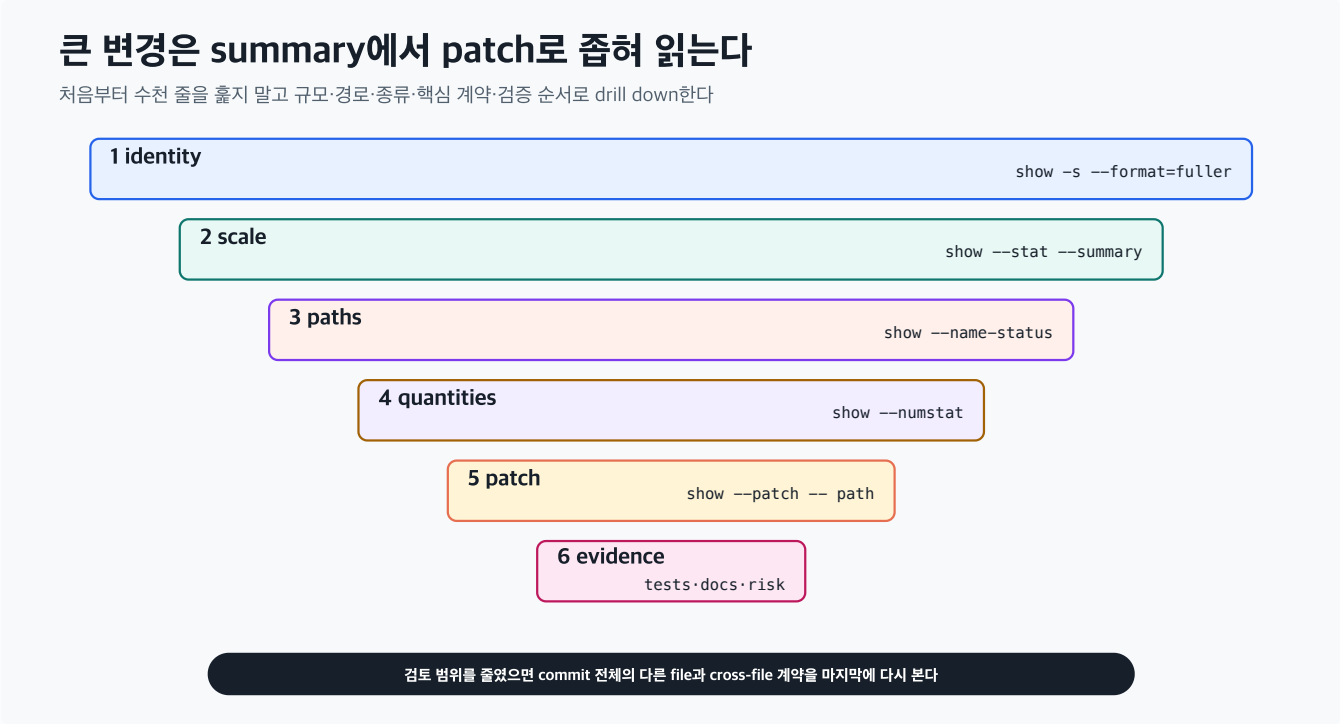
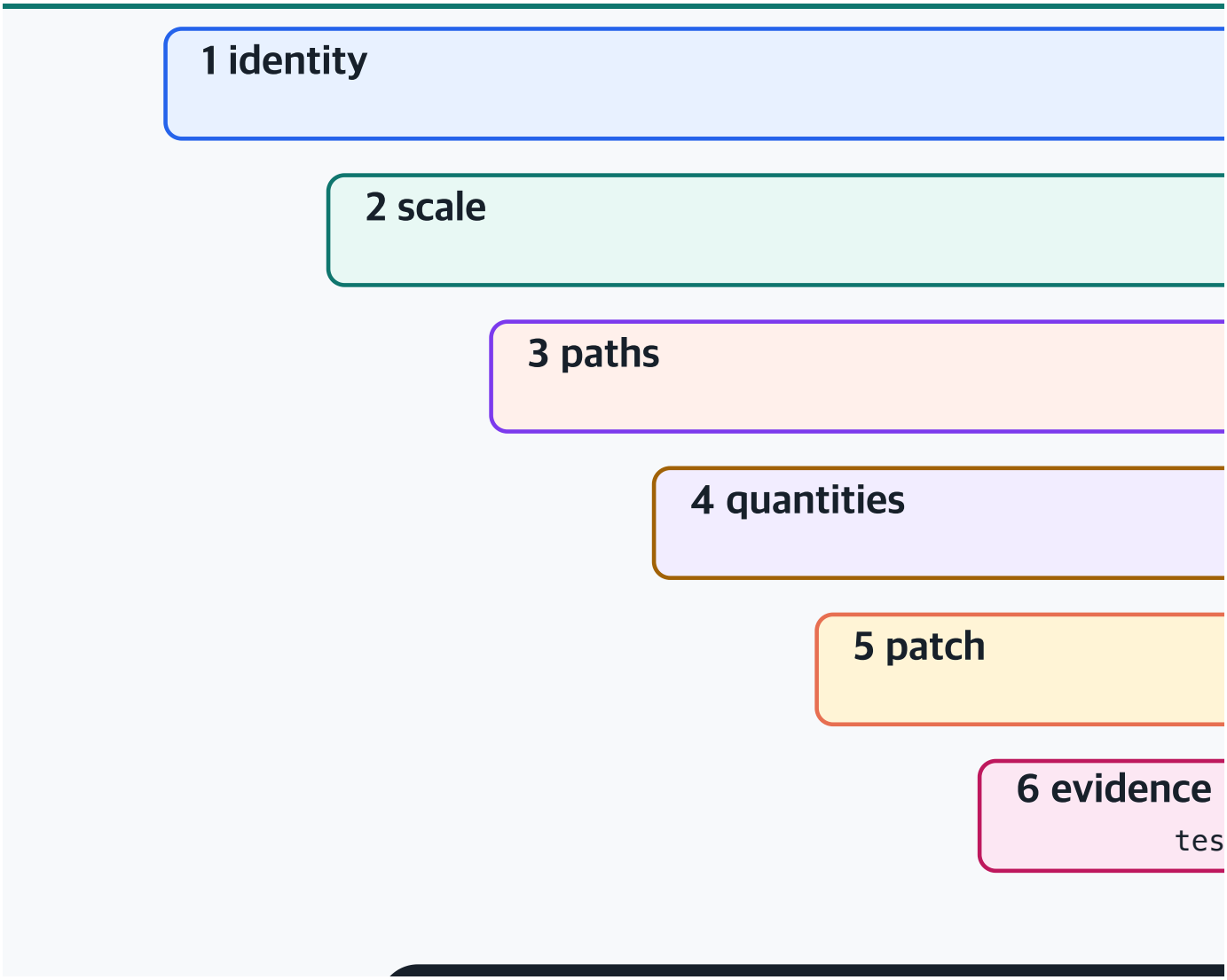


그림 8. 수천 줄 patch에 바로 들어가지 않고 identity·규모·경로·종류·수량으로 범위를 좁힌 뒤 핵심 path를 읽습니다.


1 identity
2 scale
3 paths
4 quantities
5 patch
6 evidence
tes

```
                                        show -s --format=fuller

                                    show --stat --summary

                                show --name-status

                            show --numstat

  show --patch -- path

ts·docs·risk
```

8.1. identity

```
git show -s --format=fuller 22fcdf7
```

8.2. 규모와 file lifecycle

```
git show --stat --summary --format=fuller 22fcdf7
git show --name-status --format='' 22fcdf7
git show --numstat --format='' 22fcdf7
```

option	먼저 보는 것	한계
`--stat`	file별 line change 규모	업무 중요도와 같지 않음
`--summary`	create·delete·rename·mode	text 내용 없음
`--name-status`	path와 A·M·D·R 등	line 수·내용 없음
`--numstat`	added·deleted 수, binary `- -`	의미·의도 없음

8.3. patch

```
git show --patch 22fcdf7 -- config/course.json
git show --patch 22fcdf7 -- docs/change-notes.md
```

8.4. cross-file 주장 확인

연습 commit은 config와 change note가 함께 바뀝니다. 다음을 확인합니다.

- config의 `70 → 75`와 문서의 정책 설명이 같은가
- code가 config를 실제 읽는가
- validation range와 default가 같은가
- test가 경계값 74·75를 검증하는가
- migration·기존 데이터·UI 문구 영향이 있는가

9. history는 parent graph와 ref로 읽습니다

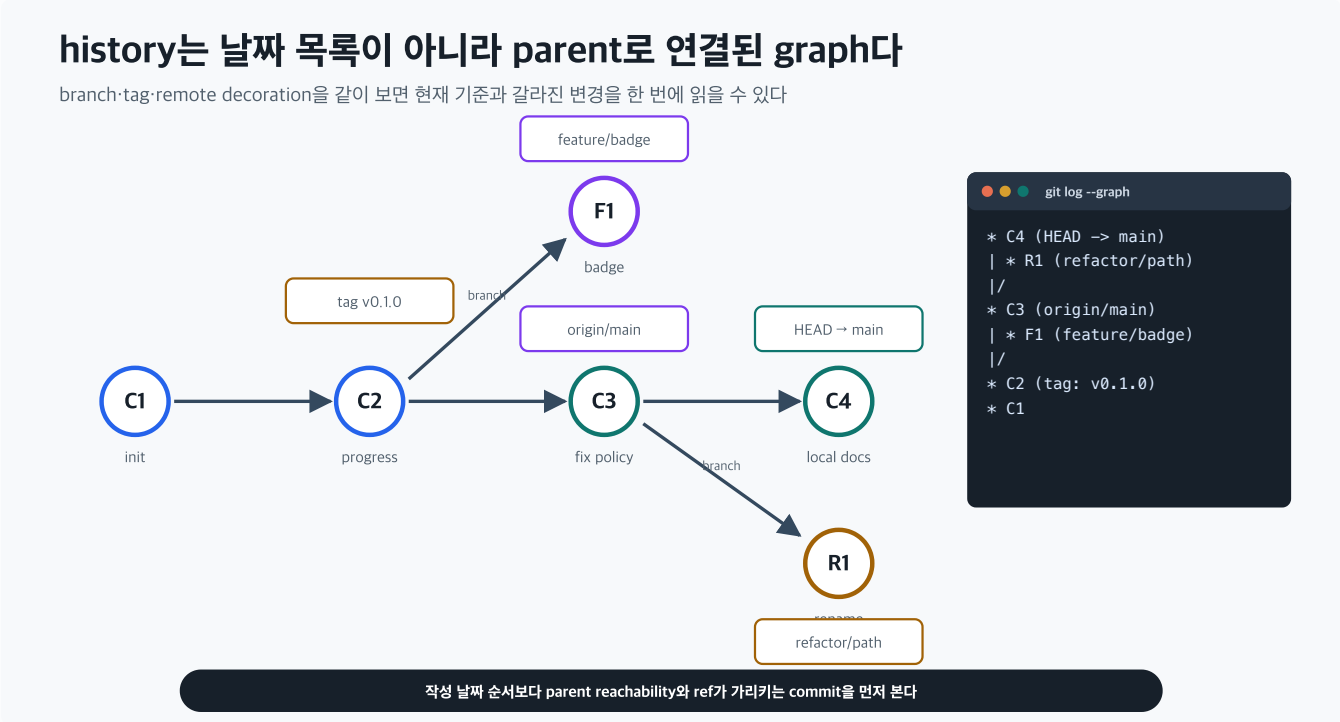
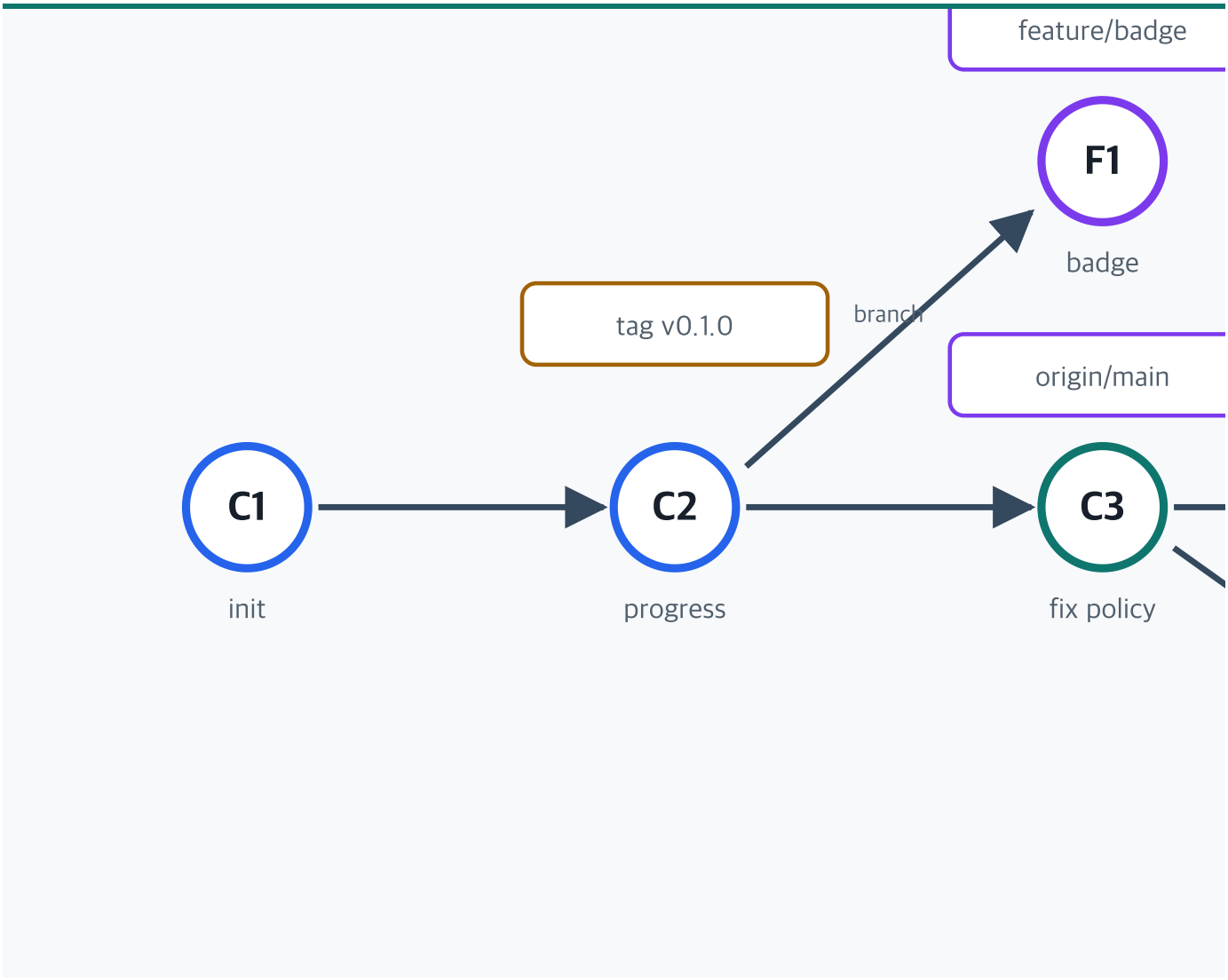
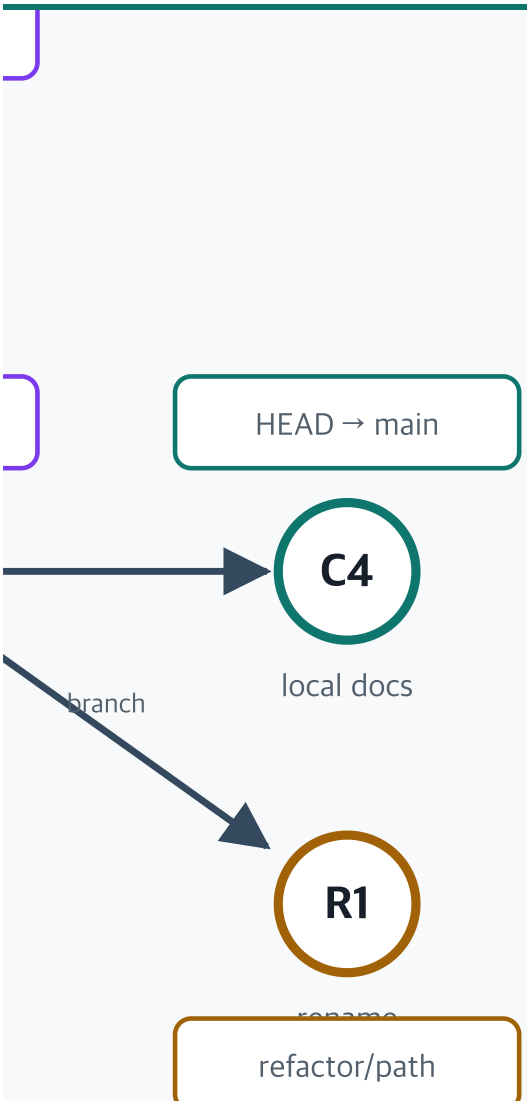


그림 9. `--graph --decorate --all`은 branch가 갈라진 parent 관계와 ref 위치를 함께 보여 줍니다.


feature/badge
F1
badge
tag v0.1.0
branch
origin/main
C1
C2
C3
init
progress
fix policy



git log --graph

```
* C4 (HEAD -> main)
| * R1 (refactor/path)
|/
* C3 (origin/main)
| * F1 (feature/badge)
|/
* C2 (tag: v0.1.0)
* C1
```

```
git log --graph --decorate --oneline --all
```

연습 결과:

```
* 7e0337b (HEAD -> main) docs: add safe inspection checklist
| * d287935 (origin/refactor/catalog-path, refactor/catalog-path) refactor: rename change notes
|/
* 22fcdf7 (origin/main) fix: align passing score with policy
| * 0896413 (origin/feature/completion-badge, feature/completion-badge) feat: add completion badge
|/
* 12f846f (tag: v0.1.0) feat: calculate lesson progress
* 4962d14 chore: initialize course service
```

graph 읽는 순서

1. HEAD와 current branch 위치를 찾습니다.
2. upstream·remote-tracking ref 위치를 찾습니다.
3. tag가 어느 commit을 가리키는지 찾습니다.
4. parent line을 따라 common ancestor를 찾습니다.
5. review 대상 branch tip과 target branch tip을 찾습니다.
6. message를 실제 commit 내용의 주장으로 기록합니다.

특정 file history

```
git log --follow --oneline -- docs/release-notes.md
git log -p --follow -- docs/release-notes.md
```

`--follow`는 한 file의 rename 이전 history를 이어 보는 데 도움을 줍니다. 모든 복잡한 copy·split·merge를 완벽하게 보존하는 file identity database라고 생각하지 않습니다.

text가 들어오거나 사라진 commit 찾기

```
git log -S'passingScore' --oneline --all -- config/course.json
git log -G'passingScore.*75' --oneline --all -- config/course.json
```

`-S`는 문자열의 출현 횟수 변화, `-G`는 diff line이 정규식에 맞는 commit을 찾는 용도로 구분합니다. secret value 자체를 shell history·screen에 다시 노출하지 않습니다.

10. revision과 range의 점을 구분합니다

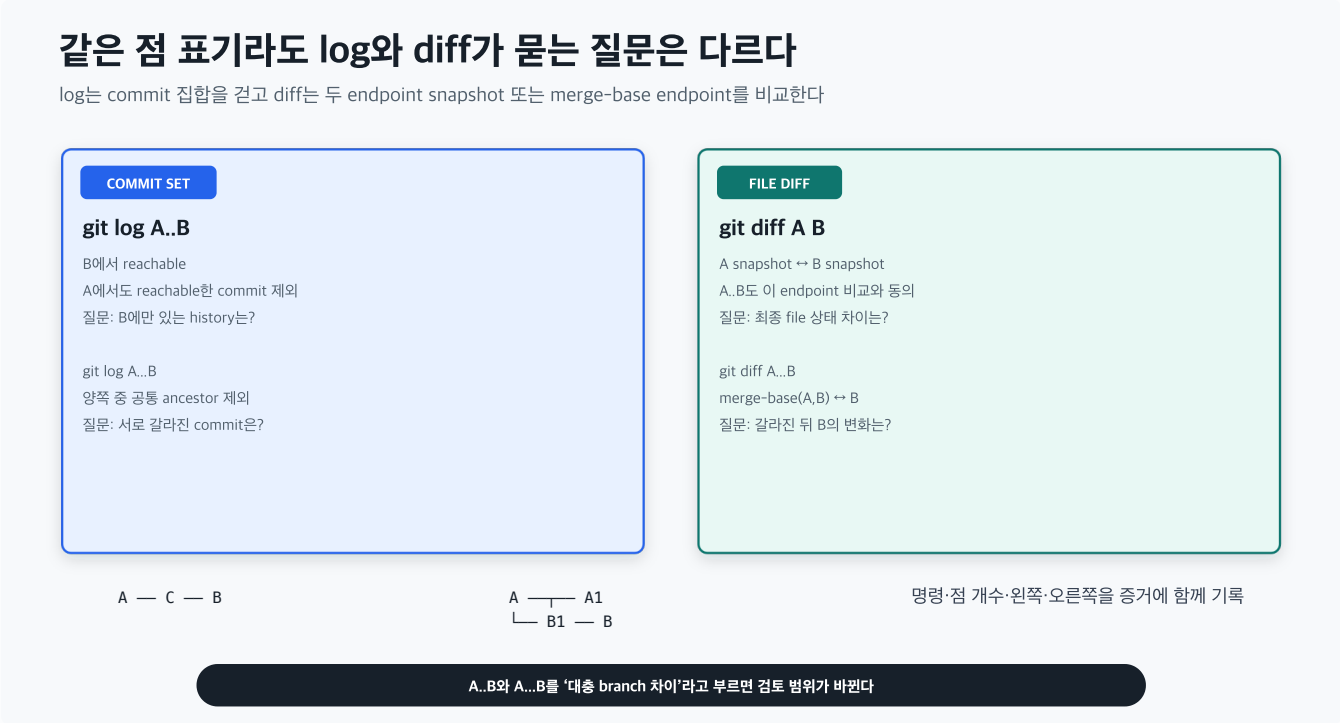


그림 10. history를 걷는 `git log`와 snapshot을 비교하는 `git diff`는 `..`·`...`를 다른 질문으로 사용합니다.

COMMIT SET

git log A..B

B에서 reachable

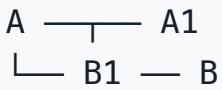
A에서도 reachable한 commit 제외

질문: B에만 있는 history는?

git log A...B

양쪽 중 공통 ancestor 제외

질문: 서로 갈라진 commit은?



FILE DIFF

git diff A B

A snapshot ↔ B snapshot

A..B도 이 endpoint 비교와 동의

질문: 최종 file 상태 차이는?

git diff A...B

merge-base(A,B) ↔ B

질문: 갈라진 뒤 B의 변화는?

명령·점 개수·왼쪽·오른쪽을 증거에 함께 기록

Git revisions 공식 문서는 revision과 reachable commit set을 정의합니다.

10.1. `git log A..B`

B에서 reachable하지만 A에서도 reachable한 commit은 제외합니다. 질문은 “B 쪽에만 있는 commit 집합은?”입니다.

```
git log --oneline origin/main..main
```

10.2. `git log A...B`

A 또는 B에서 reachable하지만 양쪽 공통 history에는 없는 symmetric difference commit 집합입니다. 서로 갈라진 양쪽 commit을 봅니다.

10.3. `git diff A B`와 `A..B`

두 snapshot endpoint를 비교합니다. `git diff A..B`도 이 문맥에서는 두 endpoint 비교와 같습니다. log의 두 점 commit range 의미를 diff에 그대로 옮겨 말하지 않습니다.

10.4. `git diff A...B`

merge-base(A,B)와 B snapshot을 비교합니다. change review에서 feature branch가 common ancestor 이후 만든 최종 file 상태 변화를 보는 데 자주 씁니다.

```
git diff --stat main...feature/completion-badge
git diff --name-status main...feature/completion-badge
```

명령·점 개수·왼쪽·오른쪽을 생략하지 않습니다.
“main과 feature 차이”만 기록하면 commit set인지 snapshot diff인지, target 방향이 어느 쪽인지 재현할 수 없습니다.

11. changed file 종류마다 검토 증거가 다릅니다

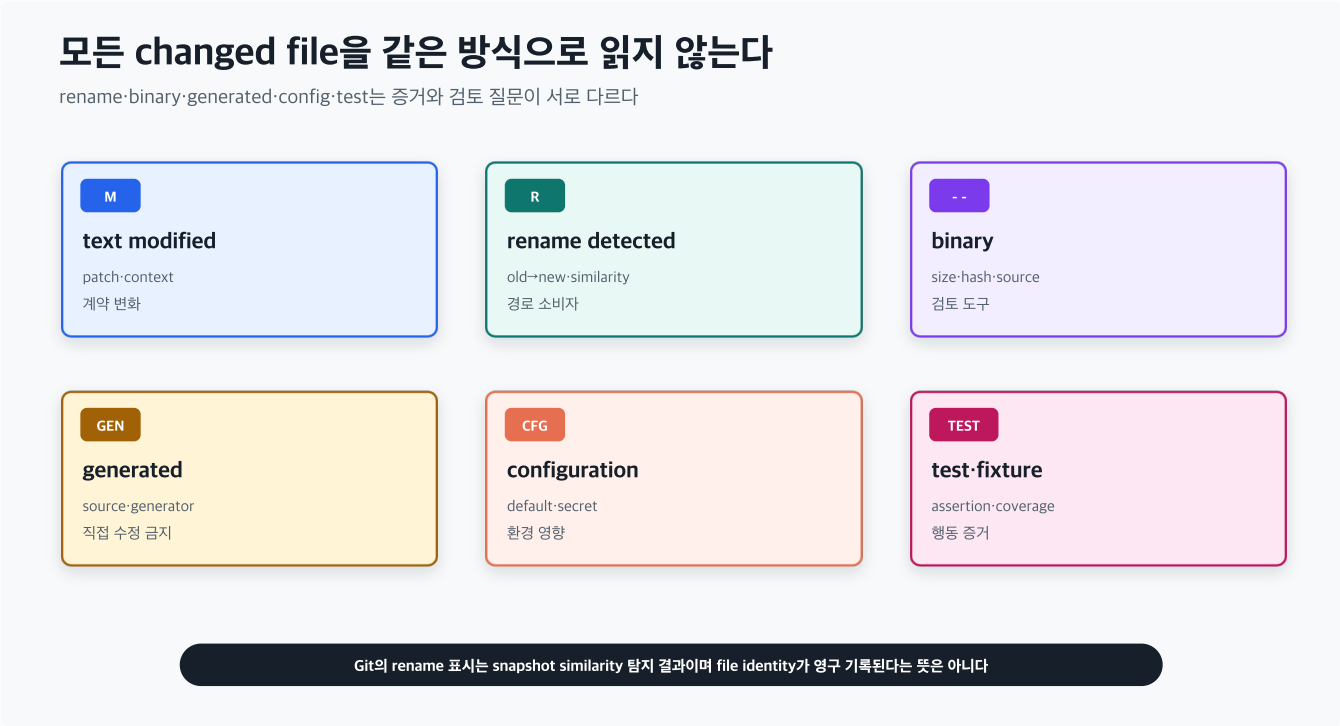


그림 11. line patch로 읽을 수 없는 binary, source에서 재생성해야 하는 generated file, 경로 소비자를 확인해야 하는 rename을 따로 분류합니다.

M

text modified

patch·context

계약 변화

R

rename detected

old→new·similarity

경로 소비자

GEN

generated

source·generator

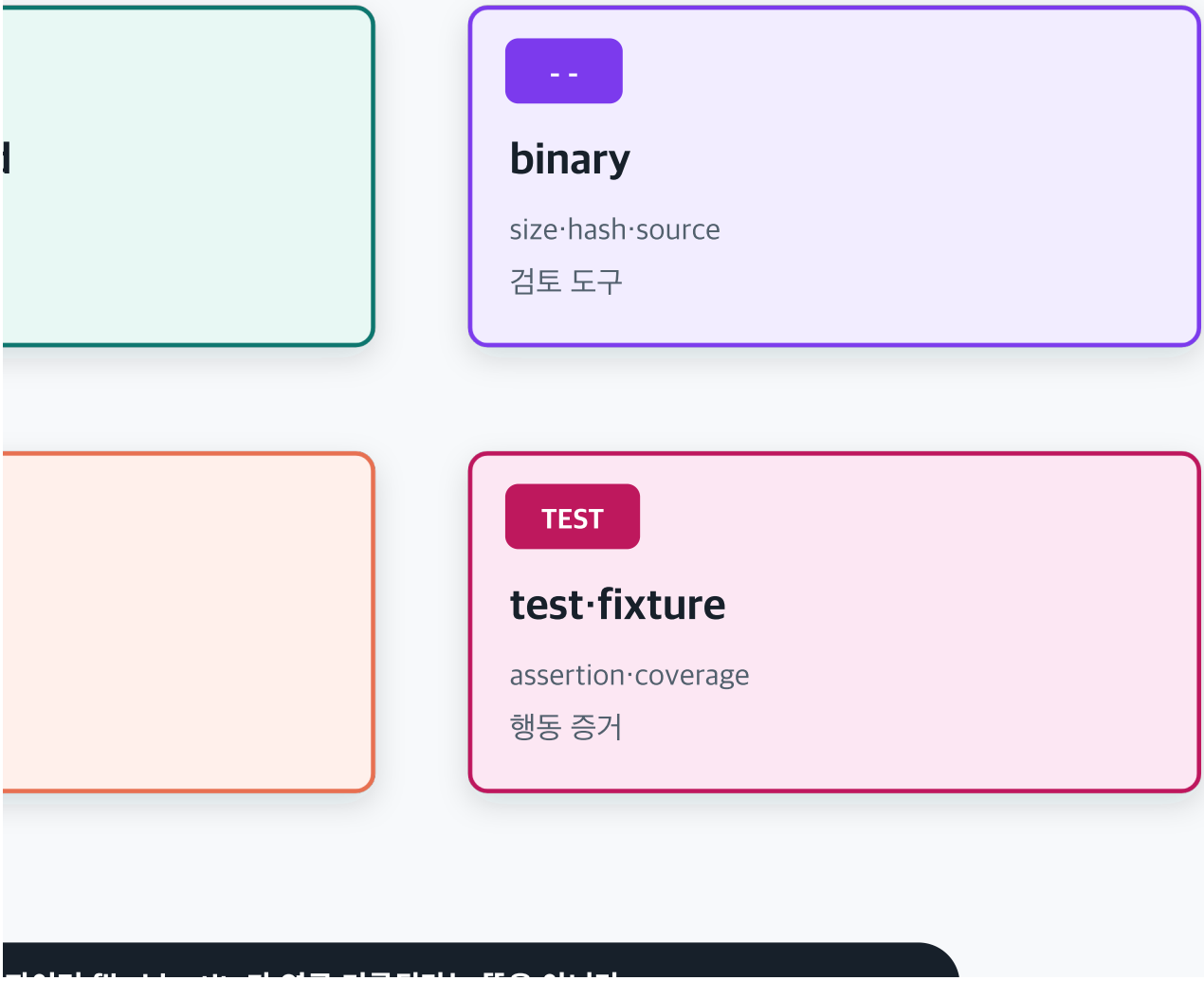
직접 수정 금지

CFG

configuration

default·secret

환경 영향


- -
binary
size·hash·source
검토 도구
TEST
test·fixture
assertion·coverage
행동 증거

11.1. rename

```
git show --summary --find-renames refactor/catalog-path
```

```
rename docs/{change-notes.md => release-notes.md} (63%)
```

63%는 similarity 기반 탐지 결과입니다. old path를 참조하는 link·import·CI·deployment·docs가 깨지지 않는지 확인합니다. 내용 변화가 rename 안에 섞였으면 `--find-renames`와 path별 patch를 함께 봅니다.

11.2. binary

```
git show --numstat --format='' feature/completion-badge
```

```
-  -  assets/completion-badge.bin
```

`- -`는 변경 없음이 아니라 text line 통계를 제공하지 않는 binary입니다. file size, MIME·format, hash, source asset, 생성 tool version, malware scan, visual·functional inspection을 별도 증거로 둡니다.

11.3. generated file

generated file은 어느 source·generator·version·command에서 나왔는지 확인합니다. 사람이 직접 수정한 patch를 정답으로 승인하지 않습니다.

```
source → generator version + command → generated output → reproducibility check
```

11.4. configuration

config 변경은 line 수가 적어도 blast radius가 클 수 있습니다.

- default·fallback·environment override
- unit·timezone·locale
- secret·endpoint·tenant
- restart·migration 필요

- backward compatibility
- production·stage 차이

11.5. test·fixture

test 변경은 기능 증거이면서 동시에 assertion을 약화시킬 수도 있습니다. source behavior가 바뀌어 test가 바뀌는지, test를 통과시키려고 기대값만 바꿨는지 확인합니다.

12. .gitignore와 secret 대응을 구분합니다

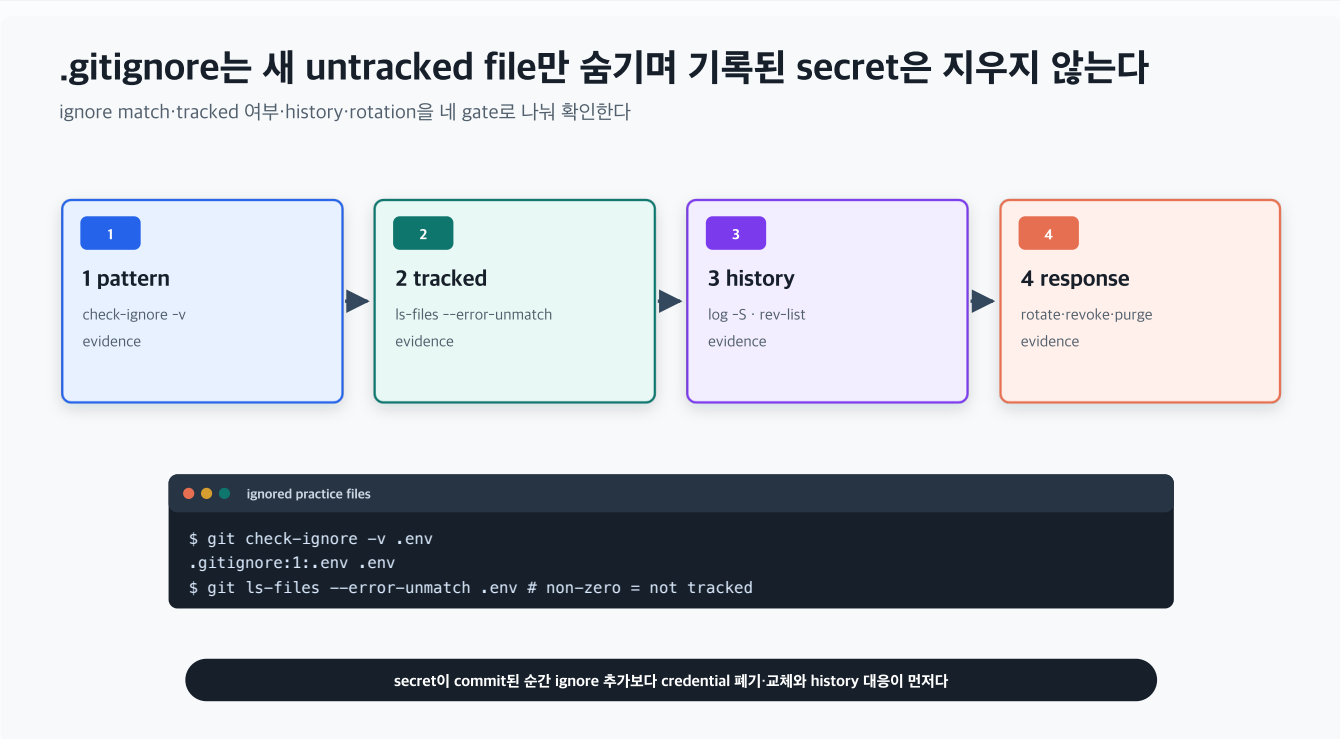


그림 12. ignore pattern match, tracked 여부, history 노출, credential 대응은 서로 다른 네 단계입니다.

ignored practice files

```
$ git check-ignore -v .env
.gitignore:1:.env .env
$ git ls-files --error-unmatch .env # non-zero = not
```


3
3 history
log -S · rev-list
evidence
4
4 response
rotate·revoke·purge
evidence

```
t tracked
```

gitignore 공식 문서는 intentionally untracked file을 ignore하며 이미 tracked된 file에는 영향을 주지 않는다고 설명합니다.

12.1. 어느 pattern이 match했는가

```
git check-ignore -v .env dist/app.js debug.log notes/private/operator.txt
```

`-v`는 pattern source file·line·pattern과 대상 path를 보여 줍니다.

12.2. 이미 tracked됐는가

```
git ls-files --error-unmatch .env
```

exit code가 0이면 index에서 tracked path입니다. `.gitignore`에 추가해도 현재 tracked 상태와 과거 commit이 자동 제거되지 않습니다.

12.3. history에 들어간 적이 있는가

secret의 이름·일반 marker를 찾되 실제 secret 값을 command history에 다시 쓰지 않습니다. repository history 재작성은 다른 clone·branch·tag·remote와 협업에 큰 영향을 주므로 보안 대응 절차와 별도 전문가 검토가 필요합니다.

12.4. 노출 대응

실제 credential이 commit되었다면 다음 우선순위를 검토합니다.

1. credential을 revoke·rotate합니다.
2. 사용·접근 log와 영향 범위를 확인합니다.
3. secret manager·환경변수·권한을 바로잡습니다.
4. 필요하면 승인된 history 정리·remote 대응 절차를 수행합니다.
5. ignore·scanner·pre-commit·CI guard를 보완합니다.

핵심 판정 .gitignore는 예방 보조 장치입니다.

이미 노출된 secret의 폐기·교체를 대신하지 않습니다.

13. 검토 중에는 관찰 명령부터 사용합니다

그림 13. read-only 관찰, remote 정보 갱신, state·history 변경을 나눕니다. 명령 이름이 익숙해도 영향 범위를 확인합니다.

READ

안전한 관찰

rev-parse · status

log · show · diff

ls-tree · ls-files

branch --show-current

SYNC

remote 정보 갱신

remote -v

fetch --prune

remote refs 변화

network·credential

untrusted .git directory의 config·hook은 co

검토 명령이라도 신뢰하지 못한 repository meta

MUTATE

상태·history 변경

add · commit

restore · reset

switch · merge · rebase

clean · push --force

ommand 실행 위험이 있으므로 출처를 확인한다

data 안에서 실행하면 안전하다고 단정할 수 없다.

관찰 중심 명령

질문	명령 예시
어디인가	`git rev-parse --show-toplevel`
현재 기준은	`git rev-parse --short HEAD`, `git branch --show-current`
local 상태는	`git status --short --branch --untracked-files=all`
history graph는	`git log --graph --decorate --oneline --all`
commit identity는	`git show -s --format=fuller <rev>`
변경 규모는	`git show --stat --summary <rev>`
staged·unstaged는	`git diff --cached`, `git diff`
snapshot 차이는	`git diff <left> <right>`
tracked tree는	`git ls-tree -r --name-only HEAD`
ignore 근거는	`git check-ignore -v <path>`

상태를 바꾸는 명령

`git add`, `commit`, `restore`, `switch`, `merge`, `rebase`, `reset`, `clean`, `cherry-pick`, `push`는 index·working tree·refs·history·remote를 바꿀 수 있습니다. 특히 `reset --hard`, `clean -fd`, force push 같은 명령을 “상태를 깨끗하게 만들기” 용도로 무심코 실행하지 않습니다.

이번 매뉴얼은 상태를 바꾸지 않고 설명하는 데 집중합니다. 되돌리기와 AI 생성 변경 검토는 M07-03에서 목적·복구 기준과 함께 다룹니다.

14. 변경 검토는 재현 가능한 evidence packet입니다

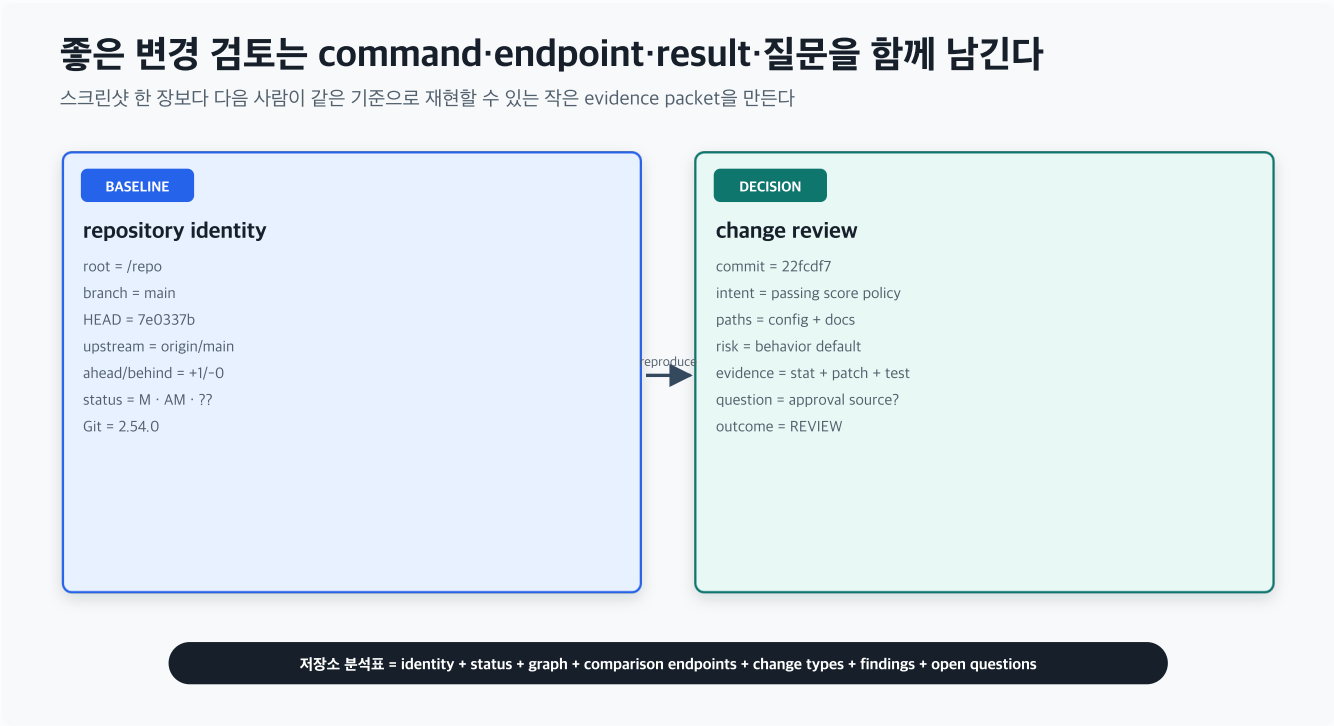


그림 14. command·endpoint·result·question을 한 묶음으로 남기면 다음 사람이 같은 기준으로 변경을 다시 읽을 수 있습니다.

BASELINE

repository identity

root = /repo

branch = main

HEAD = 7e0337b

upstream = origin/main

ahead/behind = +1/-0

status = M · AM · ??

Git = 2.54.0

repro

DECISION

change review

commit = 22fcdf7

intent = passing score policy

paths = config + docs

risk = behavior default

evidence = stat + patch + test

question = approval source?

outcome = REVIEW

14.1. baseline

```
Git version = 2.54.0
repository root = /practice/L06-01/repo
branch = main
HEAD = 7e0337b
upstream = origin/main
ahead/behind = +1/-0
status = README M, review-checklist AM, local note ??
captured_at = 2026-07-15T...
```

14.2. change scope

```
review target = 22fcdf7
left endpoint = 22fcdf7^
right endpoint = 22fcdf7
paths = config/course.json, docs/change-notes.md
change types = text config + docs
scale = 2 files, +2/-1
```

14.3. intent·risk·question

구분	기록
claimed intent	passing score를 승인된 정책과 일치
observed change	config 70→75, change note 추가
user impact	합격·미합격 경계가 바뀔 수 있음
data impact	기존 결과 재계산 여부
test evidence	74·75 boundary test 필요
open question	정책 승인 source와 적용 시점은
outcome	approve / request change / blocked / more evidence

evidence의 한계

- short hash는 repository 안에서 unique해야 하며 장기 외부 기록에는 full ID를 고려합니다.

- working tree가 바뀌면 같은 `git diff` 결과도 바뀝니다.
- remote-tracking ref는 fetch 시각에 따라 바뀝니다.
- author date는 graph 순서와 다를 수 있습니다.
- screenshot은 검색·복사·재현이 어렵기 때문에 command·text result·artifact를 함께 둡니다.

15. Git 증거 리더 실습

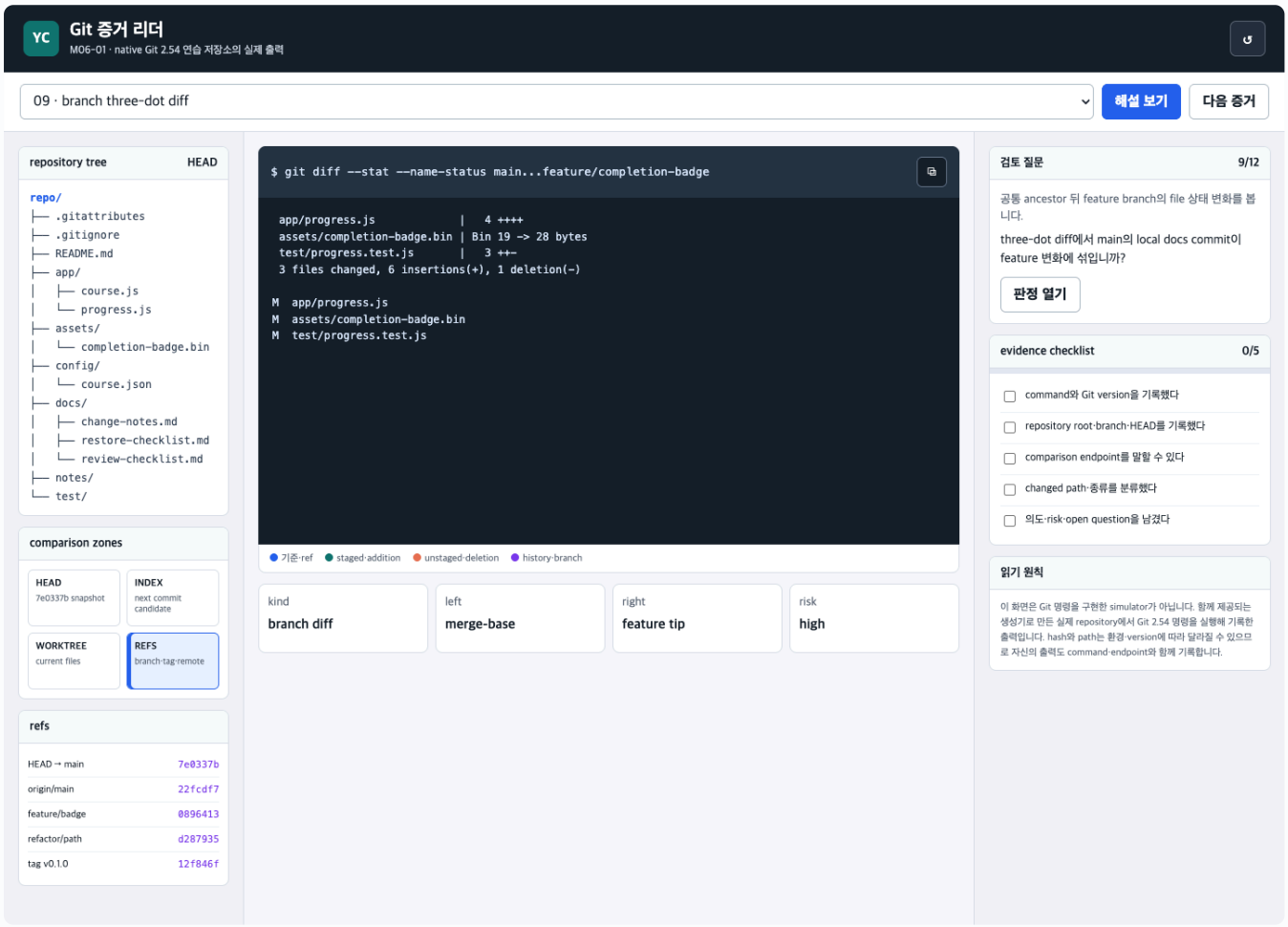


그림 15. 왼쪽 zone highlight로 비교 기준을 가리키고, 실제 command output과 endpoint·risk·검토 질문을 연결합니다.

15.1. 12개 증거

#	증거	먼저 말할 것
1	repository identity	root·branch·HEAD
2	short status	X·Y·ahead

#	증거	먼저 말할 것
3	porcelain v2	branch OID·upstream·AB
4	history graph	ref·common ancestor
5	commit stat	identity·paths·scale
6	text patch	deletion·addition·context
7	unstaged diff	index↔worktree
8	staged diff	HEAD↔index
9	branch three-dot	merge-base↔feature
10	binary numstat	`- -` 와 별도 검증
11	rename summary	similarity·path consumer
12	ignored path	pattern·tracked·history·rotation

15.2. 실습기는 Git을 대신하지 않습니다

화면의 hash와 output은 제공된 생성기를 Git 2.54.0에서 실행한 기록입니다. 내 환경의 값은 달라질 수 있습니다. 실제 결과는 실습 안내서를 따라 기록하고, 낯선 말은 용어집에서 확인합니다.

16. 60분 실전 과제

Mission A · 10분 · baseline

```
git --version
git rev-parse --show-toplevel
git rev-parse --verify HEAD
git branch --show-current
git status --short --branch --untracked-files=all
```

command·exit code·result를 기록합니다. secret 값이 출력되지 않는지 확인합니다.

Mission B · 10분 · graph·refs

```
git log --graph --decorate --oneline --all
git branch --verbose --verbose
git tag --list
git remote --verbose
```

HEAD, current branch, upstream, tag, feature·refactor branch, common ancestor를 표시합니다.

Mission C · 15분 · three diff

```
git diff -- README.md docs/review-checklist.md
git diff --cached -- docs/review-checklist.md
git diff HEAD -- README.md docs/review-checklist.md
```

각 명령의 left·right endpoint와 path별 line 차이를 비교합니다.

Mission D · 15분 · commit review

```
git show -s --format=fuller 22fcdf7
git show --stat --summary 22fcdf7
git show --name-status --format='' 22fcdf7
git show --patch 22fcdf7 -- config/course.json
```

claimed intent, observed change, user·data risk, missing test, open question을 적습니다.

Mission E · 10분 · non-text·ignore

```
git show --numstat --format='' feature/completion-badge
git show --summary --find-renames refactor/catalog-path
git check-ignore -v .env dist/app.js debug.log notes/private/operator.txt
git ls-files --error-unmatch .env
```

binary, rename, generated, ignored, tracked 여부를 구분합니다. `ls-files --error-unmatch .env`의 non-zero exit는 실습의 정상 관찰입니다.

17. 자주 실패하는 읽기 12가지

실패	왜 틀리는가	바꿀 증거
폴더 이름만 기록	상위 repo일 수 있음	top-level·HEAD
branch 이름만 기록	branch ref는 이동	full commit ID
status를 file 목록으로만 읽음	X·Y zone 혼동	HEAD/index/worktree
diff가 비면 clean이라 결론	staged change 누락	status + cached diff
`git diff HEAD` 만 봄	staged·unstaged 분리 안 됨	세 diff
commit message를 사실로 승인	message는 주장	patch·test·issue
`+` line만 읽음	삭제·context 누락	full hunk
stat line 수로 위험 판단	config 1줄도 큰 영향	file type·consumer
log를 날짜 순 목록으로 읽음	parent graph 누락	graph·decorate·all
`..` · `...` 를 섞음	commit set·endpoint 변화	command·left·right
rename을 영구 file identity로 봄	similarity 탐지 결과	old/new consumer
ignore면 secret 안전이라 결론	tracked·history·노출 대응 별도	check-ignore·ls-files·rotation

18. 셀프 테스트 · 먼저 답을 적으세요

문제 1 · identity

변경 검토를 시작할 때 Git version 외에 반드시 기록할 repository 기준 세 가지를 쓰세요.

답: _____

문제 2 · zones

working tree, index, HEAD의 역할을 각각 한 문장으로 쓰세요.

답: _____

문제 3 · snapshot

commit을 “이전 commit과의 patch”라고만 설명하면 부족한 이유를 쓰세요.

답: _____

문제 4 · refs

`main`, `v0.1.0`, `origin/main`, `HEAD`의 차이를 설명하세요.

답: _____

문제 5 · status XY

`AM docs/review-checklist.md`의 X와 Y를 각각 설명하세요.

답: _____

문제 6 · three diff

`git diff`, `git diff --cached`, `git diff HEAD`의 endpoint를 쓰세요.

답: _____

문제 7 · patch

patch에서 `---`, `+++`, `@@`, `-`, `+`, space marker가 각각 무엇을 나타내는지 쓰세요.

답: _____

문제 8 · inspection funnel

큰 commit을 patch 전에 좁혀 읽는 네 command·option을 쓰세요.

답: _____

문제 9 · range

`git log A..B`, `git log A...B`, `git diff A...B`가 각각 묻는 질문을 쓰세요.

답: _____

문제 10 · binary·rename

`--numstat`의 `- -`와 rename `(63%)`를 어떻게 해석하고 무엇을 추가 검증합니까?

답: _____

문제 11 · gitignore

`.env`가 `.gitignore`에 match한다는 사실만으로 알 수 없는 세 가지를 쓰세요.

답: _____

문제 12 · safe inspection

이번 매뉴얼에서 관찰 명령을 먼저 쓰는 이유와 별도 승인 없이 피할 state-changing command 예시 네 가지를 쓰세요.

답: _____

19. 셀프 테스트 정답·해설

정답 1

repository top-level absolute path, current branch, HEAD commit ID입니다. upstream과 captured time까지 기록하면 remote 비교를 더 잘 재현할 수 있습니다.

정답 2

working tree는 현재 디스크의 checked-out file과 local 수정, index는 다음 commit 후보 snapshot, HEAD는 current branch가 가리키는 기준 commit입니다.

정답 3

commit object는 root tree snapshot, parent, author·committer, message를 기록합니다. patch는 두 snapshot을 비교할 때 계산되며 rename·binary·mode·parent graph를 포함한 문맥이 필요합니다.

정답 4

`main`은 움직이는 local branch ref, `v0.1.0`은 보통 특정 object를 표시하는 tag ref, `origin/main`은 마지막 fetch로 본 remote branch의 local remote-tracking ref, HEAD는 현재 작업 기준을 가리키는 symbolic ref 또는 commit 기준입니다.

정답 5

X=A는 HEAD에 없던 새 file content가 index에 staged됐다는 뜻입니다. Y=M은 stage 뒤 working tree가 다시 수정되어 index와 다르다는 뜻입니다.

정답 6

기본 `git diff`는 index↔working tree, `git diff --cached`는 HEAD↔index, `git diff HEAD`는 HEAD↔working tree입니다.

정답 7

`---` old path, `+++` new path, `@@` old·new hunk line range, `-` old에서 제거, `+` new에 추가, space는 양쪽의 context line입니다.

정답 8

예: `git show -s --format=fuller`, `git show --stat --summary`, `git show --name-status --format=''`, `git show --numstat --format=''`입니다. 이후 핵심 path의 patch와 cross-file evidence를 봅니다.

정답 9

`git log A..B`는 B에서 reachable하지만 A에서는 아닌 commit, `git log A...B`는 양쪽 중 공통 history 밖의 symmetric difference commit, `git diff A...B`는 merge-base(A,B)와 B snapshot의 file 차이를 묻습니다.

정답 10

`- -`는 text line count를 제공하지 않는 binary change입니다. size·hash·source·tool·visual·functional 검증을 추가합니다. `(63%)`는 snapshot similarity 기반 rename 탐지 결과이며 old·new path 소비자와 내용 변화를 확인합니다.

정답 11

이미 tracked됐는지, 과거 commit·branch·tag·remote history에 들어갔는지, 실제 credential이 노출·사용됐는지 알 수 없습니다. 노출됐다면 revoke·rotate가 우선입니다.

정답 12

기준 상태를 보존하고 의도하지 않은 working tree·index·ref·history·remote 변경을 막기 위해 관찰부터 합니다. 예: `add`, `commit`, `restore`, `reset`, `clean`, `switch`, `merge`, `rebase`, `push` 중 네 가지입니다.

20. 완료 체크리스트

baseline

- [ ] Git version을 기록했다.

- [ ] repository top-level을 확인했다.
- [ ] branch와 full HEAD ID를 기록했다.
- [ ] upstream과 captured time을 기록했다.
- [ ] status의 staged·unstaged·untracked를 분리했다.

history·diff

- [ ] log graph에서 HEAD·branch·tag·remote ref를 표시했다.
- [ ] common ancestor와 review target을 찾았다.
- [ ] diff마다 left·right endpoint를 적었다.
- [ ] unstaged·staged·HEAD 전체 diff를 구분했다.
- [ ] pathspec으로 범위를 명시했다.
- [ ] patch의 삭제·추가·context를 함께 읽었다.
- [ ] message·patch·test·docs의 주장이 일치하는지 확인했다.

change type·safety

- [ ] text·rename·binary·generated·config·test를 분류했다.
- [ ] binary의 별도 검증 증거를 적었다.
- [ ] rename의 old·new path 소비자를 확인했다.
- [ ] ignore match와 tracked·history를 구분했다.
- [ ] secret 노출 시 rotation이 우선임을 설명했다.
- [ ] untrusted repository의 config·hook 위험을 확인했다.
- [ ] 검토 중 state-changing command를 실행하지 않았다.

evidence

- [ ] command·exit code·result를 기록했다.
- [ ] claimed intent와 observed change를 분리했다.
- [ ] user·data·security·operation risk를 적었다.
- [ ] missing evidence와 open question을 적었다.
- [ ] 최종 outcome과 reviewer를 적었다.

21. 공식 참고 자료

Git current documentation

- Git command overview and object discussion
- Git glossary
- git status
- git diff
- git log
- git show
- git revisions and ranges
- git rev-parse
- git branch
- git tag
- git remote
- git ls-tree
- git ls-files
- git check-ignore
- gitignore
- gitattributes
- Git user manual

버전 기준일: 2026-07-15. Git 공식 current 문서는 2.55.0 계열 페이지를 확인했고, 실습 생성기·12개 command output·repository integrity는 macOS의 Git 2.54.0에서 직접 검증했습니다. hash·absolute path·일부 human-readable output은 Git version·OS·configuration에 따라 달라질 수 있습니다. script가 parse할 때는 porcelain·NUL-delimited 형식 등 공식 stable interface를 검토합니다.