

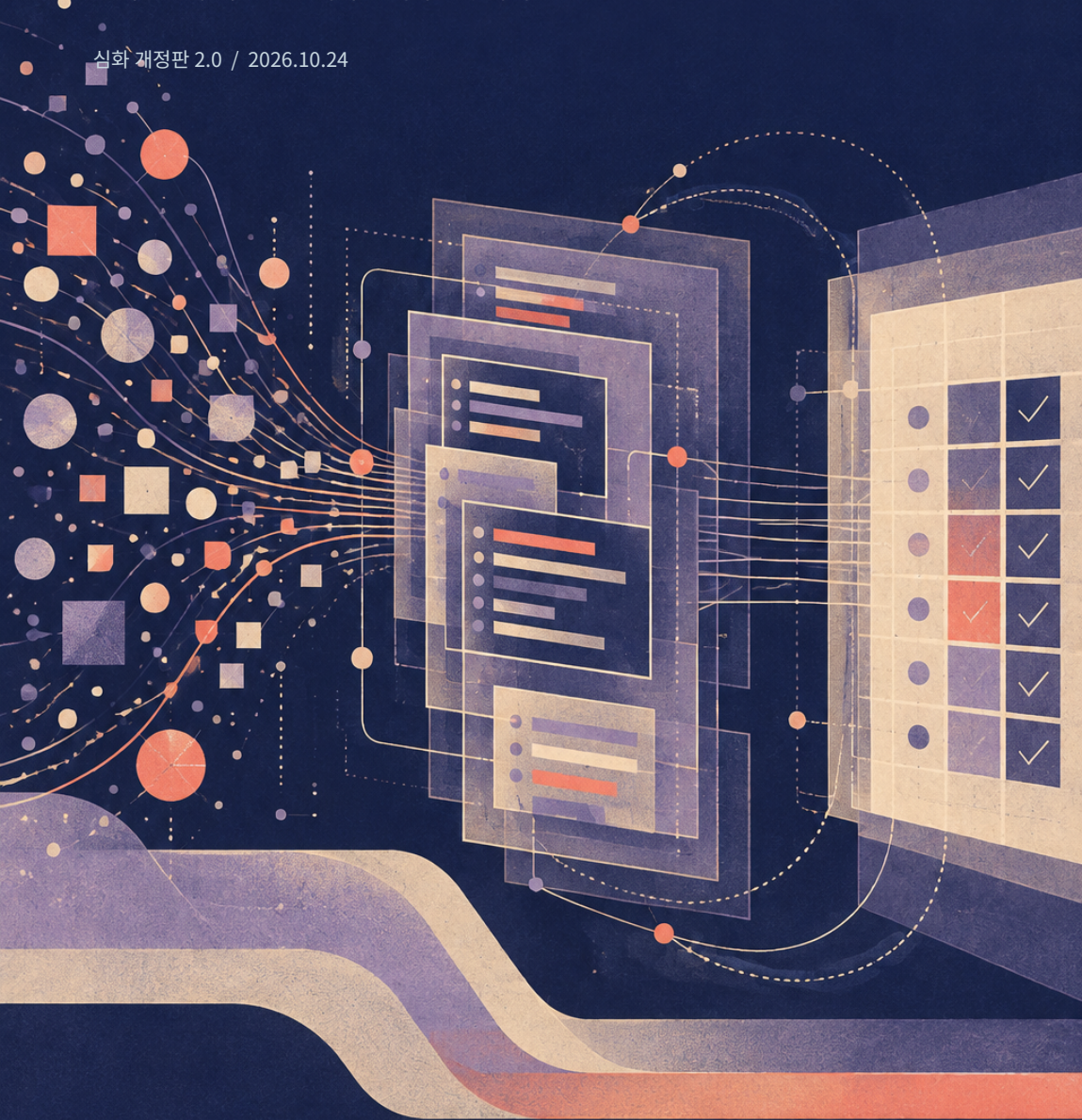
SOFTWARE / PRINCIPLES & PRACTICE

바이브 코딩

원리와 실전

언어 모델부터 소프트웨어 설계와 검증까지

심화 개정판 2.0 / 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.

만드는 속도에서 판단의 깊이로	5
1장 바이트 코딩을 소프트웨어 공학의 언어로 이해하기	6
2장 문제 정의와 요구의 조작화	13
3장 웹 실행 모델과 도구의 책임 경계	20
4장 코드를 읽는 사람의 사고 모형	26
5장 명세를 데이터와 불변식으로 표현하기	30
6장 언어 모델과 맥락 설계의 기술적 기반	38
7장 AI 도구와 작업 방식을 평가하는 실험	44
8장 화면을 만드는 일에서 수직 기능을 완성하는 일로	49
9장 상태 관리와 데이터 일관성	55
10장 API와 분산 시스템의 실패 모델	68
11장 신뢰 경계와 접근 제어	79
12장 테스트와 디버깅의 추론 구조	88
13장 구조 개선과 성능을 함께 판단하기	100
14장 접근성과 보안을 제품 품질로 통합하기	106
15장 AI가 바꾼 코드를 이력으로 통제하기	114

16장 AI와 사람의 결과를 함께 검토하는 협업	133
17장 자동 검사를 반복 가능한 피드백으로 만들기	145
18장 버전 관리와 배포의 재현성	156
19장 의존성과 실행 권한의 공급망을 읽기	162
20장 운영 관찰과 비용 및 복구 설계	177
21장 복원과 인수인계를 실제 절차로 만들기	188
22장 자료 탐색기를 설계 근거와 함께 완성하기	195
23장 공용 업무 요청판의 시스템 설계	204
24장 품질 평가와 지식의 인계	215
원리를 다른 분야로 옮기는 방법	224
프로젝트 기획서와 완료 조건	230
검증 기록과 오류 보고 양식	231
AI 협업 요청과 인계 기록	232
용어 사전 1	233
용어 사전 2	234
자주 묻는 질문 1	235
자주 묻는 질문 2	236
네 번의 실습과 인계 점검	237
자료 탐색기 전체 소스	238

참고문헌과 기술 자료	243
마지막 점검과 인계	256

만드는 속도에서 판단의 깊이로

이 책은 바이트 코딩을 처음 만나는 독자가 자연어로 프로그램을 만들어 보고 그 결과의 구조와 한계를 이해하도록 돕는다. 코드 생성의 편리함을 설명하는 데서 멈추지 않고 명세와 데이터 및 실행과 검증 및 운영의 원리를 연결한다.

공통 사례인 자료 탐색기는 검색과 필터 및 즐겨찾기를 갖춘 실제 정적 앱이다. 이어지는 공용 요청판은 인증과 동시성 및 운영의 요구를 분석하는 설계 사례다. 본문은 이 두 사례를 통해 개인 도구와 공용 시스템의 차이를 설명한다.

각 장은 개념을 정의하고 그 개념이 필요한 이유를 논증한 뒤 작은 코드와 반례로 확인한다. 핵심 함수 사례는 해당 원리를 설명하는 장 안에 배치했다. 순수 함수의 통과를 실제 서버의 보안과 동시성 보장으로 확대하지 않는다.

연구 결과에는 출처와 조건을 함께 제시한다. 프리프린트와 학술대회 논문 및 연구기관 보고와 기술 문서를 구분한다. 이 책은 연구를 해석해 학습에 연결한 해설서이며 동료 심사 논문이나 운영 서비스의 인증 보고서가 아니다.

HTML은 GIF와 실행 예시 및 메모 기능을 포함한다. PDF는 같은 본문을 연속해서 읽도록 구성했다. 모든 예시의 숫자가 실험 결과인 것은 아니며 가상 수치에는 그 의미를 표시한다.

이 책을 단독으로 읽을 수 있도록 Git의 변경 관리와 리뷰 및 CI와 공급망과 복구를 별도 장으로 설명한다. GitHub 가이드북과 공유하는 원리는 AI가 만든 코드를 검토하는 맥락에 맞춰 다시 연결했다.

1장 바이브 코딩을 소프트웨어 공학의 언어로 이해하기

자연어가 개발 인터페이스가 된다는 뜻

바이브 코딩은 자연어로 의도를 전달하고 AI가 제안한 코드를 실행하며 결과를 반복적으로 조정하는 개발 방식을 가리킨다. 다만 이 용어는 학계에서 단일하게 표준화된 개발 방법론이 아니다. 생성된 코드의 내부를 거의 읽지 않고 결과 중심으로 진행하는 좁은 의미와 AI의 도움을 받는 개발 전반을 가리키는 넓은 의미가 함께 쓰인다. 이 책은 처음에는 자연어로 만들기를 시작하되 점차 구조와 증거를 읽는 AI 보조 소프트웨어 개발로 나아가는 학습 경로를 택한다. [R01]

여기서 달라지는 것은 프로그램이 만족해야 할 조건이 아니라 사람이 프로그램을 표현하는 방식이다. 검색창을 만들어 달라는 문장은 구현을 시작하게 할 수 있다. 그러나 제목만 검색하는지 본문도 검색하는지 빈 검색어는 무엇을 뜻하는지 결정하지 않는다. AI는 이 빈자리를 어떤 방식으로든 채워야 한다. 따라서 빠르게 나온 화면에는 사람의 요구뿐 아니라 모델이 추정한 요구가 섞여 있다. 전문적인 작업의 첫 단계는 이 두 종류의 결정을 구분하는 일이다.

이 책의 자료 탐색기는 이 문제를 끝까지 따라가는 공통 사례다. 처음에는 자료 여섯 개를 검색하는 작은 페이지다. 이후 과목 필터와 즐겨찾기를 추가하고 저장 실패와 잘못된 입력을 다룬다. 공용 업무 요청판으로 확장하면 사용자 식별과 권한 및 동시 수정 문제가 등장한다. 같은 자연어 요청이 규모가 커질 때 왜 다른 설계를 요구하는지를 관찰하는 것이 목표다. 기능의 수보다 변경의 이유를 이해하는 데 초점을 둔다.

생성과 실행과 검증은 서로 다른 판단이다

생성은 후보 프로그램을 얻는 과정이다. 실행은 특정 환경과 입력에서 그 프로그램이 어떤 행동을 했는지 관찰하는 과정이다. 검증은 관찰된 행동이 요구와 일치하는지 판단하는 과정이다. 생성에 성공했다는 말은 파일이 만들어졌다는 뜻일 수 있다. 실행에 성공했다는 말은 예외 없이 끝났다는 뜻일 수 있다. 사용자가 원하는 결과를 얻었다는 결론은 그보다 더 강한 주장이다.

예를 들어 자료 탐색기의 검색 결과가 세 개 나왔다고 하자. 화면이 그려진 사실만으로 검색의 정확성을 판단할 수 없다. 기대했던 세 자료인지 확인해야 한다. 빈 검색어에서 전체 자료가 보이는지와 공백만 입력했을 때의 결과도 확인해야 한다. 이처럼 관찰을 해석할 기준을 테스트 오라클이라고 부른다. 오라클은 거창한 프로그램일 필요가 없다. 사전에 작성한 입력과 기대 결과의 표도 좋은 출발점이다.

AI가 구현과 테스트를 동시에 만들면 같은 오해가 두 곳에 복제될 수 있다. 제목 검색이라는 요구를 모델이 본문 검색으로 잘못 이해하면 구현도 본문을 검색하고 테스트도 그 동작을 정답으로 인정할 수 있다. 테스트가 있다는 사실과 요구를 독립적으로 검증한다는 사실은 다르다. 사람은 적어도 핵심 사례의 기대 결과를 구현과 별개로 정해야 한다.

생산성 연구를 읽는 방법

Peng 등의 통제 실험에서는 JavaScript HTTP 서버를 만드는 과제에 Copilot을 제공한 집단이 비교 집단보다 빠르게 과제를 마쳤다. 논문은 55.8%라는 결과를 보고한다. 그러나 측정 대상은 정해진 과제의 완료 시간이다. 조직 전체의 유지보수 비용이나 장기간의 결함률을 직접 측정한 값은 아니다. [S01]

Becker 등의 2025년 연구는 자신이 익숙한 성숙한 오픈소스 저장소에서 일하는 개발자 16명의 과제 246개를 다뤘다. 이 조건에서는 당시 AI 도구를 허용했을 때 완료 시간이 평균적으로 19% 늘었다. 이는 특정 시기와 참여자 및 작업 조건의 결과이며 모든 개발자나 이후 모델의 효과를 뜻하지 않는다. [S02]

METR는 2026년 후속 보고에서 참여자 선택과 과제 선택의 편향 때문에 새 실험의 효과 크기를 신뢰성 있게 추정하기 어렵다고 설명했다. 따라서 앞선 수치를 최신 도구의 고정된 성능처럼 재사용해서는 안 된다. 서로 다른 연구 결과를 읽을 때는 도구 이름보다 과제의 범위와 개발자의 경험 및 측정 단위를 먼저 비교한다. [S03]

이 책이 제안하는 실무적 해석은 간단하다. 초안을 얻는 시간만 재지 말고 요구를 정리한 시간과 결과를 고친 시간 및 검토 시간을 함께 켜다. 자료 탐색기의 첫 화면이 5분 만에 나왔더라도 검색 오류를 찾고 접근성을 고치는 데 2시간이 들었다면 그 전체가 작업 비용이다. 이 시간들은 학습을 위한 관찰 예시이며 특정 도구의 실험 결과가 아니다.

생성 비용이 낮아져도 판단 비용은 남는다

코드를 얻는 시간이 줄어들면 전체 개발 비용도 같은 비율로 줄 것이라고 기대하기 쉽다. 그러나 요구를 명확하게 만드는 시간과 결과를 검토하는 시간 및 실패를 복구하는 시간은 다른 방식으로 변한다. 코드 생성이 매우 빨라도 변경의 의미를 이해하지 못하면 검토와 재작업 비용이 커질 수 있다. 개발 과정을 하나의 생성 속도로 표현하지 않는 이유다.

작업의 총비용을 요구 분석 비용과 생성 비용 및 검증 비용과 운영 비용의 합으로 생각해 볼 수 있다. 이는 보편적인 계량 공식이 아니라 빠뜨린 비용을 찾기 위한 분해다. 한 부분의 비용을 낮추면서 다른 부분으로 비용을 옮기는 선택이 있다. 짧은 프롬프트로 큰 앱을 만드는 방식은 생성 시점의 수고를 줄이지만 이후 구조 파악에 더 많은 시간을 요구할 수 있다.

이 관점에서 바이브 코딩의 숙련은 자연어를 길게 쓰는 능력으로만 평가할 수 없다. 어떤 결정을 맡길 수 있고 어떤 결과를 독립적으로 확인해야 하며 어느 시점에 더 작은 문제로 나눌지를 판단하는 능력이 필요하다. 프로그램의 위험과 되돌림 가능성에 따라 자동화 범위를 조절한다.

불확실성을 줄이는 순서가 개발 순서를 만든다

요구가 불확실한 상태에서 코드의 성능을 먼저 최적화하면 잘못된 목표를 빠르게 수행하는 결과가 될 수 있다. 자료 탐색기에 의미 검색이 필요한지 결정되지 않았다면 먼저 사용자가 무엇을 찾는지 관찰한다. 반면 규칙은 확정되어 있고 느린 입력만 문제라면 성능 측정이 우선이다. 같은 도구를 사용해도 현재의 불확실성이 다르면 다음 행동이 달라진다.

불확실성을 요구와 구현 및 환경과 운영으로 나누어 적을 수 있다. 요구의 불확실성은 사용자 행동을 확인해 줄이고 구현의 불확실성은 작은 시험으로 줄이며 환경의 불확실성은 재현 가능한 실행 조건으로 줄인다. 운영의 불확실성은 관찰과 복구 연습으로 줄인다. 모든 문제를 프롬프트 수정으로 해결하려 하면 적절한 증거를 얻기 어렵다.

AI의 설명이 그럴듯할수록 이 구분이 중요하다. 설명을 얻었다는 사실은 불확실성이 실제로 줄었다는 뜻이 아니다. 다음 실행에서 어떤 결과를 예상할 수 있게 되었는지와 그 예상을 확인할 수 있는지를 본다. 관찰 가능한 차이를 만드는 설명이 개발 판단에 유용하다.

상태 변화 / 요구와 관찰 사이의 반복

요구

제목 검색

후보 코드

없음

실행 관찰

없음

판정

기준 작성

1 / 4 실행 전에 기대 결과를 정한다

생성과 실행 및 검증은 다른 과정이다. 독립적으로 정한 기대 결과가 없으면 잘못된 구현도 성공처럼 보일 수 있다.

요구

규칙 명확화

후보 코드

수정 구현

실행 관찰

기대와 일치

판정

범위 내 통과

4 / 4 검사한 범위를 명확히 기록한다

생성과 실행 및 검증은 다른 과정이다. 독립적으로 정한 기대 결과가 없으면 잘못된 구현도 성공처럼 보일 수 있다.

생성과 실행 및 검증은 다른 과정이다. 독립적으로 정한 기대 결과가 없으면 잘못된 구현도 성공처럼 보일 수 있다.

- 1. 실행 전에 기대 결과를 정한다
- 2. 화면이 나타났다고 정확한 것은 아니다
- 3. 관찰을 기준과 비교한다
- 4. 검사한 범위를 명확히 기록한다

사례 | 보기 좋은 자료실이 쓸 수 없는 이유

교사가 자료실을 만들었다. 검색창은 멋지지만 과목을 바꾸어도 결과가 같았다. AI는 화면 예시만 만들고 검색을 실제 데이터와 연결하지 않았다. 교사는 디자인을 더 고치기 전에 가짜 자료 여섯 개를 넣고 검색 전후의 결과 수를 비교했다. 작은 확인 하나가 다음 작업의 대상을 명확하게 했다.

관찰	아직 말할 수 없는 것	다음 확인
검색창이 보임	검색이 동작함	다른 키워드로 목록 비교
저장 안내가 뜸	새로고침 뒤 유지됨	다시 열어 같은 값 확인
로그인 화면이 있음	권한이 보호됨	다른 계정으로 같은 요청
검사 성공 표시	모든 상황이 정상임	검사의 입력과 범위 읽기

설명할 수 있는 작은 결과

첫 목표는 완벽한 서비스가 아니다. 검색어를 넣으면 제목에 그 말이 있는 자료만 보인다는 한 가지 동작을 완성한다. 사용자가 실제로 볼 결과가 좁혀지면 AI의 변경도 읽기 쉬워진다. 그 다음 과목 필터를 추가하고 마지막에 둘을 함께 시험한다.

이 책의 학습 기록

매 작업마다 요구와 변경과 확인 결과를 짧게 남긴다. 실패한 시도도 원인과 함께 기록한다. 나중에 다른 도구로 옮겨도 프로젝트의 목적과 현재 상태를 다시 설명할 수 있다. 채팅 전체를 기억하는 것보다 현재 결정과 증거를 찾는 일이 중요하다.

적용과 확인 | 작동한다는 말을 구체적으로 쓰기

1. 직접 만들고 싶은 도구 하나를 적는다. 예시는 수업 자료 탐색기다.
2. 주로 사용할 사람 한 종류를 정한다. 예시는 다음 수업 자료를 찾는 교사다.
3. 그 사람이 처음 할 행동과 보고 싶은 결과를 한 문장으로 쓴다.
4. 예시 자료 세 개를 만들어 기대 결과를 손으로 정한다.
5. 아직 넣지 않을 기능 두 가지를 적는다.

기록 항목	예시
첫 행동	과학이라는 과목을 선택한다
기대 결과	과학 자료 한 개만 보인다
확인 방법	과목 표시와 결과 수를 함께 읽는다

결과를 읽는 기준

이 문서를 다른 사람에게 주고 같은 검사를 할 수 있는지 본다. 편리하다와 세련되었다는 말만 있다면 행동과 결과를 더한다. 예시 자료의 정답을 먼저 정하면 AI가 만든 결과에 맞춰 성공 기준을 바꾸는 실수를 줄일 수 있다.

검토 질문 AI가 모든 코드를 썼다면 사람이 배울 것은 없어질까? **해설** 요구의 범위와 데이터 흐름과 권한 및 검증의 판단은 계속 필요하다. 처음에는 작은 결과를 읽는 능력부터 기른다.

검증 기준과 문자열 검색

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x가 입력을 받고 return이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L01 / 제목에서 검색어 찾기

여러 자료 중 제목에 검색어가 있는 항목만 찾는다. 이 사례의 입력 계약은 items는 제목 배열 / query는 검색어다.

```
const q = x.query.trim().toLocaleLowerCase('ko-KR');
return x.items.filter(title =>
  title.toLocaleLowerCase('ko-KR').includes(q)
);
```

분수라는 검색어를 제목마다 대조한다. 분수 활동과 분수 카드는 남고 실험 기록은 빠진다. 검색어가 공백뿐이면 공백을 정리한 빈 문자열을 포함하는 모든 제목이 남는다.

입력

기대 결과

```
{"items": ["분수 활동", "실험 기록", "분수 카드"], "query": "분수"}
```

```
["분수 활동", "분수 카드"]
```

```
{"items": ["분수 활동"], "query": "없는말"}
```

```
[]
```

```
{"items": ["분수 활동"], "query": " "}
```

```
["분수 활동"]
```

해석의 경계. 검색어의 앞뒤 공백과 빈 검색을 별도로 확인한다. 검색 대상을 제목과 설명 중 어디까지 넓힐지 정한다.

분석 사례 L02 / 과목과 검색을 함께 적용

서로 다른 두 조건을 모두 만족하는 자료를 찾는다. 이 사례의 입력 계약은 items는 과목과 제목을 가진 배열 / subject와 query는 조건다.

```
return x.items.filter(item =>
  (!x.subject || item.subject === x.subject) &&
  item.title.includes(x.query.trim())
);
```

제목에 실험이 있는지와 과목이 과학인지 두 번 묻는다. 두 질문이 모두 참일 때만 자료를 남긴다. 수학이면서 제목만 일치하는 자료를 넣으면 AND와 OR를 혼동한 오류를 찾기 쉽다.

입력	기대 결과
<pre>{ "items": [{ "subject": "과학", "title": "실험 기록" }, { "subject": "수학", "title": "분수 카드" }], "subject": "과학", "query": "실험" }</pre>	<pre>[{ "subject": "과학", "title": "실험 기록" }]</pre>
<pre>{ "items": [{ "subject": "과학", "title": "실험 기록" }, { "subject": "수학", "title": "" }], "subject": "수학", "query": "" }</pre>	<pre>[]</pre>

해석의 경계. 두 조건 중 하나만 만족하는 자료를 정답에 넣지 않는다. 학년 조건을 더하고 세 조건의 교집합을 검사한다.

관련 근거 [\[R01\]](#) [\[R02\]](#)

2장 문제 정의와 요구의 조작화

기능 목록보다 먼저 관찰할 행동을 정한다

문제 정의는 만들고 싶은 기능을 길게 나열하는 일이 아니다. 누가 어떤 상황에서 어떤 어려움을 겪는지를 관찰 가능한 문장으로 바꾸는 일이다. 자료를 잘 정리하고 싶다는 표현은 바람을 설명하지만 실패를 판별하기 어렵다. 수업 직전에 특정 과목의 활동지를 찾는 데 여러 폴더를 확인한다는 표현은 작업의 시작과 끝을 보여 준다. 여기서 탐색 시간과 잘못된 선택한 자료의 수 같은 측정 가능성이 생긴다.

추상적인 개념을 관찰할 수 있는 지표로 바꾸는 일을 조작화라고 한다. 사용하기 쉽다는 품질을 자료 하나를 찾는 데 필요한 조작 수로만 표현하면 일부 측면만 포착한다. 익숙한 사용자는 조작이 많아도 빨리 수행할 수 있고 처음 쓰는 사용자는 버튼 하나의 뜻을 이해하지 못할 수 있다. 그러므로 시간과 성공 여부 및 설명이 필요한 지점을 함께 기록해야 한다. 하나의 수치가 전체 경험을 대표한다고 가정하지 않는다.

자료 탐색기의 첫 요구를 다음처럼 정리해 보자. 사용자는 과목을 선택한 뒤 제목 일부로 자료를 찾는다. 검색 결과에는 선택한 과목과 제목 조건을 모두 만족하는 자료만 나타난다. 검색 조건을 지우면 전체 자료로 돌아간다. 이 세 문장은 화면의 색이나 기술 스택을 결정하지 않지만 기능의 의미를 제한한다. AI가 디자인을 자유롭게 제안하더라도 결과의 정확성을 검토할 기준이 남는다.

기능 요구와 품질 요구의 관계

기능 요구는 시스템이 수행할 행동을 말한다. 품질 요구는 그 행동이 어떤 조건에서 어느 수준으로 수행되어야 하는지를 말한다. 즐겨찾기를 저장한다는 요구만으로는 새로고침 이후의 지속 여부나 다른 기기와의 공유 여부를 알 수 없다. 이 브라우저에서 다시 열면 유지된다고 정하면 로컬 저장으로 시작할 수 있다. 계정으로 로그인한 모든 기기에서 유지되어야 한다면 서버 저장과 계정 연결이 필요하다.

품질 요구는 구호가 아니라 측정 절차까지 포함해야 한다. 빠르게 검색된다는 표현 대신 학습용 가짜 자료 1000개를 불러온 상태에서 필터 변경 후 결과가 표시되는 시간을 측정한다고 적을 수 있다. 여기서 1000개는 실제 수요를 대표한다고 검증된 값이 아니라 설계자가 정한 시험 규모다. 이 구분을 남기면 실제 운영 자료가 늘었을 때 재평가할 근거가 생긴다.

요구가 서로 충돌할 때도 있다. 모든 입력을 자동 저장하면 편리하지만 실수로 지운 내용도 곧바로 저장된다. 변경 이력을 모두 보관하면 복원은 쉬워지지만 저장 비용과

개인정보 보존 범위가 늘어난다. 전문적인 설계 문서는 충돌을 숨기지 않는다. 어떤 가치를 우선했으며 다른 가치를 어떻게 보완했는지를 기록한다.

수용 기준을 명세와 테스트 사이의 다리로 사용한다

수용 기준은 변경을 받아들일 수 있는 조건이다. 자료 탐색기의 과목 필터에는 전체와 국어 및 영어라는 선택이 있다고 하자. 국어를 선택하고 제목에 토론을 입력했을 때 국어 자료 중 제목에 토론이 포함된 항목만 보여야 한다. 이 문장은 요구를 구체화하면서 테스트 데이터의 구조도 결정한다. 국어 토론 자료뿐 아니라 영어 토론 자료와 국어 읽기 자료가 함께 있어야 조건을 분리해 검증할 수 있다.

반례를 먼저 설계하면 요구의 빈틈이 드러난다. 제목이 비어 있는 자료는 검색에서 제외할 것인지 목록에서 오류로 표시할 것인지 결정해야 한다. 과목명이 없는 자료를 전체에서 보여 줄지도 정해야 한다. 정상 자료만 놓고 작업하면 이러한 선택은 구현 중에 우연히 이루어진다. 반례는 기능을 방해하는 예외가 아니라 기능의 경계를 정의하는 자료다.

이후 AI에게 보내는 요청에는 문제 상황과 데이터 계약 및 수용 기준을 함께 넣는다. 첫 응답에서는 코드를 요구하기 전에 모델이 이해한 규칙을 요약하게 한다. 요약이 원래 요구와 다르면 아직 구현을 시작할 시점이 아니다. 이 짧은 확인은 장황한 기획 문서를 만드는 목적이 아니라 해석이 다른 상태로 작업이 진행되는 것을 줄이기 위한 절차다.

요청자의 문장을 관찰 가능한 주장으로 바꾼다

사용자가 검색이 편리해야 한다고 말하면 그 문장에는 여러 요구가 들어 있을 수 있다. 검색창을 빨리 찾는 문제와 철자가 조금 달라도 결과를 얻는 문제 및 결과를 기다리는 시간이 짧은 문제는 해결 방법이 다르다. 개발자는 문장을 곧바로 기능 이름으로 바꾸기 전에 실제로 어려웠던 행동을 확인해야 한다. 요구 분석은 사용자의 표현을 교정하는 일이 아니라 경험의 원인을 분리하는 일이다.

자료 탐색기에서 원하는 자료를 찾기 어렵다는 문제를 생각해 보자. 자료 제목을 알고 있는 사용자는 제목 포함 검색으로 충분할 수 있다. 제목을 모르고 수업 목표만 알고 있는 사용자는 과목과 주제 분류 또는 의미 검색이 필요할 수 있다. 같은 검색창을 만들어도 두 사용자의 성공 기준은 다르다. 먼저 지원할 행동을 정하면 현재 버전의 범위와 다음 버전의 과제가 분명해진다.

이때 사용자 요구와 설계 결정을 구분한다. 필요한 자료를 찾는 것은 요구다. 검색어를 소문자로 바꾸어 제목과 비교하는 것은 그 요구를 구현하는 결정이다. 대소문자가 없는 언어에서도 이 결정이 필요한지와 제목 외에 설명까지 검색할지는 별도의 판단이다. 결정을 요구처럼 고정하면 다른 해결책을 비교하기 어렵다. 반대로 모든 요구를 취향 정도로 취급하면 완료 여부를 판정할 수 없다.

추적 가능성은 문서의 양보다 연결의 질이다

추적표는 요구 하나가 어떤 코드와 검사로 이어지는지 보여 준다. 작은 프로젝트에서는 복잡한 도구 없이 표 몇 줄로 시작할 수 있다. 자료 탐색기의 검색 규칙을 Q1이라고 부르고 이를 구현하는 selectItems와 공백 입력 시험을 연결한다. 규칙이 바뀌었을 때 관련 코드와 검사 및 설명을 함께 찾을 수 있다면 추적표가 제 역할을 한 것이다.

요구와 이유	구현 결정	확인할 증거
앞뒤 공백으로 검색이 실패하지 않는다	검색어를 비교 전에 정규화한다	공백을 넣은 입력과 제거한 입력의 결과 비교
과목 선택과 검색이 함께 적용된다	두 조건을 모두 만족하는 항목만 반환한다	두 조건을 각각 만족하는 항목과 함께 만족하는 항목의 대조
다시 열어도 즐겨찾기를 기억한다	ID 목록을 로컬 저장소에 기록한다	저장 후 같은 실행 환경에서 다시 연 결과

표의 마지막 행에서 실행 환경이라는 표현은 중요하다. 다른 브라우저나 다른 주소로 열었는데 즐겨찾기가 공유되지 않는 것은 현재 설계의 경계일 수 있다. 사용자가 기기 간 동기화를 기대한다면 같은 요구를 로컬 저장만으로 충족했다고 판단할 수 없다. 추적표는 구현을 정당화하는 문서가 아니라 요구와 구현의 불일치를 발견하는 도구다.

추적을 지나치게 세밀하게 만들면 유지 비용이 커진다. 화면의 색상 한 줄마다 식별자를 부여할 필요는 없다. 변경이 잦거나 실패 비용이 크거나 여러 곳에 영향을 주는 규칙을 우선 기록한다. 저장 형식과 권한 및 삭제 정책은 작은 변화가 넓게 퍼질 수 있으므로 우선순위가 높다. 기록의 크기는 프로젝트의 위험과 협업 방식에 맞춘다.

결정 기록에는 버린 대안의 조건도 남긴다

결정 기록은 현재 선택과 이유 및 다시 검토할 조건을 짧게 담는다. 예를 들어 첫 버전에서 외부 검색 서버를 사용하지 않고 브라우저 내부 검색을 선택했다면 자료량이 작고 배포와 운영을 단순하게 유지하려는 이유를 적는다. 이후 자료가 많아지거나 여러 사람이 동시에 편집하게 되면 이 결정을 재검토한다. 처음 선택이 잘못되었기 때문이 아니라 선택의 전제가 바뀌었기 때문이다.

시가 세 가지 대안을 제안했다는 사실만 기록하면 이후 판단에 도움이 적다. 대안마다 얻는 이익과 감수하는 비용을 자신의 조건으로 설명해야 한다. 검색 서버를 도입하면 중앙 관리가 가능하지만 네트워크 실패와 인증 및 운영 비용이 추가된다. 정적 파일은

배포가 단순하지만 자료 갱신과 기기 간 공유에 제약이 있다. 이런 교환 관계를 명시하면 전문 용어가 많은 답변보다 실질적인 설계 근거가 된다.

요구 변경은 실패의 표시가 아니다. 초기 가정이 실제 사용과 만나 수정되는 것은 자연스러운 일이다. 다만 변경 이유와 영향 범위를 남기지 않으면 같은 논의를 반복하게 된다. AI와의 대화가 길어져도 결정 기록을 별도로 유지하면 현재 유효한 규칙과 이미 폐기한 제안을 구분할 수 있다.

빠르다는 요구에는 측정 시작점과 종료점이 필요하다

검색이 빨라야 한다는 요구를 검증하려면 시간을 어디에서 재는지 정해야 한다. 키를 누른 시점부터 화면에 결과가 나타나는 시점까지인지 서버가 요청을 받은 뒤 응답을 만드는 시간인지에 따라 값이 달라진다. 사용자가 체감하는 시간에는 네트워크와 렌더링 및 대기열이 포함될 수 있다. 함수 실행 시간만으로 전체 경험을 대표하지 않는다.

자료량과 기기 및 동시 사용 조건도 함께 정한다. 항목 여섯 개의 검색 결과가 빠르다는 사실로 자료 수만 개의 성능을 예측하기는 어렵다. 목표값을 정할 때는 임의의 좋은 숫자보다 실제 사용 행동을 기준으로 삼는다. 수업 중 즉시 찾는 자료와 밤에 한 번 처리하는 통계는 허용할 대기 시간이 다르다.

품질 속성	불명확한 표현	검증 가능한 표현의 요소
성능	빠르게 검색한다	자료량과 기기 및 측정 구간과 목표 분포
신뢰성	오류가 없다	실패로 판정할 사건과 관찰 기간 및 허용 수준
사용성	누구나 쉽게 쓴다	대상 사용자와 과제 및 도움 없이 완료한 행동
유지보수성	수정하기 쉽다	예상 변경과 영향 범위 및 확인에 필요한 작업

품질 요구 사이의 긴장은 설계 근거가 된다

모든 결과를 즉시 서버에서 새로 읽으면 최신성은 높일 수 있지만 네트워크 실패와 응답 시간이 영향을 준다. 로컬 캐시는 빠른 표시를 돕지만 오래된 자료를 보여 줄 수 있다. 최신성과 속도의 관계는 어느 하나가 항상 우월한 문제가 아니다. 어떤 자료가 얼마나 오래되어도 되는지 업무의 의미로 판단해야 한다.

접근성과 시각적 밀도 사이에서도 선택이 필요할 수 있다. 한 화면에 정보를 많이 넣으면 전문가가 빠르게 비교할 수 있지만 처음 쓰는 사람은 핵심 행동을 찾기 어려울 수 있다. 역할과 과제에 따라 기본 화면과 상세 화면을 나눌 수 있다. 전문적인 설계는 모든 요구를 동시에 최대화한다고 약속하기보다 교환 관계를 설명한다.

결정 기록에는 선택한 기준과 검증 방법을 함께 남긴다. 성능 목표를 충족했지만 특정 보조 기술에서 조작할 수 없다면 완료 조건을 다시 검토해야 한다. 하나의 점수가 높다고 필수 조건의 실패가 사라지는 것은 아니다. 합산 가능한 품질과 반드시 만족해야 하는 제약을 구분한다.

사례 | 예약 도구를 만들기 전에 질문을 바꾸다

동아리 담당자는 예약 사이트가 필요하다고 말했다. 더 살펴보니 가장 잦은 문제는 같은 시간에 장비가 중복 예약되는 것이었다. 장식과 프로필보다 시간의 겹침을 확인하고 거절하는 규칙이 먼저였다. 어떤 사람이 어떤 자료에 접근할지의 판단도 뒤따랐다.

요청 표현	먼저 풀 질문
예약 사이트	어떤 자원이 어떤 시간에 겹치는가
설문 사이트	누가 답하고 중복 응답을 어떻게 볼 것인가
업무 대시보드	어떤 결정에 필요한 자료를 보여 주는가
AI 채점 도구	무엇을 판정하고 사람이 무엇을 검토하는가

작은 범위를 정하는 세 가지 선

첫 사용자 한 종류를 정한다. 자료의 종류를 한정한다. 성공 행동을 한 가지로 시작한다. 이 셋을 좁힌 뒤 다른 요구는 대기 목록에 남긴다. 버린다는 뜻이 아니라 첫 결과를 평가할 수 있는 시점을 만드는 것이다.

개발하지 않는 선택도 결과다

기존 표의 열과 필터를 정리하는 것만으로 문제가 충분히 줄어들 수 있다. 프로그램을 만들기 전에 현재 방식으로 해결되는 부분을 살핀다. 새 도구의 편익뿐 아니라 유지할 사람과 비용과 자료 이동의 부담을 함께 생각한다. 이 판단이 구현의 우선순위를 만든다.

적용과 확인 | 문제 정의 한 장

1. 최근 일주일에 반복해서 번거로웠던 상황을 적는다.
2. 실제 사용자를 한 종류로 좁힌다.
3. 현재 해결 순서를 세 단계로 적는다.
4. 새 도구로 줄일 단계 하나를 고른다.
5. 새 기능 없이 개선할 방법도 하나 비교한다.

내 문서에 넣을 항목	작성 안내
상황	언제 어디서 어떤 자료를 다루는가
사용자	역할로 적고 실명은 넣지 않는다
불편의 근거	걸린 시간이나 반복 횟수의 관찰
첫 결과	사용자의 한 행동과 보일 결과
미루는 범위	다음 판본에서 검토할 기능

동료에게 설명해 보기

기능 이름을 먼저 말하지 않고 사용 장면을 설명한다. 상대가 그래서 무엇이 편해지는가를 다시 묻는다면 기대 변화를 더 구체화한다. 문제가 선명해지면 어떤 기술을 쓸지의 선택도 쉬워진다.

검토 질문 요청한 기능이 많으면 더 좋은 첫 설계일까? **해설** 기능 수보다 문제와 연결되는지 및 실제 검증할 수 있는지가 중요하다. 우선순위를 적고 첫 범위를 정한다.

문제 범위와 수용 인원

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x 가 입력을 받고 `return`이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L46 / 모임 신청 정원 계산

정원과 현재 접수 수에 따라 다음 요청의 허용을 계산한다. 이 사례의 입력 계약은 `capacity`와 `used`는 0 이상 정수 / `requested`는 양의 정수다.

```
if (![x.capacity,x.used,x.requested].every(Number.isInteger) ||
    x.capacity < 0 || x.used < 0 || x.requested <= 0) throw Error
    ('정원');
return x.used + x.requested <= x.capacity;
```

현재 19명에 1명을 더하면 정원 20과 같으므로 허용한다. 2명을 더하면 21이므로 거절한다. 실제 동시 신청에서는 이 계산과 저장을 한 조건 아래에서 처리해야 한다.

입력	기대 결과
{"capacity": 20, "used": 19, "requested": 1}	true
{"capacity": 20, "used": 19, "requested": 2}	false

해석의 경계. 계산 결과와 동시에 들어오는 실제 신청 저장은 다른 문제다. 서버에서 정원 검사와 등록을 함께 처리하는 구조를 검토한다.

관련 근거 [\[R01\]](#)

3장 웹 실행 모델과 도구의 책임 경계

파일에서 화면까지 일어나는 일

브라우저는 HTML을 읽어 문서 구조를 만들고 CSS를 적용해 표시 방법을 결정한다. JavaScript는 사건에 반응해 상태와 문서 구조를 바꾼다. 자료 카드 하나가 보인다는 현상 뒤에는 데이터를 읽는 과정과 요소를 만드는 과정 및 화면에 배치하는 과정이 있다. 이 층을 구분하면 화면이 비어 있는 문제를 곧바로 AI의 실패라고 부르지 않고 어느 단계가 멈췄는지 조사할 수 있다. [R05]

소스 파일과 실행 결과는 같은 것이 아니다. 편집기에 새 코드가 있어도 브라우저가 이전 파일을 보고 있을 수 있다. 개발 서버가 다른 폴더에서 실행 중일 수도 있다. 배포 사이트가 이전 커밋을 사용하고 있을 수도 있다. 따라서 오류 보고에는 파일 경로와 실행 명령 및 접속 주소가 필요하다. 무엇을 실행했는지 모르면 같은 코드를 검토하면서도 서로 다른 결과를 이야기하게 된다.

자료 탐색기의 첫 판본은 외부 서버 없이 하나의 HTML로 실행한다. 이것은 배포 아키텍처의 최종 정답이 아니라 변수를 줄이는 학습 설계다. 네트워크 요청과 빌드 도구 및 계정 인증을 잠시 제외하면 데이터와 화면의 관계를 직접 볼 수 있다. 이후 서버 기능을 도입할 때 무엇이 새로 추가되는지 비교할 기준도 생긴다.

프로세스와 환경을 이해해야 재현할 수 있다

터미널의 현재 디렉터리는 명령이 상대 경로를 해석하는 기준이다. 같은 명령이라도 실행한 폴더가 다르면 다른 설정과 파일을 읽을 수 있다. 환경 변수는 프로세스에 전달되는 설정 값이다. 브라우저에 전달되는 공개 설정과 서버에서만 읽는 비밀 설정을 구분해야 한다. 변수 이름에 SECRET이라는 단어를 붙였다는 이유만으로 비밀성이 보장되지는 않는다.

의존성은 실행 환경의 또 다른 부분이다. 직접 작성한 소스가 같아도 라이브러리 버전이 바뀌면 결과가 달라질 수 있다. 잠금 파일은 의존성 해결 결과를 기록해 환경의 변동을 줄인다. 하지만 운영체제와 런타임 및 외부 서비스까지 모두 고정하는 것은 아니다. 재현 가능한 실행은 소스만 보관하는 것보다 넓은 문제다.

AI에게 설치 오류를 물을 때는 운영체제와 런타임 버전 및 실행 명령을 함께 전달한다. 긴 로그 전체보다 최초로 의미 있게 실패한 위치와 직전 단계가 중요할 때가 많다. 단순히 안 된다고 말하면 모델은 흔한 원인을 추정한다. 실제 관찰을 제공하면 가능한 원인의 집합을 좁힐 수 있다.

도구 선택은 기능보다 통제 가능성으로 판단한다

브라우저 기반 생성 도구는 시작 비용이 낮다. 편집기 기반 도구는 파일과 변경 이력을 자세히 볼 수 있다. 에이전트형 도구는 여러 파일을 수정하고 명령을 실행할 수 있다. 이 차이는 어느 도구가 절대적으로 우수한지를 뜻하지 않는다. 과제의 범위와 사용자가 확인할 수 있는 수준이 맞는지를 판단해야 한다.

자료 탐색기를 만드는 초보자는 먼저 결과 파일을 내려받을 수 있는지 확인하는 편이 좋다. 공용 요청판을 운영하는 팀은 접근 권한과 로그 및 변경 검토 기능을 더 중요하게 볼 수 있다. 모델의 응답 품질 외에 데이터 반출 조건과 실행 권한 및 복구 경로도 도구 선택의 일부다. 가격과 제품 기능은 자주 바뀌므로 이 책은 특정 요금제를 고정된 정답으로 제시하지 않는다.

화면이 멈추는 이유를 실행 모델로 설명한다

브라우저의 JavaScript 작업은 이벤트 루프와 실행 큐의 영향을 받는다. 오래 걸리는 계산을 하나의 동기 작업으로 수행하면 사용자의 입력과 화면 갱신이 지연될 수 있다. `async`라는 단어를 함수 앞에 붙였다고 계산이 자동으로 다른 스레드로 이동하는 것은 아니다. `await`는 비동기 결과를 기다리는 흐름을 표현하며 CPU 계산을 무조건 병렬화하는 장치가 아니다. [T15]

자료 탐색기에서 수십 개의 제목을 검색하는 계산은 보통 작지만 수십만 개의 긴 본문을 매 입력마다 처리하면 화면이 느려질 수 있다. 첫 해결책은 무조건 더 복잡한 기술을 넣는 것이 아니다. 어떤 작업이 오래 걸리는지 측정하고 중복 계산과 불필요한 렌더링을 줄인다. 큰 계산을 나누거나 Web Worker를 사용하는 선택은 그다음에 검토할 수 있다.

다음 코드는 실행 순서를 생각하는 작은 사례다. A와 C는 현재 동기 작업에서 실행되고 Promise에 등록한 B는 그 이후에 처리된다. 이 순서를 알면 로그가 코드의 위아래 순서와 다르게 나타나는 상황을 이해할 수 있다. 다만 실제 네트워크 요청 여러 개의 완료 순서는 이 예제처럼 고정되지 않는다.

```
console.log('A');
Promise.resolve().then(() => console.log('B'));
console.log('C');
// 관찰할 순서: A → C → B
```

실행 순서의 이해는 디버깅에서 중요하다. 상태를 바꾼 직후 비동기 저장기 완료되었다고 가정하면 성공 메시지가 너무 일찍 나타날 수 있다. 저장 요청을 시작한 상태와 서버가 저장을 확인한 상태를 분리한다. 사용자는 두 상태의 차이를 알 수 있어야 한다.

비동기는 일을 동시에 끝낸다는 보장이 아니다

비동기 함수는 결과를 기다리는 동안 실행 흐름을 다른 작업으로 넘길 수 있게 한다. 그러나 CPU를 오래 사용하는 동기 계산을 async 함수 안에 넣었다고 자동으로 별도 스레드에서 수행되는 것은 아니다. 브라우저의 긴 계산은 입력 처리와 화면 갱신을 지연시킬 수 있다. 함수에 붙은 문법과 실제 작업이 실행되는 장소를 구분한다.

자료 탐색기의 배열을 순회하는 계산과 원격 서버 응답을 기다리는 작업은 병목의 성격이 다르다. 전자는 자료 구조와 알고리즘 또는 작업 분할을 검토하고 후자는 요청 수와 대기 시간 및 취소와 재시도 정책을 검토한다. 느리다는 같은 현상에서 출발해도 필요한 수정은 달라진다.

동시에 요청을 많이 보내면 전체 시간이 줄 수 있지만 서버 제한과 메모리 및 연결 수를 압박할 수 있다. 동시성의 수를 제한하고 대기 중인 작업을 관찰하면 과부하를 관리하기 쉽다. 모든 Promise를 한 번에 시작하는 코드가 짧다는 이유만으로 적절한 실행 전략이라고 판단하지 않는다.

자원의 소유자와 해제 시점을 정한다

이벤트 리스너와 타이머 및 네트워크 요청은 화면의 생명주기와 연결된다. 화면이 바뀌었는데 이전 타이머가 계속 동작하면 보이지 않는 작업이 누적될 수 있다. 다시 열 때 리스너를 중복 등록하면 한 번의 클릭이 여러 번 처리되는 문제가 생길 수 있다. 생성한 자원을 누가 정리하는지 코드에서 읽을 수 있어야 한다.

작은 정적 파일에서도 대화상자를 반복해서 열고 닫거나 자료를 다시 불러오는 경로를 시험한다. 최초 실행만 확인하면 누적되는 오류를 놓칠 수 있다. 한 번의 성공과 반복 사용의 안정성을 구분한다. 메모리 사용의 증가가 관찰되면 어떤 객체가 계속 참조되는지 조사한다.

이런 문제는 화면의 기능 목록만으로 드러나지 않는다. 동작이 시작되고 멈추는 시점과 자원의 소유권을 설명하는 간단한 상태 표가 도움이 된다. 실행 모델을 이해하면 시가 만든 코드가 왜 두 번째 사용부터 이상해지는지 원인을 좁힐 수 있다.

사례 | 실행 버튼이 같아 보여도 다른 일이다

HTML 파일을 직접 열어 보는 것과 개발 서버를 실행하는 것은 다르다. 첫 방식은 간단한 화면을 확인하기 편하다. 서버 주소로 열면 요청과 경로 및 저장의 동작을 더 일관된 조건에서 살필 수 있다. 빌드 도구가 필요한 프로젝트는 안내된 설치와 실행 절차를 따라야 한다.

```
작업 폴더
index.html
style.css
```

app.js

위 목록은 역할을 설명하는 예시다. 이 책의 첫 실행 예제는 필요한 코드를 한 HTML 파일에 넣어 준비 부담을 줄인다. 프로젝트가 커지면 역할별 파일과 폴더로 나누고 실제 구조를 문서에 남긴다.

도구를 고르기 전 다섯 가지 확인

내 파일을 내려받을 수 있는가. 어떤 권한으로 명령을 실행하는가. 변경을 비교할 수 있는가. 다른 환경에서 다시 실행할 안내가 있는가. 사용량과 과금 주체를 확인할 수 있는가. 첫 결과가 빨리 나온다는 이유만으로 이 질문을 생략하지 않는다.

작업 도구의 에이전트가 테스트를 실행했다면 어떤 명령을 어느 폴더에서 실행했는지도 함께 읽는다.

적용과 확인 | 내 작업 위치를 연결한다

1. 연습용 폴더 하나를 정한다.
2. 편집기에서 같은 폴더를 연다.
3. HTML의 실행 예제를 내려받아 그 폴더에 둔다.
4. 브라우저에서 열고 제목과 자료 목록을 확인한다.
5. 터미널 사용이 가능하면 그 폴더에서 아래 명령으로 로컬 서버를 실행한다.

```
python -m http.server 8000 --bind 127.0.0.1
```

Python 설치와 실행 이름은 환경에 따라 python3일 수 있다. 서버가 실행되면 <http://127.0.0.1:8000>에서 파일을 선택한다. 끝낼 때 터미널에서 Ctrl+C로 중지한다. 이 명령은 로그인이나 공용 데이터베이스를 만드는 명령이 아니다.

관찰	설명할 내용
편집한 제목이 보임	같은 파일을 보고 있는가
주소에 127.0.0.1	내 PC의 로컬 서버인가
서버 중지 뒤 새 요청 실패	실행 과정이 종료됐는가

검토 질문 미리보기 화면을 열었다면 외부 사람도 접속할 수 있을까? **해설** 로컬 보기와 외부 공개는 다르다. 실제 배포 주소와 접근 조건을 별도로 확인한다.

기술 해설 / HTML은 화면의 뜻을 정한다

제목과 문단과 목록 및 입력은 서로 다른 역할을 가진다. 화면에서 비슷하게 보여도 단순상자와 실제 버튼은 기본 동작이 다르다. AI가 만든 화면을 읽을 때 어떤 요소로 어떤 의미를 표현했는지 확인한다.

예문을 여는 동작에는 button을 사용한다. 입력에는 연결된 label을 둔다. 자료 카드의 제목은 문서의 제목 단계 안에서 읽히도록 배치한다.

```
<label for="q">제목 검색</label>
<input id="q" type="search">
<button type="button">조건 초기화</button>
```

기술 해설 / CSS는 공간과 읽는 순서를 만든다

고정한 너비로만 배치하면 작은 화면에서 넘칠 수 있다. 콘텐츠가 자라는 경우와 줄바꿈 및 확대를 함께 본다. 시각적 순서를 바꾸더라도 실제 읽는 순서와 조작 순서가 어긋나지 않는지 확인한다.

자료 카드 두 열이 좁은 화면에서 한 열이 되도록 한다. 긴 제목이 있어도 버튼이 카드 밖으로 밀리지 않아야 한다.

```
.cards { display:grid; gap:16px;
  grid-template-columns:repeat(2,minmax(0,1fr)); }
@media(max-width:600px) {
  .cards { grid-template-columns:1fr; }
}
```

값의 단위와 실행 결과

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x 가 입력을 받고 return이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L14 / 센티미터를 미터로 변환

입력 단위를 표시 단위로 바꾸되 원본을 설명한다. 이 사례의 입력 계약은 cm은 0 이상 숫자다.

```
if (!Number.isFinite(x.cm) || x.cm < 0) throw Error('길이');
return {value: x.cm / 100, unit: 'm'};
```

250cm를 100으로 나누면 2.5m다. 숫자와 단위가 함께 바뀌어야 한다. 원본이 어떤 단위였는지 알 수 없으면 나중에 표시 숫자만 보고 잘못 계산할 수 있다.

입력	기대 결과
<code>{"cm": 250}</code>	<code>{"value": 2.5, "unit": "m"}</code>
<code>{"cm": 0}</code>	<code>{"value": 0, "unit": "m"}</code>

해석의 경계. 단위 글자만 바꾸고 숫자를 그대로 두지 않는다. 다른 길이 단위도 추가하고 역변환을 검사한다.

관련 근거 [\[R02\]](#) [\[R03\]](#) [\[R04\]](#) [\[R05\]](#)

4장 코드를 읽는 사람의 사고 모형

문법보다 먼저 데이터의 경로를 찾는다

처음 코드를 읽을 때 모든 기호의 뜻을 한꺼번에 외우려 하면 흐름을 놓치기 쉽다. 먼저 외부 입력이 어디에서 들어오는지 찾는다. 그 값이 어떤 함수와 조건을 거쳐 어디에 표시되거나 저장되는지 따라간다. 자료 탐색기에서는 검색창의 value가 입력이고 상태 객체의 query가 현재 조건이며 filter의 반환 배열이 계산 결과다. 마지막으로 렌더 함수가 그 배열을 화면 요소로 바꾼다.

이 흐름을 따라가며 각 변수의 값이 언제 바뀌는지 기록한다. 같은 이름의 변수가 다른 범위에 있을 수 있고 객체를 여러 이름으로 참조할 수도 있다. 코드의 줄 순서만 읽는 것보다 데이터의 소유권과 변경 지점을 함께 보는 편이 오류를 이해하기 쉽다. AI에게 코드를 설명해 달라고 요청할 때도 함수 목록의 요약보다 입력 하나가 결과까지 이동하는 경로를 요구할 수 있다.

예를 들어 검색어에 앞뒤 공백이 들어간 경우를 추적한다. 원본 입력은 두 칸의 공백과 분수라는 글자다. 정규화 단계에서 공백을 제거하고 비교에 사용할 문자열을 얻는다. 각 제목을 같은 기준으로 변환한 뒤 포함 여부를 판단한다. 이 설명을 실제 코드의 줄과 연결하면 정규화가 입력 쪽에만 적용되었는지와 제목에도 적용되었는지 확인할 수 있다.

값과 참조의 차이는 수정 범위를 결정한다

JavaScript에서 배열과 객체는 참조를 통해 다룬다. 한 배열을 다른 변수에 대입했다고 독립적인 복사본이 생기는 것은 아니다. 두 이름이 같은 배열을 가리키면 한쪽에서 변경한 결과를 다른 쪽에서도 관찰한다. 정렬 함수가 원본 목록을 바꾸어 초기화 순서가 달라지는 문제는 이 관계를 이해하면 설명할 수 있다.

```
const original = ['영어', '국어'];
const alias = original;
alias.sort();
console.log(original); // 원본도 정렬된 상태.
```

```
const source = ['영어', '국어'];
const sorted = [...source].sort();
console.log(source); // 입력 순서를 유지한다.
```

전개 구문으로 배열을 복사해도 안쪽 객체까지 모두 독립적으로 복제되는 것은 아니다. 자료 객체 배열에서 바깥 배열만 복사한 뒤 첫 객체의 title을 바꾸면 원본 객체도 달라질

수 있다. 얇은 복사와 깊은 복사를 구분해야 한다. 어떤 수준의 복사가 필요한지는 변경하려는 대상과 데이터 소유권에 따라 결정한다.

모든 객체를 깊게 복사하면 안전하다는 결론도 성립한다. 큰 자료에서는 비용이 높고 함수나 특별한 객체의 복사 의미가 복잡할 수 있다. 변경이 필요한 부분만 새로 만들고 공유해도 되는 값은 유지하는 방식을 사용할 수 있다. 중요한 것은 별칭 관계를 의식하고 함수의 입력을 바꾸는지 계약에 적는 것이다.

함수의 이름과 반환값을 계약으로 읽는다

함수 이름이 validate라고 되어 있어도 실제로 무엇을 반환하는지 확인해야 한다. true나 false를 반환하는지 오류 목록을 반환하는지 예외를 던지는지에 따라 호출 코드가 달라진다. 이름과 실제 계약이 어긋나면 개발자는 직관적으로 잘못 사용하기 쉽다. AI가 지은 이름도 검토 대상이다.

부수 효과가 있는 함수는 이름이나 인터페이스에서 그 사실을 드러내는 편이 좋다. 자료를 계산만 하는 selectItems와 화면을 바꾸는 renderItems 및 저장을 수행하는 saveFavorites를 구분하면 읽는 사람이 영향 범위를 예상할 수 있다. 함수 하나가 이 세 일을 모두 수행한다면 작은 수정에도 여러 종류의 실패가 섞일 수 있다.

반환값이 없다는 사실도 의미가 있다. 이벤트 처리 함수는 화면을 바꾼 뒤 아무 값도 반환하지 않을 수 있다. 이를 검색 함수처럼 사용해 결과를 기대하면 undefined를 받는다. 데이터 계산과 사건 처리를 같은 모양으로 취급하지 않는다. AI에게 함수 분리를 요청할 때는 반환값과 부수 효과를 표로 설명하게 하면 계약을 검토하기 쉽다.

오류를 값으로 다룰지 예외로 다룰지 선택한다

잘못된 입력은 프로그램에서 예상할 수 있는 상태다. 제목이 비어 있으면 사용자에게 수정 요청을 보여 줄 수 있다. 반면 파일이 있어야 한다는 내부 가정이 깨진 경우에는 예외로 실행을 중단하는 것이 적절할 수 있다. 모든 실패를 예외로 만들거나 모두 null로 숨기면 호출자가 의미를 구분하기 어렵다.

작은 검증 함수는 성공 여부와 이유를 명시한 결과를 반환할 수 있다. 이렇게 하면 화면이 사용자의 입력 오류를 표시하는 경로를 쉽게 만들 수 있다. 예상하지 못한 오류는 별도의 경계에서 기록하고 사용자에게 안전한 메시지를 보여 준다. 실패를 처리한다는 것은 오류를 보이지 않게 만드는 것과 다르다.

```
function validateTitle(value) {
  if (typeof value !== 'string') {
    return { ok: false, reason: '문자열이 필요합니다' };
  }
  const title = value.trim();
  if (title.length === 0) {
```

```

    return { ok: false, reason: '제목을 입력하세요' };
  }
  return { ok: true, value: title };
}

```

이 함수는 제목의 존재와 자료형만 확인한다. 길이 제한과 금지된 내용 및 저장 권한을 검사하지 않는다. 함수가 보장하는 범위를 적으면 호출자가 부족한 검증을 어디에서 추가해야 하는지 판단할 수 있다. 짧은 함수의 장점은 작은 계약을 명확하게 읽을 수 있다는 데 있다.

추상화의 이익과 비용을 함께 평가한다

추상화는 반복되는 구체적인 구현을 하나의 개념으로 묶어 다루는 방법이다. 모든 카드가 같은 구조로 표시된다면 카드 생성 함수를 둘 수 있다. 하지만 카드 두 개의 차이가 앞으로 더 커질 예정이라면 하나의 함수에 조건을 계속 추가하는 것이 오히려 복잡해질 수 있다. 현재의 중복과 미래의 변경 이유를 함께 본다.

좋은 경계는 변경의 이유를 분리한다. 자료의 저장 방식은 서버 연결 때문에 바뀌고 카드의 배치는 화면 요구 때문에 바뀐다. 두 책임이 강하게 섞여 있으면 한쪽을 바꿀 때 다른 쪽을 이해해야 한다. 저장 함수를 교체해도 필터의 의미가 유지되도록 경계를 두면 변경 비용이 줄 수 있다.

초보자가 전문가의 코드에서 배워야 할 것은 파일 수와 패턴 이름이 아니다. 어떤 결정을 경계 안에 숨겼으며 호출자에게 어떤 약속만 공개했는지를 읽는다. AI에게 전문적으로 리팩터링해 달라고 하기보다 특정 변경을 독립적으로 수행할 수 있게 해 달라고 요청하면 평가 기준이 선명해진다.

코드 읽기의 산출물은 짧은 설명과 반례다

코드를 읽은 뒤에는 입력과 출력 및 외부 영향과 실패 조건을 네 문장으로 설명한다. 이어서 요구를 깨뜨릴 수 있는 입력 하나를 생각한다. 자료가 비어 있거나 ID가 중복되거나 제목이 문자열이 아닌 상황이 그 예다. 코드의 현재 동작과 원하는 동작을 구분해서 적는다.

AI의 설명을 그대로 받아 적는 데서 끝내지 않는다. 설명이 맞다면 어떤 입력에서 어떤 결과가 나와야 하는지 직접 실행한다. 주석이 오래되었거나 설명이 구현을 잘못 읽었을 수 있다. 코드와 설명과 관찰 결과를 대조하는 습관은 특정 언어의 문법을 넘어 다른 프로젝트에서도 사용할 수 있다.

타입은 허용되는 값의 집합으로 생각할 수 있다

문자열이라는 타입은 매우 넓은 값의 집합이다. 요청 상태라는 개념에는 접수와 검토 중 및 완료 같은 제한된 값만 필요할 수 있다. 일반 문자열을 사용하는 것과 제한된 값의 합집합으로 표현하는 것은 가능한 오류의 범위를 다르게 만든다. 타입을 기호의 장식보다 허용되는 상태를 표현하는 도구로 읽는다.

예를 들어 결과 객체에 ok와 value 및 error가 모두 선택적으로 존재하면 ok가 참인데 error만 있는 상태도 표현될 수 있다. 성공일 때는 value가 있고 실패일 때는 error가 있도록 모형을 나누면 모순된 상태를 줄일 수 있다. 그러나 외부 JSON이 그 모형을 따르는지는 실행 시점에 검증해야 한다. 정적 타입과 외부 경계의 검증은 서로 보완한다.

```
type Result<T> =  
  | { ok: true; value: T }  
  | { ok: false; error: string };  
  
function label(result: Result<string>): string {  
  return result.ok ? result.value : result.error;  
}
```

이 예시의 의미는 타입 문법 자체보다 두 상태의 구분에 있다. 사용자는 성공한 결과에서만 value를 읽는 흐름을 확인할 수 있다. 새로운 실패 종류가 추가되면 그 차이를 표현해야 하는지도 판단한다. 모든 실패를 같은 문자열로 묶으면 화면의 다음 행동을 결정하기 어려울 수 있다.

함수의 효과를 읽으면 시험의 경계가 보인다

같은 입력에서 같은 값을 계산하는 함수와 외부 상태를 바꾸는 함수는 검토 방법이 다르다. 계산 함수는 입력과 출력의 관계를 중심으로 읽고 저장 함수는 외부 효과와 실패 시 남는 상태를 함께 읽는다. 반환값이 true라는 사실만으로 데이터가 영구 저장되었다고 판단하지 않는다. 그 값이 어떤 사건의 완료를 뜻하는지 계약을 확인한다.

함수의 이름에 save가 들어 있더라도 메모리 객체만 바꾸는지 로컬 저장소를 갱신하는지 서버의 응답을 기다리는지 다를 수 있다. 호출자가 기대하는 보장이 실제 구현보다 크면 오류가 생긴다. AI에게 함수를 설명하게 할 때도 반환값과 외부 효과 및 완료 시점을 나누어 적게 하면 검토가 쉬워진다.

상태 전이를 표로 읽으면 숨은 효과를 찾을 수 있다. 검색어를 바꾸는 행동이 즐겨찾기를 초기화하는지와 오류가 발생했을 때 이전 결과를 보존하는지를 확인한다. 관련 없는 상태가 함께 바뀐다면 책임이 섞였을 가능성이 있다. 코드의 구조를 바꾸기 전에 그 변화가 요구된 행동인지 판단한다.

5장 명세를 데이터와 불변식으로 표현하기

데이터 모델은 화면 뒤의 약속이다

자료 카드의 제목과 과목과 링크는 단순한 화면 장식이 아니다. 각각 어떤 값이 들어올 수 있는지 약속해야 하는 필드다. 제목은 문자열이며 공백만으로 이루어진 값은 허용하지 않는다고 정할 수 있다. ID는 화면 순서가 바뀌어도 같은 자료를 가리켜야 한다. 링크는 허용한 프로토콜과 형식인지 확인해야 한다. 이 약속을 데이터 계약이라고 부른다.

계약이 없으면 화면 코드는 입력을 추정하며 동작한다. 제목이 언제나 있다고 가정할 코드에서 누락된 필드가 들어오면 검색 중에 오류가 날 수 있다. 숫자로 보이는 문자열을 그대로 더하면 계산이 아니라 문자열 결합이 일어날 수도 있다. 타입 검사는 일부 잘못된 사용을 미리 찾지만 외부에서 들어오는 실제 값이 계약을 만족하는지는 실행 시점에도 확인해야 한다.

불변식은 허용된 모든 상태에서 유지되어야 할 조건이다. 자료 ID는 중복되지 않는다는 조건이나 즐겨찾기에는 존재하는 자료의 ID만 들어간다는 조건이 그 예다. 사용자가 버튼을 누르는 경로마다 다른 규칙을 적용하면 불변식이 쉽게 깨진다. 입력 경계에서 데이터를 정리하고 상태를 바꾸는 함수를 좁은 범위로 모으면 규칙을 검토하기 쉬워진다.

전제 조건과 사후 조건을 연결한다

함수의 전제 조건은 호출 전에 필요한 상태다. 사후 조건은 정상적으로 끝난 뒤 보장할 결과다. 검색 함수의 전제 조건을 유효한 자료 배열과 문자열 검색어라고 정하면 함수 안에서 모든 종류의 외부 오류를 처리할 필요는 줄어든다. 대신 외부 입력을 그 함수에 넘기기 전에 계약을 확인하는 경계가 필요하다.

예를 들어 제목 검색의 사후 조건은 반환된 모든 자료가 검색어 조건을 만족한다는 것만으로는 부족하다. 조건을 만족하는 자료를 빠짐없이 반환해야 한다는 완전성도 필요하다. 첫 조건만 검사하면 항상 빈 배열을 반환하는 함수도 테스트를 통과할 수 있다. 올바른 결과를 정의할 때는 잘못된 것을 포함하지 않는 성질과 필요한 것을 누락하지 않는 성질을 함께 고려한다.

이 구분은 공용 업무 요청판의 권한 설계에도 이어진다. 허용되지 않은 요청을 거부하는 것은 필요하지만 정당한 사용자의 요청을 모두 거부하는 시스템도 유용하지 않다. 보안과 사용성은 한쪽의 성공만으로 평가할 수 없다. 정상 허용 사례와 부정 거부 사례를 함께 명세해야 한다.

상태 전이표를 먼저 쓰면 모호한 기능이 드러난다

요청판의 상태를 접수와 진행 및 완료로 정했다고 하자. 완료된 요청을 다시 진행으로 돌릴 수 있는지와 작성자가 접수 상태를 삭제할 수 있는지는 상태 이름만으로 결정되지 않는다. 각 상태에서 어떤 행동을 누가 수행할 수 있는지 표로 표현해야 한다. 상태 전이는 현재 상태와 사건 및 권한 조건을 입력으로 받아 다음 상태를 결정하는 규칙이다.

AI에게 상태 전이표를 주면 구현의 범위를 더 분명하게 전달할 수 있다. 그러나 표가 정확하다는 보장은 여전히 없다. 실제 업무에서 취소와 반려가 필요한데 표에 빠졌다면 모델은 그것을 자동으로 알아낼 수 없다. 도메인 지식은 사용자의 업무 관찰에서 들어오고 구현 지식은 도구의 도움을 받을 수 있다. 두 지식의 출처를 분리해 기록하는 것이 명세의 신뢰성을 높인다.

필터의 정확성에는 포함과 누락이 모두 들어 있다

자료 집합을 D라고 하고 제목 조건을 Q라고 하며 과목 조건을 S라고 하자. 원하는 결과는 D의 원소 중 Q와 S를 모두 만족하는 원소의 집합이다. 이를 $R = \{x \in D \mid Q(x) \wedge S(x)\}$ 로 쓸 수 있다. 이 표기는 수학적 장식을 위한 것이 아니라 조건 결합을 명확하게 하기 위한 도구다.

반환된 모든 항목이 조건을 만족한다는 요구는 건전성에 해당하는 관점이다. 조건을 만족하는 모든 항목이 반환된다는 요구는 완전성에 해당하는 관점이다. 빈 결과만 반환하면 첫 요구는 만족할 수 있지만 두 번째는 실패한다. 따라서 테스트는 잘못 포함된 항목과 빠진 항목을 모두 확인해야 한다.

자료	제목 조건 Q	과목 조건 S	포함 여부
국어 토론 카드	참	참	포함
영어 토론 카드	참	거짓	제외
국어 읽기 카드	거짓	참	제외
영어 듣기 카드	거짓	거짓	제외

이 표에서 네 행은 임의의 예제가 아니라 두 불리언 조건의 가능한 조합이다. 논리곱을 논리합으로 잘못 작성한 구현은 두 번째와 세 번째 행에서 드러난다. 테스트 데이터를 설계할 때 기능의 논리 구조를 읽으면 사례를 무작정 늘리지 않고도 중요한 오류를 찾을 수 있다.

정적 타입과 실행 시점 검증을 분리한다

TypeScript의 타입 주석은 개발 과정에서 잘못된 사용을 찾는 데 도움이 된다. 그러나 타입 정보가 외부 JSON을 자동으로 검사해 주는 것은 아니다. 컴파일 과정에서 제거되는 타입 주석과 실행 중 실제 값을 확인하는 코드를 구분해야 한다. 외부 응답을 특정 타입이라고 단언하는 것은 검증이 아니라 개발자가 그 값을 믿겠다고 선언하는 것에 가깝다. [T16]

자료 배열을 받아들이는 경계에서는 배열 여부와 각 항목의 필드 및 ID 중복을 확인한다. 내부 계산 함수는 유효한 자료만 받는 계약으로 단순화할 수 있다. 검증을 모든 줄에 중복하는 대신 입력 경계를 분명히 두면 코드의 책임이 읽히기 쉽다. 신뢰 경계가 바뀌는 서버 요청과 파일 가져오기에서도 같은 원리를 적용한다.

구조 검증과 업무 규칙 검증의 역할을 나눈다

JSON Schema는 객체의 필드와 타입 및 필수 항목 같은 구조를 설명하는 방법이다. properties에 필드를 적었다고 그 필드가 자동으로 필수가 되는 것은 아니며 required를 별도로 정한다. 추가 필드를 허용할지도 계약으로 선택해야 한다. 필드가 없다는 상태와 값이 null인 상태도 같지 않다. [T31]

```
{
  "type": "object",
  "properties": {
    "title": { "type": "string", "minLength": 1 },
    "version": { "type": "integer", "minimum": 0 }
  },
  "required": ["title", "version"],
  "additionalProperties": false
}
```

이 스키마를 통과한 제목에도 공백만 들어 있을 수 있다. 길이가 하나 이상이라는 구조 조건과 사람이 읽을 수 있는 제목이라는 업무 규칙은 다르기 때문이다. 스키마를 통과했다고 소유권과 상태 전이 및 중복 정책까지 검증되었다고 판단하지 않는다. 구조 검사 이후의 규칙을 별도로 정의한다.

추가 필드를 거부하는 정책은 오타를 빨리 발견하는 데 도움이 될 수 있다. 반면 클라이언트와 서버의 배포 시점이 다른 시스템에서는 호환성 요구와 충돌할 수 있다. 어느 쪽이 보편적으로 옳은 것이 아니라 경계의 변경 전략에 맞게 선택한다. 검증의 엄격함과 버전 진화의 유연성을 함께 고려한다.

불변식은 개별 입력보다 넓은 약속이다

자료 목록의 ID가 유일하다는 규칙은 특정 입력 하나의 기대값이 아니다. 모든 유효한 상태에서 지켜야 하는 불변식이다. ID가 중복되면 카드 선택과 즐겨찾기 및 수정 대상의 의미가 함께 흔들릴 수 있다. 이런 규칙을 중앙에서 검사하면 각 화면에서 서로 다른 방식으로 오류를 처리하는 일을 줄일 수 있다.

불변식은 저장 직전에만 확인하면 충분한 경우와 변경 중에도 유지해야 하는 경우가 있다. 여러 단계의 갱신 중 일시적인 중간 상태를 외부에서 관찰할 수 있다면 그 상태도 계약에 포함된다. 트랜잭션이나 상태 전이 함수는 허용하지 않는 중간 상태가 공개되지 않게 만드는 도구가 될 수 있다.

AI에게 불변식을 전달할 때는 명사 목록보다 위반 사례를 함께 제시한다. 같은 ID의 자료 두 개가 들어오면 어떤 오류를 반환해야 하는지와 기존 자료를 보존해야 하는지를 설명한다. 규칙과 실패 정책을 함께 주어야 생성된 코드의 행동을 판정할 수 있다.

사례 | 필터 둘을 합칠 때 빠지는 조건

AI가 검색과 과목 선택을 각각 만들었다. 검색창에서는 자료가 잘 좁혀졌지만 과목을 바꾸면 검색어가 무시됐다. 두 기능을 개별로만 검사했기 때문이다. 설계에 조건의 결합을 적으면 어떤 결과가 맞는지 판단할 수 있다.

검색어	과목	기대 결과
비어 있음	전체	예시 자료 여섯 개
실험	전체	실험이 제목에 있는 자료
비어 있음	과학	과학 자료
실험	과학	두 조건을 모두 만족
없는제목	전체	결과 없음 안내

성공과 실패를 함께 설계

정상 입력만 있는 화면은 실제 사용에서 쉽게 막힌다. 빈 값과 긴 값 및 중복 동작을 생각한다. 모든 예외를 한꺼번에 구현할 필요는 없지만 이번 범위에서 어떻게 다룰지 기록한다. 조건을 모른 채 AI가 임의로 자료를 지우거나 저장하도록 두지 않는다.

변경의 영향을 적는다

필드 하나를 바꾸면 입력 화면과 목록 및 저장과 내보내기가 영향을 받을 수 있다. 화면만 고치고 기존 자료를 읽지 못하는 문제가 생기지 않도록 영향을 받는 위치를 함께 살핀다.

적용과 확인 | 구현 전에 검사 표를 만든다

1. 자료 ID와 제목 및 과목을 가진 가짜 자료 여섯 개를 준비한다.
2. 정상 입력과 빈 입력 및 결과 없는 입력을 정한다.
3. 기대 결과 수를 직접 계산한다.
4. 한 가지 조건과 조건의 조합을 모두 적는다.
5. 이 표를 AI에게 주고 아직 구현하지 말고 모호한 부분부터 찾게 한다.

요청 예시

목표: 제목 검색과 과목 필터가 함께 작동하는 자료 탐색기
범위: 가짜 자료 여섯 개와 브라우저 화면
완료: 아래 검사 표의 결과 수와 제목이 맞아야 함
제외: 로그인과 서버 저장과 외부 API
먼저 데이터 구조와 미정인 규칙을 설명해 줘.

요청문의 줄을 복사하는 것보다 자신의 검사 표를 넣는 일이 중요하다. 이 표를 바꾸면 구현이 맞는지의 기준도 바뀌므로 변경 이유를 남긴다.

검토 질문 디자인 시안과 요구 문서는 같은가? **해설** 시안은 배치와 표현을 보여 준다. 데이터와 동작 및 예외와 완료 기준은 추가로 설명해야 한다.

기술 해설 / 값과 자료형을 먼저 확인한다

숫자처럼 보이는 문자열과 숫자는 서로 다른 값이다. 빈 입력을 숫자로 바꾸면 의도하지 않은 0이 될 수 있다. 형식을 확인하고 필요한 변환을 한 뒤 업무 규칙을 검사한다.

관찰값 입력에서 빈 칸과 실제 0을 다르게 처리한다. 날짜 문자열도 사용 형식을 정한 뒤 비교한다.

```
const raw = '  ' ;  
const value = raw.trim() ? Number(raw) : null;
```

입력 제약과 데이터 검증

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 `x`가 입력을 받고 `return`이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L26 / 빈 제목과 긴 제목 검증

공백뿐인 제목과 지나치게 긴 입력을 거절한다. 이 사례의 입력 계약은 `title`은 문자열 / `limit`은 최대 코드 포인트 수다.

```
if (typeof x.title !== 'string') return {ok:false, reason:'문자열'
};
const title = x.title.trim();
if (!title) return {ok:false, reason:'빈 제목'};
if ([...title].length > x.limit) return {ok:false, reason:'길이'};
return {ok:true, title};
```

자료라는 제목의 앞뒤 공백을 제거한다. 빈 문자열은 다른 오류로 다루고 너무 긴 값도 구분한다. 코드 포인트의 수와 화면에서 하나로 보이는 글자 묶음의 수가 다를 수 있다.

입력	기대 결과
<code>{"title": " 자료 ", "limit": 10}</code>	<code>{"ok": true, "title": "자료"}</code>
<code>{"title": " ", "limit": 10}</code>	<code>{"ok": false, "reason": "빈 제목"}</code>
<code>{"title": "가나다", "limit": 2}</code>	<code>{"ok": false, "reason": "길이"}</code>

해석의 경계. 유니코드 코드 포인트와 사용자가 보는 글자 묶음은 다를 수 있다. 오류 안내와 입력 보존을 화면에서도 확인한다.

분석 사례 L27 / 허용 범위의 숫자 입력

문자열을 숫자로 바꾸기 전에 빈 입력을 거절한다. 이 사례의 입력 계약은 `raw`는 문자열 / `min`과 `max`는 허용 범위다.

```
if (typeof x.raw !== 'string' || !x.raw.trim()) return null;
const value = Number(x.raw);
return Number.isFinite(value) && value >= x.min && value <= x.max
  ? value : null;
```

빈 칸을 먼저 찾아 숫자 변환을 중단한다. 문자열 0은 실제 0으로 받아들이고 11은 허용 범위 0부터 10을 벗어나므로 null을 돌려준다. null을 숫자 0처럼 저장하지 않는다.

입력	기대 결과
<code>{"raw": "0", "min": 0, "max": 10}</code>	0
<code>{"raw": "", "min": 0, "max": 10}</code>	null
<code>{"raw": "11", "min": 0, "max": 10}</code>	null

해석의 경계. Number가 빈 문자열을 0으로 바꾸는 경우를 먼저 차단한다. 소수 자릿수와 정수 여부의 정책도 추가한다.

분석 사례 L28 / 중복 ID 가져오기 거절

같은 ID가 두 번 있으면 반영 전에 거절한다. 이 사례의 입력 계약은 items는 비어 있지 않은 문자열 ID를 가진 배열다.

```
const ids = x.items.map(i => i.id);
return ids.every(id => typeof id === 'string' && id.trim().length > 0) &&
  new Set(ids).size === ids.length;
```

가져올 ID를 모두 모아 개수와 중복 제거 뒤의 개수를 비교한다. 두 값이 다르면 같은 ID가 있다는 뜻이다. 현재 보관한 자료와 충돌하는지는 이 검사 다음에 따로 판단해야 한다.

입력	기대 결과
<code>{"items": [{"id": "a"}, {"id": "b"}]}</code>	true
<code>{"items": [{"id": "a"}, {"id": "a"}]}</code>	false
<code>{"items": [{"id": ""}]}</code>	false

해석의 경계. 유효한 형식과 현재 보관 자료와의 충돌은 다른 검사다. 새 파일 전체를 검증한 뒤 기존 자료와 합친다.

분석 사례 L29 / 가져올 자료의 기본 형식

배열 안 모든 항목이 필요한 필드를 갖는지 확인한다. 이 사례의 입력 계약은 items는 ID와 title을 가진 배열로 기대다.

```
return Array.isArray(x.items) && x.items.every(i =>
  i && typeof i === 'object' && !Array.isArray(i) &&
  typeof i.id === 'string' && i.id.trim() &&
  typeof i.title === 'string' && i.title.trim()
) ? true : false;
```

전체가 배열인지 확인한 뒤 각 항목에 ID와 제목이 있는지 읽는다. 필요한 필드가 빠지면 자료 전체를 반영하지 않는다. JSON의 문법을 읽은 단계와 이 업무 형식의 검증 단계를 구분한다.

입력	기대 결과
<code>{"items": [{"id": "a", "title": "책"}]}</code>	true
<code>{"items": [{"id": "a"}]}</code>	false
<code>{"items": "잘못된 값"}</code>	false

해석의 경계. JSON으로 읽힌다는 사실만으로 서비스의 자료 형식이 맞는 것은 아니다. 추가 필드 허용과 파일 크기 제한 및 버전 정책을 더한다.

관련 근거 [\[R06\]](#)

6장 언어 모델과 맥락 설계의 기술적 기반

모델은 왜 그럴듯한 코드를 생성하는가

언어 모델은 입력 문맥을 바탕으로 다음 토큰의 분포를 계산하며 출력을 이어 간다. 토큰은 사람이 생각하는 한 단어나 한 글자와 반드시 일치하지 않는다. 코드에서는 식별자와 구두점 및 공백도 문맥의 일부다. 모델이 생성한 함수 이름이 자연스러워 보이더라도 실제 라이브러리에 존재한다는 보장은 없다. 생성의 유창성과 외부 사실의 정확성은 별도로 평가해야 한다.

Transformer의 attention은 입력 표현 사이의 관계를 계산하는 핵심 구성 요소다. Vaswani 등의 원 논문은 이를 기계 번역 과제에서 제시했다. 오늘날 코딩 도구의 구체적인 구조와 학습 방식은 제품마다 다르므로 그 논문만으로 특정 서비스의 내부 동작을 단정해서는 안 된다. 이 책에서는 attention을 문맥 사이의 관련성을 계산하는 기반 개념으로 사용한다. [S04]

소프트웨어 개발에서 중요한 함의는 모델이 프로젝트의 모든 사실을 자동으로 알고 있지 않다는 점이다. 현재 파일과 과거 대화 및 도구가 읽은 결과 중 무엇이 실제 입력에 포함되는지에 따라 답이 달라진다. 모델에게 이미 설명했다고 생각한 규칙이 작업 맥락에서 빠져 있을 수 있다. 따라서 중요한 제약은 저장소의 짧은 문서로 남기고 현재 작업에서 다시 확인할 수 있게 한다.

맥락의 양과 맥락의 유효성은 다르다

Liu 등의 장문 문맥 연구는 관련 정보의 위치에 따라 문서 질의응답과 키 검색 성능이 달라질 수 있음을 보여 주었다. 연구에 포함된 모델과 과제에서 관찰된 현상이며 모든 최신 모델에 동일한 곡선이 나타난다는 뜻은 아니다. 그래도 자료를 많이 넣는 것만으로 정보 사용이 보장되지 않는다는 문제를 생각하게 한다. [S05]

자료 탐색기의 검색 오류를 수정할 때 프로젝트 전체를 무작정 붙여 넣는 대신 검색 함수와 데이터 예시 및 실패 입력을 제공한다. 주변 의존성이 필요한 경우에는 해당 함수를 호출하는 위치를 추가한다. 이 방식은 길이를 줄이는 것 자체보다 문제와 관련된 정보를 식별하는 데 목적이 있다. 누락한 의존성 때문에 잘못된 수정을 할 수 있으므로 모델이 추가로 읽어야 할 파일을 먼저 제안하도록 할 수도 있다.

검색 증강 생성은 외부 자료를 검색해 생성 과정에 연결하는 접근이다. Lewis 등의 RAG 연구는 매개변수에 저장된 지식과 검색된 지식을 결합하는 방식을 다뤘다. 코딩 도구가 저장소 파일을 검색해 문맥으로 넣는 과정과 개념적 관련이 있지만 모든 파일 검색 기능이 논문의 학습 구조를 그대로 사용하는 것은 아니다. [S06]

실무에서는 검색 결과의 최신성과 권위 및 범위를 확인해야 한다. 오래된 README와 현재 코드가 서로 다르면 둘 다 제공하는 것만으로 모순이 해결되지는 않는다. 어느 파일을 기준으로 판단해야 하는지 명시하고 실제 실행으로 확인한다. 검색은 증거를 가져오는 과정이며 증거를 평가하는 과정까지 대신하지 않는다.

좋은 프롬프트는 검토 가능한 작업 계약이다

작업 요청에는 목적과 현재 상태 및 변경 범위와 완료 조건이 필요하다. 자료 탐색기에 과목 필터를 추가할 때는 제목 검색과 즐겨찾기의 기존 동작을 유지한다고 적는다. 변경해야 할 파일과 바꾸지 않을 공용 인터페이스를 명시한다. 결과 보고에는 수정 요약과 실행한 검사 및 남은 불확실성을 요구한다. 이렇게 하면 답변의 설득력보다 확인 가능한 증거를 중심으로 검토할 수 있다.

한 번에 많은 기능을 요청하면 기능 사이의 상호작용을 검토하기 어렵다. 작게 나눈 작업은 변경의 원인을 추적하기 쉽게 한다. 그러나 모든 일을 한 줄 단위로 쪼개면 맥락 전달 비용이 늘고 전체 설계가 흐려질 수 있다. 적절한 작업 단위는 독립적으로 설명하고 검사하고 되돌릴 수 있는 의미 있는 변화다.

가속과 탐색에서는 필요한 답변이 다르다

Barke와 동료들은 20명의 참여자가 네 가지 언어의 과제를 수행하는 모습을 분석했다. 연구는 다음 행동을 아는 상태에서 구현을 빠르게 진행하는 가속 모드와 다음 행동을 정하지 못한 상태에서 선택지를 찾는 탐색 모드를 구분했다. 이는 특정 제품이 언제나 두 상태로만 사용된다는 법칙이 아니라 관찰된 상호작용을 설명하는 질적 연구의 개념이다. [S17]

자료 탐색기의 공백 처리 규칙을 이미 정했다면 가속에 가까운 요청을 할 수 있다. 수정할 함수와 유지할 동작 및 확인할 입력을 전달하고 작은 변경을 받는다. 반면 제목 검색만으로 충분한지 판단하지 못했다면 탐색이 먼저다. 이때는 곧바로 파일을 수정하기보다 대안의 전제와 실패 조건을 비교하는 설명이 필요하다. 두 상태를 혼동하면 결정되지 않은 설계가 코드로 먼저 굳어진다.

탐색 단계의 좋은 산출물은 많은 코드가 아니라 좁혀진 불확실성이다. 자료량이 어느 정도인지와 오프라인 사용이 필요한지 및 누가 자료를 수정하는지 확인하면 후보 구조를 줄일 수 있다. 가속 단계에서는 이미 결정한 경계를 지키는 것이 중요하다. 함수 하나의 변경을 요청했는데 저장 방식까지 바꾸었다면 작업량이 늘어난 것보다 판단 범위를 넘어선 것이 문제다.

이 구분은 사람의 능력을 고정적으로 평가하는 기준도 아니다. 익숙한 언어에서는 가속 모드에 있던 개발자가 새로운 인증 체계를 다룰 때는 탐색 모드로 이동할 수 있다. 같은

사람도 과제에 따라 필요한 지원이 달라진다. 스스로 현재 무엇을 알고 무엇을 결정하지 못했는지 밝히면 AI에게 맡길 일과 검토할 일을 더 정확하게 나눌 수 있다.

유용한 설명은 사용자의 수정 행동으로 이어진다

Amershi와 동료들의 인간과 AI 상호작용 지침은 처음 사용할 때와 평상시 사용 및 오류가 발생했을 때와 시간이 흐른 뒤의 행동을 함께 다룬다. 연구진은 18개 지침을 정리하고 다단계 검토를 통해 평가했다. 이 연구는 코드 생성만을 다룬 실험은 아니지만 능력의 범위를 알리고 오류의 수정 경로를 제공해야 한다는 설계 관점을 제시한다. [S18]

이 관점을 개발 도구에 적용하면 작업 시작 전에 접근 가능한 파일과 실행 가능한 검사의 범위를 드러내는 것이 유용하다. 파일 내용을 읽지 못한 상태에서 구조를 추정한 답변과 실제 저장소를 읽고 작성한 답변은 같은 증거를 갖지 않는다. 사용자는 결과 문장의 자신감이 아니라 도구가 관찰한 자료와 수행한 행동을 알아야 한다.

오류가 발생했을 때는 재시도 버튼만 제공하는 것으로 충분하지 않을 수 있다. 무엇이 실패했으며 입력을 수정해야 하는지와 환경을 바꾸어야 하는지 및 이전 상태로 돌아갈 수 있는지를 보여 주어야 한다. 자료 탐색기의 저장에 실패했다면 검색과 표시 기능은 계속 사용할 수 있는지 설명할 수 있다. 실패를 시스템 전체의 붕괴로 표현하지 않고 실제 영향 범위에 맞게 안내한다.

설명 가능성을 모든 내부 계산을 공개하는 것으로 이해할 필요는 없다. 사용자의 다음 결정을 돕는 이유와 근거가 중요하다. 이 변경은 제목이 없는 항목을 제외하며 원본 자료는 수정하지 않는다는 설명은 실제 결과를 예상하게 한다. 매우 복잡한 내부 사고를 장황하게 서술해도 어느 파일이 바뀌었는지 알 수 없다면 검토에는 도움이 적다.

대화의 기억과 프로젝트의 사실을 구분한다

긴 대화에는 확정된 규칙과 잠정 제안 및 실패한 시도가 섞인다. 이를 모두 같은 수준의 사실로 취급하면 폐기한 구조가 다시 등장할 수 있다. 프로젝트의 현재 규칙은 명세와 소스 및 검사 결과에 남기고 대화는 작업의 맥락으로 사용한다. 새 대화를 시작할 때도 전체 기록을 복사하기보다 현재 목표와 제약 및 관련 파일과 실패 증거를 전달하는 편이 검토하기 쉽다.

AI에게 먼저 현재 상황을 요약하게 할 수는 있다. 다만 그 요약이 사실과 맞는지 확인해야 한다. 사용하지 않는 패키지를 사용 중이라고 하거나 아직 실행하지 않은 검사를 통과했다고 서술했다면 다음 작업의 전제가 잘못된다. 요약의 품질은 자연스러운 문장이 아니라 실제 저장소와의 일치로 판정한다.

반복적으로 같은 오해가 생긴다면 프롬프트를 더 강하게 쓰는 것만으로 해결하려 하지 않는다. 규칙이 코드에서 드러나지 않거나 문서가 오래되었거나 검사 명령이 불분명할

수 있다. 이름을 바꾸고 계약을 정리하며 재현 가능한 검사를 만드는 일이 더 안정적인 개선이 될 수 있다. AI 협업의 품질은 대화 기술과 프로젝트 구조가 함께 결정한다.

다음 토큰의 가능성과 프로그램의 정확성은 다른 값이다

자기회귀 언어 모델은 앞선 문맥에 조건부로 다음 토큰을 생성하는 방식으로 이해할 수 있다. 긴 출력의 확률은 각 단계의 조건부 확률을 결합한 값이다. 그러나 자연스러운 토큰의 연결이 업무 규칙을 만족하는 프로그램이라는 보장은 없다. 모델이 보기에 흔한 코드 패턴과 현재 요구에 맞는 코드가 다를 수 있다.

이를 자료 탐색기의 필터로 생각해 보자. 모델은 흔히 쓰는 filter 구문을 매끄럽게 생성할 수 있다. 하지만 과목 조건과 검색어 조건을 AND로 결합해야 하는지 OR로 결합해야 하는지는 요구의 의미에 달려 있다. 문법과 스타일이 자연스러워도 결합 규칙이 틀리면 결과는 잘못된다. 생성의 확률과 요구 충족의 판정을 분리하는 이유다.

샘플링 설정을 바꾸면 출력의 다양성과 반복성이 달라질 수 있다. 같은 요청에서 다른 후보가 나오는 것은 탐색에 도움이 될 수 있지만 검증 비용을 늘린다. 낮은 무작위성을 사용했다고 논리적 정확성이 보장되는 것도 아니다. 현재 과제에서 후보 다양성이 필요한지와 안정적인 재현이 중요한지를 구분한다.

컨텍스트는 저장소 전체의 자동 이해를 뜻하지 않는다

도구가 저장소에 접근할 수 있다는 사실과 모델이 현재 응답에서 모든 파일을 읽고 이해했다는 사실은 다르다. 실제로 제공된 파일과 검색 결과 및 도구 호출의 범위가 문맥을 구성한다. 관련 파일이 빠졌다면 모델은 이름과 관례로 연결을 추정할 수 있다. 추정이 필요한 부분을 확인 질문이나 추가 관찰로 바꾸는 과정이 중요하다.

검색으로 관련 코드를 찾을 때는 호출 관계와 데이터 경로를 함께 확인한다. 같은 함수 이름이 여러 파일에 있거나 생성 파일과 원본 파일이 함께 존재할 수 있다. 검색 결과의 첫 항목을 수정했다고 실제 실행 경로가 바뀌는 것은 아니다. 생성된 파일을 고쳤다면 다음 빌드에서 변경이 사라질 수도 있다.

컨텍스트를 많이 넣는 것보다 결정에 필요한 증거를 정리하는 편이 효과적일 수 있다. 현재 목표와 변경하지 않을 계약 및 관련 코드와 실패 입력을 연결한다. 특히 오래된 문서와 현재 구현이 다르면 어느 쪽을 기준으로 삼아야 하는지 명확히 한다. 모델은 모순된 입력의 우선순위를 항상 올바르게 추론하지 않는다.

모델의 출력이 도구의 명령이 되는 순간 권한이 개입한다

설명을 생성하는 모델과 파일을 수정하거나 명령을 실행하는 시스템은 영향 범위가 다르다. 출력이 실제 행동으로 이어지려면 도구 이름과 인자 및 실행 권한을 해석하는 계층이 필요하다. 이 계층에서 입력을 검증하고 허용된 대상과 행동을 제한한다. 모델의 언어적 판단에 모든 권한 결정을 맡기지 않는다.

이 분리는 도구 사용을 느리게 만들기 위한 것이 아니다. 관찰과 수정 및 공개의 경계를 명확히 해 실패를 줄이는 방법이다. 읽기 전용 조사는 자동으로 수행하고 외부에 영향을 주는 행동에는 필요한 조건을 적용할 수 있다. 어떤 행동을 실제로 수행했는지 기록하면 설명과 실행의 차이를 검토할 수 있다.

사례 | 같은 요청을 세 번 반복하던 이유

업무 요청판에서 완료 버튼을 눌렀는데 목록의 수가 바뀌지 않았다. 사용자는 잘 고쳐달라는 말만 반복했다. 이후 완료 전후의 자료와 화면 수치를 함께 제시하자 목록과 집계가 다른 상태를 읽는 문제가 드러났다. 반복되는 추측 대신 관찰 근거가 수정 범위를 좁혔다.

현재: 대기 두 건과 완료 한 건
동작: 첫 대기 항목의 완료 버튼을 한 번 누름
기대: 대기 한 건과 완료 두 건
실제: 항목 색만 바뀌고 집계는 그대로임
요청: 집계가 읽는 상태와 목록의 상태를 비교해 줘.

다른 분야에 적용하면

예약 도구에서는 선택한 날짜와 예약 요청의 날짜가 같은지 본다. 독서 기록에서는 편집 중인 글과 저장된 글을 구분한다. 설문에서는 화면의 응답 수와 실제 저장 건수를 비교한다. 사례가 달라도 입력과 상태와 결과의 연결을 찾는 질문은 같다.

새 대화를 시작할 때 남길 인계문

프로젝트 목적과 실행 명령과 현재 작동하는 기능을 적는다. 마지막 변경과 실패한 검사 및 다음 작업도 남긴다. 실제 비밀이나 개인정보는 요약문에 옮기지 않는다. 새로운 AI가 무엇을 먼저 읽어야 하는지 파일 위치를 안내한다.

적용과 확인 | 모호한 요청을 바꾸기

1. 기존 요청 하나를 고른다. 예시는 모바일에서 예쁘게 해 줘다.
2. 화면 너비와 문제가 생기는 요소를 적는다.

3. 유지할 행동과 바꿀 표현을 구분한다.
4. 변경 후 확인할 동작을 적는다.
5. AI의 수정 파일 목록을 보고 요청 밖의 변경이 있는지 읽는다.

요청 전	검토 가능한 요청
모바일에서 예쁘게	375px에서 과목 선택과 검색창이 가로로 넘치지 않게
오류를 모두 고쳐	이 재현 순서의 첫 오류를 원인과 함께 수정
전문가처럼 만들어	변경 근거와 검사 결과 및 남은 한계를 정리

결과 비교

문장만 더 길어졌는지 살피지 않는다. 서로 다른 사람이 같은 결과를 확인할 수 있게 됐는지 본다. AI가 실행하지 못한 검사가 있다면 그 사실을 다음 작업에 남긴다.

검토 질문 AI가 원래 있던 기능을 지웠다면 요청을 더 길게 쓰면 해결될까? **해설** 유지할 조건을 명시하는 일과 실제 변경을 검토하는 일이 함께 필요하다. 길이만으로 기존 기능의 보존을 보장할 수는 없다.

관련 근거 [\[R02\]](#)

7장 AI 도구와 작업 방식을 평가하는 실험

평가하려는 능력을 먼저 정의한다

모델이 코드를 잘 만든다는 문장은 여러 능력을 묶고 있다. 짧은 함수를 작성하는 능력과 기존 저장소에서 관련 파일을 찾는 능력 및 오류 로그를 해석하는 능력은 다르다. 요구가 모호할 때 질문을 하는 능력과 잘못된 전제를 지적하는 능력도 실제 개발에 중요하다. 하나의 점수로 모든 능력을 대표하기 어렵다.

자료 탐색기를 평가 과제로 사용할 때는 첫 화면 생성과 검색 규칙 수정 및 저장 오류 복구를 별도 과제로 정할 수 있다. 각각의 시작 파일과 요구 및 허용된 도구와 완료 조건을 기록한다. 어떤 모델은 초기 화면을 빠르게 만들지만 기존 구조의 작은 수정에서는 불필요한 변경을 많이 만들 수 있다. 평가 과제를 나누면 이런 차이를 관찰할 수 있다.

공개 벤치마크는 같은 조건에서 여러 시스템을 비교하는 데 도움이 된다. 그러나 평가 환경과 실제 업무가 다르면 순위가 그대로 유지된다고 보장할 수 없다. 저장소 언어와 프로젝트 규모 및 도구 접근과 시간 제한이 모두 영향을 줄 수 있다. 벤치마크의 이름뿐 아니라 과제 정의와 판정 방법을 읽는다.

테스트 집합을 학습과 평가에 나누어 사용한다

프롬프트를 고칠 때 계속 보는 사례는 개발 과정의 일부가 된다. 그 사례에서 잘 작동하도록 반복 조정한 뒤 같은 사례만으로 일반적인 성능을 주장하면 과대평가할 수 있다. 처음에는 보지 않았던 입력과 다른 자료를 남겨 두고 최종 확인에 사용한다. 작은 프로젝트에서도 이 구분은 유용하다.

예를 들어 제목 검색의 개발 사례에는 분수와 실험이라는 단어를 사용하고 최종 확인에는 앞뒤 공백과 대소문자 및 결과가 없는 검색어를 포함할 수 있다. 다만 이 정도의 분리도 모든 일반화가 보장되는 것은 아니다. 요구의 중요한 경계를 대표하는지 검토해야 한다. 보지 않은 사례라는 사실만으로 좋은 평가 자료가 되지는 않는다.

평가 자료에 실제 비밀과 개인정보를 넣을 필요는 없다. 구조와 경계 조건을 보존한 가짜 자료를 만들 수 있다. 다만 실제 데이터의 길이와 분포 및 오류 형태가 가짜 자료와 다르면 운영에서 다른 문제가 나타날 수 있다. 개인정보 보호와 대표성 사이의 선택을 기록한다.

정확도와 재현율을 검색 사례에서 이해한다

검색 결과를 평가할 때 반환된 자료 중 관련 자료의 비율을 정밀도라고 부른다. 전체 관련 자료 중 반환된 자료의 비율은 재현율이다. 두 값은 서로 다른 실패를 보여 준다. 관련 없는 자료를 많이 보여 주면 정밀도가 낮아지고 필요한 자료를 놓치면 재현율이 낮아진다.

가상의 자료 20개 중 현재 질문에 관련된 자료가 5개라고 하자. 검색이 4개를 반환했고 그중 3개가 관련 자료라면 정밀도는 $3/4$ 이고 재현율은 $3/5$ 다. 이 숫자는 측정 방법을 설명하는 예시이며 특정 검색 서비스의 실험 결과가 아니다. 무엇을 관련 있다고 판정했는지도 평가의 일부다.

정확한 제목 포함 검색에서는 문자열 규칙으로 관련성을 정할 수 있다. 의미 검색에서는 사람의 판정이 필요할 수 있다. 판정자 간 의견이 다르면 기준을 더 명확히 하거나 의견을 기록한다. 평가의 어려움이 모델의 출력에만 있는 것은 아니다. 정답을 정의하는 과정 자체가 중요한 작업이다.

구분	실제로 관련 있음	실제로 관련 없음
검색에 포함	필요한 자료를 찾음	불필요한 자료를 포함
검색에서 제외	필요한 자료를 놓침	불필요한 자료를 제외

사용 맥락에 따라 오류의 비용은 다르다. 간단한 자료 추천에서는 몇 개의 불필요한 항목을 보는 비용이 작을 수 있다. 중요한 자료를 반드시 찾아야 하는 작업에서는 누락의 비용이 더 클 수 있다. 지표를 선택할 때는 수학적으로 계산하기 쉬운 값보다 사용자의 실패 비용을 고려한다.

반복 실행의 변동을 기록한다

생성형 모델의 출력은 같은 요청에서도 달라질 수 있다. 제품의 내부 설정과 모델 버전 및 실행 환경이 변할 수도 있다. 한 번 성공한 출력을 골라 도구의 일반적인 능력이라고 판단하지 않는다. 같은 과제를 여러 번 수행할 때 성공률과 수정량 및 비용이 어떻게 달라지는지 관찰할 수 있다.

반복 횟수가 적으면 추정의 불확실성이 크다. 다섯 번 모두 성공했다고 언제나 성공한다고 말할 수는 없다. 반대로 한 번의 실패만으로 모든 작업에서 쓸모없다고 결론 내리기도 어렵다. 표본 수와 실패 유형을 함께 기록하고 사용하려는 위험 수준에 맞게 평가 범위를 넓힌다.

모델 비교에서는 동시에 바뀐 조건을 줄인다. 한 도구에는 정확한 요구와 테스트를 주고 다른 도구에는 모호한 요청만 주었다면 결과 차이를 모델 자체의 효과로 해석하기 어렵다. 실제로 사용할 작업 흐름 전체를 비교하려는 목적이라면 그 차이를 포함해도 되지만 평가 대상이 모델인지 작업 방식인지 명시한다.

프롬프트 개선은 가설을 시험하는 과정이다

프롬프트에 설명을 더 넣으면 좋아질 것이라는 생각은 가설이다. 개선 전후에 같은 과제를 실행하고 무엇이 달라졌는지 확인한다. 오류 메시지와 재현 입력을 추가했다니 수정 범위가 줄었다면 관련 근거가 생긴다. 다만 한 사례에서의 개선을 모든 프로젝트에 일반화하지 않는다.

성공 기준을 응답 문장의 길이나 자신감으로 두지 않는다. 바뀐 파일 수와 불필요한 변경 및 실제 테스트 결과와 검토 시간을 함께 본다. AI가 긴 설명을 했지만 실행하지 않았다면 그 설명은 가설의 수준이다. 실행했다고 보고했다면 어떤 명령과 환경에서 어떤 결과가 나왔는지 확인한다.

프롬프트 버전도 간단히 기록할 수 있다. P1은 목적만 전달했고 P2는 데이터 계약을 추가했으며 P3는 실패 입력을 추가했다고 정리한다. 대화 전체를 복제하지 않아도 어떤 정보가 결과에 영향을 주었는지 되짚을 수 있다. 이 기록은 이후 다른 도구로 옮길 때도 유용하다.

비용과 품질의 교환 관계를 의사결정에 반영한다

더 강한 모델이 모든 단계에서 필요한 것은 아니다. 작은 문서 수정과 복잡한 권한 설계는 검토 난이도가 다르다. 저렴한 초안 생성과 엄격한 검증을 결합할 수도 있고 높은 위험의 작업에는 더 많은 분석과 사람의 검토를 배정할 수도 있다. 선택은 단가만이 아니라 실패 비용과 재작업 시간을 포함해야 한다.

평가표에는 정확성 및 요구 충족과 변경 범위 및 검증 증거와 유지보수 가능성을 나누어 적는다. 점수를 단순히 합칠 때는 가중치가 어떤 가치를 우선하는지 설명한다. 보안 경계를 위반한 결과를 예쁜 화면 점수로 상쇄하는 식의 평가를 피하려면 반드시 통과해야 하는 조건을 별도로 둔다.

이 장의 평가 방법은 특정 제품의 우열을 선언하기 위한 것이 아니다. 자신의 프로젝트에서 도구가 맡을 수 있는 범위를 판단하기 위한 것이다. 평가를 마친 뒤에는 잘한 일과 실패한 일 및 사람의 검토가 필요했던 지점을 남긴다. 도구의 능력을 정확히 아는 것이 과신하거나 전면적으로 불신하는 태도보다 실무에 도움이 된다.

pass@k는 후보 집합에 대한 성공 가능성을 묻는다

Chen과 동료들의 코드 생성 연구는 생성된 프로그램을 단위 테스트로 평가하는 HumanEval과 pass@k를 설명했다. pass@k는 한 과제에서 k개의 후보 중 적어도 하나가 검사를 통과하는 상황을 다룬다. 이는 첫 번째 답변을 그대로 사용할 때의 성공과 다르다. 여러 후보에서 올바른 것을 식별하는 판정 수단이 있어야 실제 사용에 연결된다.

[S21]

한 과제에서 n개의 후보를 생성했고 그중 c개가 통과했다면 논문은 조합을 이용한 추정량을 제시한다. 식은 $1 - C(n-c, k) / C(n, k)$ 다. 여기서 C는 조합의 수이며 n은 k 이상이어야 한다. 실패 후보만 k개 고르는 비율을 전체에서 빼는 해석이다. 여러 과제에서는 과제별 값을 계산한 뒤 정한 집계 방식으로 요약한다. [S21]

가상의 예에서 n이 10이고 c가 3이며 k가 2라면 실패 후보만 두 개 고르는 경우는 C(7,2)다. 전체 경우 C(10,2)와 비교하면 추정값은 약 0.533이다. 이 값은 특정 제품의 성능이 아니라 식을 설명하는 계산 예다. 후보 세 개가 성공했으니 두 번 시도하면 반드시 성공한다는 뜻이 아니다.

pass@k가 높아져도 사람이 모든 후보를 검토하는 시간이 지나치게 크면 실제 작업의 이익은 작을 수 있다. 후보 생성 비용과 검사 비용 및 선택의 오류를 함께 평가한다. 테스트가 불완전하면 잘못된 후보를 성공으로 셀 수도 있다. 지표의 정의와 판정기의 품질이 함께 중요하다.

평가 자료를 강화하면 순위가 바뀔 수 있다

Liu와 동료들의 EvalPlus 연구는 기존 코드 생성 벤치마크의 테스트를 확장해 이전에 통과했던 코드의 오류를 더 발견했다. 연구는 불충분한 테스트가 점수뿐 아니라 모델의 순위에도 영향을 줄 수 있음을 보여 주었다. 이는 당시의 평가 대상과 확장 방식에 대한 결과이며 현재 모든 모델의 오류 비율로 옮길 수 없다. [S22]

이 연구를 자료 탐색기의 평가에 적용하면 정상 검색어 몇 개만으로 도구를 비교하지 말아야 한다는 해석을 얻을 수 있다. 공백과 중복 ID 및 누락 필드와 복합 필터를 포함하면 다른 실패가 드러날 수 있다. 어느 모델이 더 낫다는 결론은 사용한 과제와 판정 기준에 조건부로 제시한다.

평가용 자료를 개선한 뒤에는 이전 결과와 직접 비교할 수 있는지도 확인한다. 테스트 집합이 바뀌었는데 점수만 이어 붙이면 성능의 변화와 측정 도구의 변화를 혼동한다. 평가 판본과 모델 판본 및 실행 설정을 함께 기록한다. 수치의 재현성을 위해서는 질문의 재현성도 필요하다.

평균 효과와 개인의 효과를 구분한다

같은 도구가 익숙한 과제에서는 시간을 줄이고 낯선 과제에서는 검토 시간을 늘릴 수 있다. 여러 작업의 평균은 이런 차이를 가릴 수 있다. 모델 비교에서는 언어와 저장소의 규모 및 과제 유형과 사용자의 경험을 함께 기록한다. 특정 집단의 평균을 모든 개발자의 기대 효과로 표현하지 않는다.

작은 팀에서 도구를 시험한다면 작업을 가능한 한 비슷한 난이도로 나누고 순서 효과를 고려한다. 먼저 수행한 작업을 두 번째에 다시 하면 도구의 효과와 학습의 효과가 섞인다. 서로 다른 과제를 쓰면 난이도 차이가 섞일 수 있다. 완벽한 실험을 약속하기보다 남은 혼란 변수를 명시하는 것이 정확하다.

성공과 실패를 보고할 때는 표본 수와 제외한 과제를 적는다. 중간에 포기한 작업을 결과에서 빼면 도구가 실제보다 좋아 보일 수 있다. 성공 사례만 선택한 시연은 가능성을 보여 주지만 평균적인 업무 성과를 추정하는 자료는 아니다. 연구와 제품 시연의 목적을 구분해 읽는다.

효과의 크기와 실무적 중요성을 따로 판단한다

차이가 통계적으로 관찰되었다는 사실과 실무적으로 충분히 큰 차이라는 판단은 다르다. 짧은 작업에서 몇 초를 줄이는 효과가 중요한지와 큰 배포 실패 하나를 줄이는 효과가 중요한지는 사용 맥락에 달려 있다. 절대 시간과 비율 및 실패의 비용을 함께 읽는다.

가상의 작업 A가 10분에서 8분으로 줄고 작업 B가 100분에서 90분으로 줄었다고 하자. 비율로는 A의 개선이 크지만 절약한 시간은 B가 크다. 과제별 비율을 단순 평균하는 방식과 총시간의 차이를 비교하는 방식은 다른 답을 준다. 집계 방식이 어떤 질문에 답하는지 먼저 정한다.

도입 결정에는 검토자의 피로와 유지보수 부담도 포함된다. 생성 시간이 줄어도 코드가 복잡해져 이후 변경이 느려진다면 장기 효과가 달라질 수 있다. 단기 완료 시간과 일정 기간 이후의 재작업을 함께 관찰하면 더 균형 잡힌 판단이 가능하다.

8장 화면을 만드는 일에서 수직 기능을 완성하는 일로

입력에서 결과까지 하나의 흐름을 연결한다

수직 기능은 사용자의 행동이 입력 처리와 상태 변경을 거쳐 관찰 가능한 결과로 이어지는 작은 완성 단위다. 자료 탐색기에서는 검색창에 글자를 입력하면 조건이 갱신되고 자료가 걸러지며 결과 수가 바뀌는 흐름이 이에 해당한다. 검색창의 외형만 만들거나 필터 함수만 작성한 상태는 각각 일부 계층이 준비된 것이다.

수직 기능을 먼저 완성하면 계층 사이의 가정을 일찍 확인할 수 있다. 입력 이벤트가 예상대로 발생는지와 함수가 반환한 값이 화면에 표시되는지를 한 번에 볼 수 있다. 이후 과목 필터를 추가할 때는 이미 작동하는 흐름에 조건 하나를 더 연결한다. 이 방식은 완벽한 설계를 먼저 끝내야 한다는 뜻이 아니라 실제 동작으로 설계를 검증하는 반복을 짧게 만드는 것이다.

다만 첫 화면을 빠르게 만드는 편의와 장기 구조를 구분해야 한다. 모든 동작을 하나의 큰 함수에 넣으면 처음에는 연결하기 쉽지만 다음 변경에서 영향을 예측하기 어렵다. 상태를 읽는 부분과 순수 계산 부분 및 DOM을 바꾸는 부분을 구분하면 테스트와 수정의 범위가 분명해진다.

순수 함수와 부작용을 구분하는 이유

순수 함수는 같은 입력에 대해 같은 출력을 내며 외부 상태를 바꾸지 않는 함수다. 제목 검색은 자료 배열과 검색어를 받아 결과 배열을 반환하도록 만들 수 있다. 반면 DOM을 수정하거나 파일을 저장하거나 네트워크를 호출하는 일은 외부 상태와 상호작용한다. 이러한 부작용은 프로그램에 반드시 필요하지만 좁은 경계에 모아 두면 추론이 쉬워진다.

자료 탐색기의 검색 결과가 잘못되었을 때 순수 검색 함수를 독립적으로 실행하면 데이터 계산 문제인지 화면 갱신 문제인지 분리할 수 있다. 테스트에서는 브라우저 없이도 검색 함수의 입력과 출력을 비교할 수 있다. 화면 동작은 별도로 브라우저에서 확인한다. 테스트 도구 하나로 모든 문제를 해결하려는 접근보다 각 계층의 관찰 방법을 맞추는 편이 효과적이다.

컴포넌트 분리는 파일 수를 늘리는 기술이 아니다. 함께 바뀌는 책임을 모으고 독립적으로 바뀌는 책임을 분리하는 설계다. 검색 조건과 저장 방식이 서로 다른 이유로

바뀐다면 두 기능의 경계를 나눌 근거가 있다. 반대로 두 줄짜리 함수를 의미 없이 여러 파일로 나누면 읽는 비용만 늘어날 수 있다.

완료는 화면의 인상보다 행동의 증거로 판단한다

AI가 만든 화면은 시각적으로 완성되어 보일 수 있다. 그러나 검색 조건을 빠르게 바꾸거나 결과가 없는 상태로 들어가면 문제가 드러날 수 있다. 빈 결과와 오류 및 로딩 상태도 정상적인 제품 경험의 일부다. 사용자가 무엇을 해야 하는지 알 수 있는 문구와 다시 시도할 방법을 제공한다.

첫 기능을 완료할 때는 성공 입력 하나만 기록하지 않는다. 빈 입력과 잘못된 입력 및 경계 입력을 함께 남긴다. 검색어를 지웠을 때 원래 목록으로 돌아오는지와 키보드만으로 검색창에 접근할 수 있는지도 확인한다. 이러한 기록은 다음 기능을 추가한 뒤 기존 동작이 유지되는지를 판단하는 회귀 기준이 된다.

가장 작은 기능에도 입력에서 관찰까지의 연결이 있다

수직 기능은 화면 한 조각이나 함수 하나만을 뜻하지 않는다. 사용자의 입력이 실제 계산과 필요한 저장을 거쳐 관찰 가능한 결과로 이어지는 좁은 경로다. 자료 탐색기의 첫 기능은 제목 검색 하나일 수 있다. 이 경로가 완성되면 입력 정규화와 계산 및 화면 표시가 실제로 연결되었는지 확인할 수 있다.

수평으로 화면 전체를 만든 뒤 계산 전체를 만들고 마지막에 연결하면 가정의 불일치가 늦게 드러날 수 있다. 작은 수직 경로에서는 입력 이름과 반환 형식 및 오류 처리를 일찍 대조한다. 이후 기능은 이미 작동하는 경로에 추가되므로 실패의 원인을 비교적 좁게 찾을 수 있다.

단순한 성공 경로만 수직으로 연결하면 실패가 뒤로 밀린다. 검색 결과 없음과 잘못된 자료 형식 및 저장 실패 중 현재 기능에 해당하는 경계를 함께 정한다. 모든 실패를 첫 버전에 구현할 필요는 없지만 무엇을 처리하고 무엇을 미구현으로 남겼는지 알아야 한다.

작업을 나누는 기준은 독립적으로 확인할 수 있는가다

파일 수로 작업을 나누면 서로 강하게 연결된 계약이 여러 과제로 흩어질 수 있다. 한 사람이 API 필드 이름을 바꾸고 다른 사람이 화면을 만들면 통합 전까지 불일치를 발견하지 못할 수 있다. 작은 요구와 검증 가능한 결과를 기준으로 나누는 편이 진행 상태를 읽기 쉽다.

AI에게 한 번에 큰 변경을 요청하면 실패했을 때 원인을 찾기 위해 넓은 diff를 읽어야 한다. 목표 하나와 관련된 변경을 수행하고 증거를 확인한 뒤 다음 단계로 가면 판단의

부담을 줄일 수 있다. 작업 분할은 프롬프트의 길이를 줄이는 기술보다 검토의 단위를 만드는 설계다.

사례 | 공지판의 더 보기 버튼

공지 카드 세 개를 만든 뒤 더 보기 버튼을 추가했다. 첫 구현에서는 버튼이 있지만 아무 반응이 없었다. 이후 누르면 본문이 열리고 다시 누르면 닫히게 했다. 마지막으로 키보드로도 같은 행동이 되는지 확인했다. 단계마다 서로 다른 완료 기준이 있었다.

구현 단계	실제로 확인할 것
버튼 표시	이름이 목적을 설명하는가
동작 연결	본문이 열리고 닫히는가
상태 표시	현재 열림 여부가 전달되는가
키보드	Tab으로 이동하고 Enter로 동작하는가
작은 화면	본문이 카드 밖으로 잘리지 않는가

간단한 예제 읽기

```
const result = document.querySelector('#result');
const query = document.querySelector('#query');
query.addEventListener('input', () => {
  result.textContent = query.value.trim();
});
```

이 코드는 입력한 글자를 결과 영역에 표시하는 연결 예제다. 검색 필터나 서버 저장까지 구현한 코드는 아니다. 같은 방식으로 값을 읽고 처리하고 표시하는 위치를 구분해 본다.

적용과 확인 | 더 보기 카드 한 개

1. 제목과 짧은 요약 및 본문이 있는 가짜 공지 하나를 만든다.
2. 버튼을 눌렀을 때 열릴 영역을 지정한다.
3. 같은 버튼으로 다시 닫는 행동을 만든다.
4. 키보드와 작은 화면에서 확인한다.
5. 열기 전과 후 및 다시 닫은 상태를 기록한다.

다른 예시로 바꾸기

영어 표현 카드에서는 예문을 연다. 과학 자료에서는 관찰 안내를 연다. 독서 기록에서는 긴 메모를 연다. 버튼 이름을 상황에 맞게 쓰되 같은 열림 상태를 공유하지 않도록 항목별 동작을 확인한다.

흔한 실패	확인할 위치
모든 카드가 함께 열림	항목별 상태와 대상 선택
새로 고친 화면이 안 보임	저장한 파일과 열린 경로
눌러도 반응 없음	이벤트와 브라우저 오류
버튼이 다른 일을 함	form 안 버튼의 type

검토 질문 화면이 잘 보이면 첫 기능은 완성일까? **해설** 기대 행동과 예외 및 실제 조작까지 확인해야 한다. 화면의 존재와 행동의 연결을 구분한다.

기술 해설 / 배열의 선택과 변환과 집계

filter는 조건에 맞는 항목을 고른다. map은 각 항목을 다른 형태로 만든다. reduce는 여러 항목에서 하나의 집계 결과를 만들 수 있다. 어떤 질문에 답하는지 먼저 정하면 도구의 이름을 외우기 쉬워진다.

과학 자료만 고르는 것과 제목만 추출하는 것 및 과목별 개수를 세는 것을 나누어 본다.

```
const science = items.filter(i => i.subject === '과학');
const titles = science.map(i => i.title);
```

기술 해설 / 이벤트는 행동을 상태에 연결한다

사용자의 입력과 선택은 이벤트로 읽을 수 있다. 이벤트 처리에서 상태를 바꾸고 그 상태로 화면을 갱신한다. 화면 요소를 다시 만들면 초점과 연결된 처리도 어떻게 유지할지 확인한다.

즐거찾기 버튼을 누른 뒤 카드 목록이 다시 만들어진다. 같은 항목의 버튼으로 초점을 돌려 키보드 사용이 이어지게 한다.

```
query.addEventListener('input', render);
subject.addEventListener('change', render);
```

입력에서 표시까지의 계산

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 `x`가 입력을 받고 `return`이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L03 / 한글 제목순으로 정리

원본 순서를 보존하며 제목의 표시 순서를 바꾼다. 이 사례의 입력 계약은 `items`는 제목 배열다.

```
return [...x.items].sort((a, b) => a.localeCompare(b, 'ko'));
```

가방과 나무 및 하늘을 비교해 표시 순서를 바꾼다. 정렬 전에 배열을 복사했으므로 원래 입력 순서를 다른 화면에서 계속 사용할 수 있다. 같은 자료를 여러 방식으로 표시할 때 이 차이가 중요하다.

입력	기대 결과
<code>{"items": ["하늘", "가방", "나무"]}</code>	<code>["가방", "나무", "하늘"]</code>
<code>{"items": []}</code>	<code>[]</code>

해석의 경계. 정렬 함수가 원본 배열을 바꾸지 않는지 확인한다. 날짜순과 제목순을 선택할 수 있도록 기준을 분리한다.

분석 사례 L04 / 목록을 페이지로 나누기

정한 개수만큼 읽되 페이지 번호가 범위를 벗어나지 않게 한다. 이 사례의 입력 계약은 `items`는 목록 / `page`는 1부터 시작 / `size`는 페이지 크기다.

```
if (!Number.isInteger(x.page) || x.page < 1) throw Error('페이지')
;
if (!Number.isInteger(x.size) || x.size < 1) throw Error('크기');
const start = (x.page - 1) * x.size;
return x.items.slice(start, start + x.size);
```

두 번째 페이지이고 한 페이지가 두 개라면 처음 두 개를 건너뛴다. 그다음 두 개가 3과 4다. 마지막 남은 개수가 페이지 크기보다 작아도 정상 결과이며 범위를 벗어나면 빈 배열이 된다.

입력	기대 결과
<code>{"items": [1, 2, 3, 4, 5], "page": 2, "size": 2}</code>	<code>[3, 4]</code>
<code>{"items": [1, 2], "page": 3, "size": 2}</code>	<code>[]</code>

해석의 경계. 마지막 페이지의 항목 수가 작아도 정상으로 본다. 필터를 바꿀 때 페이지를 1로 되돌리는 규칙을 추가한다.

분석 사례 L06 / 질문 카드의 열림 상태

다른 카드를 열어도 선택한 카드의 상태만 바꾼다. 이 사례의 입력 계약은 open은 ID별 상태 / id는 누른 카드다.

```
return {...x.open, [x.id]: !x.open[x.id]};
```

a가 열리고 b가 닫힌 상태에서 b만 반대로 바꾼다. a의 상태는 복사해 유지한다. 화면 전체의 열림 변수 하나로 모든 카드를 연결하면 이 항목별 결과를 만들 수 없다.

입력	기대 결과
<code>{"open": {"a": true, "b": false}, "id": "b"}</code>	<code>{"a": true, "b": true}</code>
<code>{"open": {"a": true}, "id": "a"}</code>	<code>{"a": false}</code>

해석의 경계. 항목 전체가 하나의 Boolean을 공유하지 않게 한다. 하나의 카드만 열리게 할지 여러 개를 허용할지 비교한다.

관련 근거 [\[R06\]](#) [\[R07\]](#)

9장 상태 관리와 데이터 일관성

원본 데이터와 화면 상태와 파생 값을 구분한다

원본 데이터는 자료의 ID와 제목 및 과목 같은 사실이다. 화면 상태는 현재 검색어와 선택한 과목 및 즐겨찾기만 보기 여부다. 검색 결과는 이 두 종류의 입력에서 계산되는 파생 값이다. 파생 값을 별도의 저장 상태로 중복 관리하면 한쪽만 갱신되어 서로 어긋날 수 있다. 필요한 순간에 계산할 수 있다면 원본과 조건을 단일한 기준으로 두는 편이 이해하기 쉽다.

즐거찾기는 자료 자체의 속성으로 보일 수 있지만 사용자마다 다르다. 공용 자료 목록의 객체에 favorite 필드를 직접 넣으면 여러 사용자의 선호가 섞일 수 있다. 개인 화면에서는 즐겨찾기 ID 집합을 따로 보관하고 목록을 그릴 때 결합한다. 서버로 확장하면 사용자와 자료의 관계를 저장하는 별도의 테이블이 자연스러워진다.

ID와 배열 위치도 구분한다. 목록의 두 번째 자료를 즐겨찾기로 기록하면 정렬 순서가 바뀐 뒤 다른 자료를 가리킬 수 있다. 안정적인 ID를 사용하면 정렬과 필터가 바뀌어도 대상의 동일성을 유지할 수 있다. 화면에서 보이는 위치는 표현이며 ID는 참조의 기준이다.

지속성은 저장 성공과 복원 성공을 모두 포함한다

localStorage는 브라우저의 origin별 저장 공간이다. 자료가 문자열로 저장되므로 객체를 JSON으로 직렬화하고 다시 읽을 때 파싱해야 한다. 저장이 차단되거나 용량 문제가 생길 수 있고 손상된 문자열이 남아 있을 수도 있다. file URL에서의 동작은 브라우저마다 다를 수 있으므로 배포 환경과 동일한 origin에서 검증해야 한다. [R08]

저장 실패를 무시하면 사용자는 화면에서 선택한 값이 보존되었다고 오해한다. 저장되지 않았다는 상태를 전달하고 현재 세션에서는 계속 사용할지 결정해야 한다. 반대로 복원 실패 때문에 앱 전체가 시작하지 못하게 만들 필요도 없다. 손상된 저장값을 별도로 보존하거나 초기 상태로 시작하는 정책을 명시한다.

스키마 버전은 저장 구조가 바뀔 때 필요하다. 첫 버전에서는 ID 배열만 저장했지만 다음 버전에서 메모까지 저장한다면 이전 데이터의 해석 방법이 달라진다. 버전 필드를 두고 이전 형식을 새 형식으로 변환하는 마이그레이션을 설계한다. 변환에 실패한 데이터를 조용히 버리는 대신 사용자에게 내보내거나 복구 방법을 제공해야 한다.

동시 수정은 마지막 저장만의 문제가 아니다

공용 요청판에서 두 사용자가 같은 요청의 상태를 읽었다고 하자. 한 사용자는 진행으로 바꾸고 다른 사용자는 제목을 고친다. 두 번째 사용자가 오래된 전체 객체를 저장하면 첫 번째 사용자의 상태 변경이 사라질 수 있다. 이를 분실 갱신이라고 부른다. 화면에서 저장 버튼을 잠시 비활성화하는 것만으로 다른 기기의 동시 요청을 막을 수 없다.

낙관적 동시성 제어는 읽을 때 받은 버전을 수정 요청에 함께 보내고 서버가 현재 버전과 비교하는 방식이다. 버전이 같을 때만 수정하고 증가시키며 다르면 충돌로 응답한다. 핵심은 비교와 수정이 하나의 원자적 작업으로 이루어져야 한다는 점이다. JavaScript에서 두 숫자를 비교하는 순수 함수는 정책을 설명할 수 있지만 실제 데이터베이스의 원자성을 제공하지는 않는다.

충돌을 사용자에게 어떻게 보여 줄지도 설계해야 한다. 무조건 새 값을 덮어쓰는 대신 최신 내용을 다시 읽고 자신의 변경과 비교하도록 할 수 있다. 제목 수정과 상태 변경처럼 서로 독립적인 필드는 부분 업데이트로 충돌을 줄일 수도 있다. 다만 필드 사이에 업무 규칙이 있다면 독립적이라고 가정해서는 안 된다.

중복은 읽기를 쉽게 하면서 갱신을 어렵게 할 수 있다

자료마다 과목 이름과 과목 담당자의 이메일을 반복 저장한다고 하자. 담당자가 바뀌면 여러 자료를 모두 수정해야 한다. 일부만 수정되면 같은 과목의 담당자가 자료마다 달라진다. 이를 갱신 이상으로 생각할 수 있다. 공통 사실을 별도 구조로 두고 식별자로 참조하면 한 번의 수정으로 관계를 유지하기 쉬워진다.

그러나 모든 중복을 제거하는 것이 항상 최선은 아니다. 작은 정적 자료 목록에서는 과목 이름을 문자열로 반복하는 편이 단순하고 충분할 수 있다. 서버에서 대량 조회 성능을 위해 일부 값을 중복 저장하는 선택도 가능하다. 중요한 것은 어떤 값이 권위 있는 원본이며 복제된 값이 언제 갱신되는지 명확히 하는 것이다.

공용 요청판의 사용자 표시 이름과 사용자 ID도 구분한다. 표시 이름은 바뀔 수 있고 중복될 수 있다. 소유 관계는 안정적인 ID로 표현하고 화면에서는 현재 표시 이름을 결합한다. 사람이 읽기 쉬운 값과 시스템이 동일성을 판단하는 값이 같은 역할을 해야 할 이유는 없다.

삭제의 의미를 정하지 않으면 관계가 깨진다

자료가 삭제되었는데 즐겨찾기 ID가 남아 있을 수 있다. 이때 목록을 그리다 오류를 내는 대신 존재하지 않는 ID를 무시하거나 정리하는 정책을 정한다. 서버의 외래 키 제약은 관계의 무결성을 보조할 수 있지만 사용자가 기대하는 삭제 의미까지 자동으로 결정하지는 않는다.

공용 요청판에서는 담당자가 탈퇴해도 과거 요청의 감사 기록을 보존해야 할 수 있다. 사용자 행을 무조건 삭제하면 관계가 끊어질 수 있다. 비활성화와 익명화 및 실제 삭제는 다른 정책이다. 프로젝트의 데이터 필요성과 보존 조건에 맞춰 결정하고 코드가 그 정책을 따르는지 확인한다.

시간대 데이터 모델의 일부다. 날짜만 필요한 업무와 정확한 순간이 필요한 사건 기록을 구분한다. 마감일을 시간대가 있는 순간으로 잘못 바꾸면 사용자의 지역에 따라 하루가 달라질 수 있다. 반대로 사건 로그에서 시간대를 생략하면 여러 지역의 순서를 해석하기 어렵다. 값의 형식보다 값이 뜻하는 사실을 먼저 정한다.

스키마는 현재 코드보다 오래 살아남는다

프로그램 코드는 새 버전으로 교체할 수 있지만 사용자가 저장한 자료는 이전 형식으로 남아 있을 수 있다. 처음에는 즐겨찾기를 문자열 배열로 저장하다가 이후 메모와 저장 시각을 추가하면 두 형식이 동시에 존재할 수 있다. 새 코드는 새 형식만 읽는다고 가정하기 전에 기존 사용자의 값을 어떻게 해석할지 결정해야 한다.

저장 형식에 version 필드를 두면 변환 경로를 명시하기 쉽다. 버전 번호 자체가 호환성을 보장하는 것은 아니다. 번호에 따라 어떤 필드가 존재하고 어떤 기본값을 적용하며 변환에 실패하면 무엇을 보존할지를 정의해야 한다. 읽을 수 없는 자료를 조용히 빈 배열로 바꾸면 사용자는 저장한 자료가 사라졌다고 느낄 수 있다.

```
function migrateFavorites(value) {
  if (Array.isArray(value)) {
    return { version: 2, ids: value.filter(x => typeof x === 'string') };
  }
  if (value && value.version === 2 && Array.isArray(value.ids)) {
    return { version: 2, ids: value.ids.filter(x => typeof x === 'string') };
  }
  throw new Error('지원하지 않는 즐겨찾기 형식');
}
```

이 함수는 형식 변환의 원리를 설명하는 작은 예다. ID 중복 제거와 최대 개수 및 실제 자료에 존재하는 ID인지는 검사하지 않는다. 알 수 없는 형식을 오류로 남기는 선택은 데이터를 임의로 덮어쓰지 않기 위한 것이다. 실제 적용에서는 원본을 별도로 보존하고 사용자에게 복구나 초기화 선택을 안내할 수 있다.

읽기 호환성과 쓰기 호환성은 다르다

새 버전이 이전 자료를 읽을 수 있다는 사실과 이전 버전이 새 자료를 읽을 수 있다는 사실은 다르다. 사용자가 새 탭과 오래 열린 탭을 동시에 사용할 때 두 버전의 코드가 같은 저장소에 접근할 수도 있다. 새 버전이 추가한 필드를 이전 버전이 모른 채 다시 저장하면 정보가 없어질 수 있다. 이런 가능성은 형식 변경의 방향을 결정할 때 고려한다.

작은 정적 앱에서는 주요 저장 형식을 바꿀 때 이전 형식을 복사해 두고 새 키로 옮기는 방법을 사용할 수 있다. 공용 시스템에서는 서버와 클라이언트의 배포 시점이 달라질 수 있으므로 일정 기간 두 형식을 지원하거나 새 필드를 선택적으로 읽는 방식이 필요할 수 있다. 어느 전략이든 지원 기간과 삭제 시점을 정하지 않으면 호환 코드가 계속 쌓인다.

필드를 추가하는 변경도 항상 무해하지 않다. 새 필드가 필수라면 이전 요청은 검증을 통과하지 못할 수 있다. 기존 필드의 단위가 분에서 초로 바뀌었는데 이름은 같다면 더 위험하다. 자료형이 같은 숫자이므로 단순한 형식 검사로는 의미의 변화를 발견하기 어렵다. 스키마에는 구조뿐 아니라 단위와 시간대 및 값의 의미도 포함된다.

마이그레이션의 시험은 전후 자료를 함께 본다

마이그레이션 시험에서는 정상 자료만 준비하지 않는다. 필드가 빠진 자료와 중복 ID 및 알 수 없는 버전과 일부 손상된 자료를 포함한다. 변환을 두 번 수행했을 때 의도한 상태가 유지되는지도 확인한다. 이미 변환된 자료를 다시 읽는 과정은 실제 사용에서 흔하므로 한 번의 성공만으로 충분하지 않다.

검증할 속성을 구체적으로 적으면 시험이 쉬워진다. 유효한 즐겨찾기 ID의 순서가 유지되는지와 지원하지 않는 자료를 덮어쓰지 않는지 및 실패 전후에 원본을 다시 읽을 수 있는지를 확인한다. 데이터 개수가 같다는 사실만으로 내용 보존을 판단하지 않는다. 다른 ID로 바뀌어도 개수는 같을 수 있다.

API가 마이그레이션 코드를 만들었다면 변환 함수뿐 아니라 적용 시점과 실패 처리 및 복구 경로를 함께 검토한다. 함수의 출력이 맞아도 앱 시작 시 무조건 실행하여 정상 자료를 반복 저장한다면 다른 문제가 생길 수 있다. 데이터 변경은 계산의 정확성과 실행 순서 및 지속성의 문제를 함께 가진다.

같은 숫자라도 단위가 다르면 다른 값이다

API가 반환한 60이라는 값만으로는 초인지 분인지 알 수 없다. 필드 이름과 계약에 단위를 명시하고 변환 지점을 제한하면 오류를 줄일 수 있다. 여러 함수에서 각자 변환하면 두 번 변환하거나 한 경로에서 변환을 빼먹는 문제가 생긴다. 내부 기준 단위를 정하고 표시할 때 변환하는 방식을 검토할 수 있다.

금액처럼 소수점 오차가 중요한 자료는 표현 방식도 결정해야 한다. 이진 부동소수점에서는 일부 십진 소수를 정확히 표현하지 못한다. 작은 정수 단위로 관리하거나 적절한 십진 연산 도구를 사용하는 선택이 있다. 어떤 선택이든 반올림 시점과 정책을 명세에 포함한다. 화면에서 소수점을 가리는 것만으로 계산의 정확성이 보장되지는 않는다.

날짜만 필요한 자료와 특정 순간이 필요한 자료도 구분한다. 제출 마감일이 어느 시간대의 어떤 순간인지와 생일처럼 시간대 변환이 필요 없는 날짜인지는 다른 개념이다. UTC로 저장하면 모든 의미가 자동으로 해결된다고 생각하지 않는다. 자료가 표현하는 업무 개념을 먼저 정해야 한다.

파생 상태를 저장하면 동기화 책임이 생긴다

원본 목록과 검색 결과를 모두 저장하면 원본이 바뀔 때 검색 결과도 갱신해야 한다. 즐겨찾기 개수를 별도로 저장하면 선택 목록과 개수가 어긋날 수 있다. 계산 가능한 값을 저장하는 것은 읽기 비용을 줄일 수 있지만 일관성을 유지하는 책임을 추가한다.

자료 탐색기의 규모에서는 원본과 현재 필터에서 결과를 다시 계산하는 구조가 단순할 수 있다. 성능 문제가 실제로 관찰되면 캐시와 색인 같은 파생 자료를 추가할 수 있다. 이때 어떤 변경이 파생 자료를 무효화하는지 정해야 한다. 최적화 전후에 관찰 가능한 결과가 같다는 기준을 유지한다.

중복 저장을 모두 없애는 것이 항상 정답은 아니다. 자주 읽고 드물게 바뀌는 집계는 별도로 유지하는 편이 합리적일 수 있다. 다만 최신성이 얼마나 필요한지와 갱신 실패를 어떻게 복구할지 설명할 수 있어야 한다. 데이터 모델의 전문성은 정규화라는 용어보다 일관성의 책임을 분명히 하는 데 있다.

상태 변화 / 파생 결과는 원본과 조건에서 계산한다

파생 결과는 원본과 조건에서 계산한다

1 / 5 초기 상태에서 전체 자료를 표시한다

원본 자료

6개

검색어

빈 값

과목

전체

보이는 결과

6개

가상 자료의 결과 수를 사용한 상태 모형이다. 원본 자료를 파괴하지 않고 필터 결과를 계산하면 초기화가 명확해진다.

파생 결과는 원본과 조건에서 계산한다

5 / 5 두 조건을 초기화하면 전체로 돌아온다

원본 자료

6개

검색어

빈 값

과목

전체

보이는 결과

6개

가상 자료의 결과 수를 사용한 상태 모형이다. 원본 자료를 파괴하지 않고 필터 결과를 계산하면 초기화가 명확해진다.

가상 자료의 결과 수를 사용한 상태 모형이다. 원본 자료를 파괴하지 않고 필터 결과를 계산하면 초기화가 명확해진다.

- 1. 초기 상태에서 전체 자료를 표시한다
- 2. 제목 조건만 적용한다
- 3. 과목 조건을 논리곱으로 추가한다
- 4. 검색어를 지워도 과목 조건은 유지된다
- 5. 두 조건을 초기화하면 전체로 돌아온다

60

사례 | 즐겨찾기와 회원가입을 혼동하다

개인 자료실에서 즐겨찾기를 만들고 다른 PC에서 접속했더니 목록이 없었다. 첫 구현은 브라우저에만 ID를 보관했다. 로그인과 서버 저장을 만들지 않았으므로 기기 간 동기화를 기대할 근거가 없었다. 사용 안내에서 보관 위치를 설명하고 필요한 다음 기능을 따로 설계했다.

```
const key = 'reading-note-v1';
try {
  localStorage.setItem(key, '가짜 독서 메모');
} catch (error) {
  console.warn('저장을 확인할 수 없음');
}
```

이 예시는 브라우저 저장 호출의 실패를 다룬다. 실제 서비스에서는 사용자가 알아볼 안내도 제공한다. HTML 파일을 직접 여는 file 주소의 저장 동작은 환경마다 다를 수 있다. 같은 로컬 서버 주소에서 다시 열어 확인하면 실습 조건을 맞추기 쉽다.

자료를 가져오기 전에

파일이 JSON으로 읽히는지와 필요한 필드가 있는지를 따로 본다. ID가 중복되면 덮어쓸지 건너뛴지 결정한다. 잘못된 파일을 읽다가 현재 자료를 모두 지우지 않도록 검증을 마친 뒤 반영한다.

적용과 확인 | 보관 위치를 직접 확인

1. 예제의 자료 하나를 즐겨찾기한다.
2. 같은 화면에서 표시가 바뀌는지 본다.
3. 같은 주소를 새로 열어 상태를 확인한다.
4. 다른 브라우저에서 열어 차이를 관찰한다.
5. 데이터 삭제 없이 검색 조건만 초기화해 본다.

확인 상황	기록할 결과
새로고침	같은 즐겨찾기가 남았는가
다른 브라우저	별도 보관 공간인가
저장 차단	실패를 알 수 있는가

확인 상황

기록할 결과

제목 변경

ID 기반 연결이 유지되는가

손상된 가져오기 파일

기존 자료가 보존되는가

상태 설계의 다음 질문

장비 예약에서는 시간이 겹치는 요청을 동시에 받으면 어떻게 될지 생각한다. 업무 요청판에서는 두 사람이 같은 항목을 편집할 때의 규칙을 정한다. 브라우저의 배열을 바꾸는 구현과 공용 데이터의 일관성을 지키는 구현은 필요한 범위가 다르다.

검토 질문 localStorage가 있으면 데이터베이스가 필요 없을까? **해설** 개인 브라우저의 작은 상태에는 활용할 수 있다. 공용 자료와 계정별 권한 및 기기 간 공유가 필요하면 서버 쪽 설계를 별도로 검토한다.

기술 해설 / 원본과 상태와 표시값을 나눈다

원본 자료와 사용자의 조건 및 화면의 결과는 역할이 다르다. 한 위치의 변경을 다른 위치에 복사하는 방식이 늘면 서로 어긋날 수 있다. 가능한 경우 같은 원본과 상태로 결과를 다시 계산한다.

업무 요청판의 카드와 집계가 같은 items를 읽게 한다. 항목 색만 바꾸고 실제 상태를 두지 않는 구현을 피한다.

```
const visible = items.filter(matches);
count.textContent = `${visible.length}개`;
```

기술 해설 / 저장 형식에도 판본이 필요하다

새 필드를 추가하면 예전 저장 자료에 그 필드가 없을 수 있다. 자료 형식의 판본을 기록하고 예전 자료를 어떻게 읽을지 정한다. 가져오기는 형식과 중복 및 적용 단계를 나누어 설계한다.

예전 독서 메모는 제목과 메모만 가졌다. 새 판본의 읽기 상태를 추가할 때 기본값을 정하거나 가져오기 안내를 제공한다.

```
const backup = {schema:1, items:records};
const text = JSON.stringify(backup);
```

기술 해설 / 자료의 관계를 읽는다

요청과 작성자 및 첨부는 서로 연결될 수 있다. 같은 정보를 여러 곳에 복사하면 변경 때 불일치가 생길 수 있다. 무엇을 별도 자료로 관리하고 어떤 ID로 연결할지 정한다.

업무 요청의 작성자 이름을 모든 요청에 복사할지와 사용자 ID로 연결할지를 비교한다. 과거 표시 이름을 남겨야 하는 요구도 별도로 생각한다.

요청: id / ownerId / title / state

첨부: id / requestId / fileRef

원본 보존과 파생 상태

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x 가 입력을 받고 return이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L05 / 즐겨찾기를 켜고 끄기

같은 ID를 누를 때 선택과 해제가 번갈아 적용되게 한다. 이 사례의 입력 계약은 ids는 선택 ID 목록 / id는 조작할 ID다.

```
const selected = new Set(x.ids);
selected.has(x.id) ? selected.delete(x.id) : selected.add(x.id);
return [...selected];
```

a를 이미 선택한 상태에서 b를 누르면 a와 b를 모두 보관한다. 다음에 a를 누르면 a만 빠지고 b는 남는다. Set은 같은 ID를 두 번 넣어도 하나로 다루지만 실제 자료 목록에 존재하는 ID인지는 추가 판단이다.

입력	기대 결과
<code>{"ids": ["a"], "id": "b"}</code>	<code>["a", "b"]</code>
<code>{"ids": ["a", "b"], "id": "a"}</code>	<code>["b"]</code>

해석의 경계. 같은 ID가 중복 저장되지 않는지 확인한다. 알려진 자료 ID만 선택할 수 있도록 검증을 더한다.

분석 사례 L07 / 분류별 자료 수 세기

항목을 분류별로 묶어 개수를 보여 준다. 이 사례의 입력 계약은 categories는 분류 이름 배열다.

```
return x.categories.reduce((counts, name) => {
  const prev = Object.hasOwn(counts, name) ? counts[name] : 0;
  Object.defineProperty(counts, name, {
    value: prev + 1, writable: true, enumerable: true, configurable: true
  });
  return counts;
}, {});
```

관찰이라는 분류를 처음 만나면 1을 기록하고 다시 만나면 2로 바꾼다. 분류 이름을 객체의 특수 동작이 아니라 자료의 키로 다룬다. 이 예제는 속성을 직접 정의해 기본 속성 이름과의 혼동을 줄인다.

입력	기대 결과
<code>{"categories": ["관찰", "면담", "관찰"]}</code>	<code>{"관찰": 2, "면담": 1}</code>
<code>{"categories": []}</code>	<code>{}</code>

해석의 경계. 객체의 기본 속성 이름과 같은 분류도 데이터로 처리한다. 총개수와 분류별 합계가 같은지 검사한다.

분석 사례 L08 / 태그 중복을 정리하기

중복된 태그와 앞뒤 공백을 정리한다. 이 사례의 입력 계약은 tags는 문자열 배열다.

```
return [...new Set(x.tags.map(t => t.trim()).filter(Boolean))];
```

태그의 앞뒤 공백을 정리한 뒤 빈 항목을 버린다. 그다음 중복을 한 개로 모은다. 과학과 공백이 붙은 과학이 같은 표시로 정리되지만 대소문자와 다른 철자까지 같은 말로 판정하는 것은 아니다.

입력	기대 결과
<code>{"tags": [" 과학", "과학", "", "수업"]}</code>	<code>["과학", "수업"]</code>
<code>{"tags": [" ", ""]}</code>	<code>[]</code>

해석의 경계. 빈 문자열과 공백 문자열을 구분해 정리한다. 영문 대소문자를 같은 태그로 볼지 결정한다.

분석 사례 L11 / 빠진 관찰값을 제외한 평균

기록하지 않은 값과 실제 0을 구분한다. 이 사례의 입력 계약은 values는 숫자 또는 null 배열다.

```
const values = x.values.filter(v => v !== null);
if (values.some(v => !Number.isFinite(v))) throw Error('숫자');
return values.length ? values.reduce((a, b) => a + b, 0) / values
  .length : null;
```

관찰하지 않은 null은 제외하지만 실제 측정 0은 남긴다. 0과 6의 평균은 3이다. 누락까지 0으로 바꾸면 세 값의 평균 2가 되어 같은 관찰을 다르게 해석하게 된다.

입력	기대 결과
<code>{"values": [0, null, 6]}</code>	3
<code>{"values": [null, null]}</code>	null

해석의 경계. 빠진 관찰을 0으로 바꾸면 평균이 달라진다. 표시할 소수 자릿수와 원본 값 보존을 분리한다.

분석 사례 L12 / 완료 비율 표시

완료 수와 전체 수로 비율을 계산한다. 이 사례의 입력 계약은 done과 total은 정수 / 0 이상이며 done은 total 이하이다.

```
if (![x.done, x.total].every(Number.isInteger) ||
    x.done < 0 || x.total < 0 || x.done > x.total) throw Error('범위');
return x.total === 0 ? null : Math.round(x.done / x.total * 100);
```

완료 3을 전체 4로 나눈 뒤 100을 곱하면 75다. 전체가 0이면 분모가 없어 비율을 만들지 않는다. 화면에서는 숫자만 보여 주기보다 전체와 완료의 개수도 함께 읽게 한다.

입력	기대 결과
<code>{"done": 3, "total": 4}</code>	75
<code>{"done": 0, "total": 0}</code>	null

해석의 경계. 전체 0개를 완료 0퍼센트와 같은 의미로 표시할지 정한다. 숫자와 함께 완료 3개 중 4개처럼 분자와 분모도 표시한다.

분석 사례 L17 / 날짜순 관찰 목록

날짜를 같은 형식으로 관리한 자료를 정렬한다. 이 사례의 입력 계약은 items는 date와 value / date는 유효한 YYYY-MM-DD다.

```
return [...x.items].sort((a, b) => a.date.localeCompare(b.date));
```

날짜를 네 자리 연도와 두 자리 월 및 일로 맞췄기 때문에 문자열 순서가 날짜 순서와 일치한다. 2월과 11월을 두 자리 없이 보관하면 같은 비교 규칙을 그대로 적용하기 어렵다.

입력	기대 결과
<pre>{ "items": [{ "date": "2026-11-02", "value": 4 }, { "date": "2026-02-11", "value": 2 }] }</pre>	<pre>[{ "date": "2026-02-11", "value": 2 }, { "date": "2026-11-02", "value": 4 }]</pre>

해석의 경계. 월과 일을 두 자리로 맞추지 않은 입력은 먼저 정규화한다. 같은 날짜의 관찰을 보존할지 덮어쓸지 결정한다.

분석 사례 L19 / 이전 관찰과 변화량

순서가 정해진 수치의 이웃 차이를 계산한다. 이 사례의 입력 계약은 values는 관찰 순서의 유한 숫자 배열다.

```
if (x.values.some(v => !Number.isFinite(v))) throw Error('값');
return x.values.slice(1).map((v, i) => v - x.values[i]);
```

2에서 5로 바뀐 양은 3이다. 다음은 5에서 4로 바뀌었으므로 -1이다. 이 결과는 이웃한 관찰의 차이이며 관찰 간격이 다르면 시간당 변화와 같은 의미가 아니다.

입력	기대 결과
<code>{"values": [2, 5, 4]}</code>	<code>[3, -1]</code>
<code>{"values": [2]}</code>	<code>[]</code>

해석의 경계. 첫 관찰은 이전 값이 없으므로 변화량을 만들지 않는다. 날씨가 빠진 구간을 하루 변화량과 혼동하지 않게 표시한다.

관련 근거 [\[R08\]](#)

10장 API와 분산 시스템의 실패 모델

함수 호출과 네트워크 요청은 실패 방식이 다르다

한 프로세스 안의 함수 호출은 보통 호출자와 피호출자가 같은 실행 환경에 있다. 네트워크 요청은 서버와 통신 경로 및 외부 서비스의 상태에 영향을 받는다. 응답이 늦다는 사실만으로 서버가 작업을 수행하지 않았다고 판단할 수 없다. 요청은 처리되었지만 응답이 돌아오지 않았을 수도 있다. 이 불확실성이 재시도와 중복 처리 문제를 만든다.

Fetch는 HTTP 오류 상태를 자동으로 예외로 바꾸지 않는다. 404나 500 응답을 받은 경우에도 Response가 반환될 수 있으므로 response.ok나 status를 확인해야 한다. 네트워크 자체가 실패한 경우와 서버가 오류 응답을 보낸 경우를 구분해야 사용자가 이해할 수 있는 메시지와 재시도 정책을 만들 수 있다. [R09]

자료 탐색기의 읽기 요청은 실패하면 다시 시도할 수 있지만 공용 요청판의 생성 요청은 같은 행동을 반복할 수 있다. 새 요청을 만들라는 POST를 무조건 재전송하면 항목이 두 개 생길 수 있다. 따라서 읽기와 쓰기에 동일한 재시도 규칙을 적용하지 않는다.

멥등성과 원자성을 함께 이해한다

멥등성은 같은 의도의 요청을 반복해도 의도한 효과가 한 번 수행한 것과 같도록 만드는 성질이다. 생성 작업에서는 클라이언트가 작업 식별자를 보내고 서버가 이미 처리한 식별자의 결과를 재사용하도록 설계할 수 있다. 하지만 식별자를 메모리 배열에서 한 번 확인하는 것만으로 충분하지 않다. 동시 요청 두 개가 모두 미처리 상태를 확인한 뒤 각각 생성할 수 있기 때문이다.

멥등성 키와 결과 기록은 실제 저장소의 고유 제약과 트랜잭션 같은 장치로 결합해야 한다. 키의 유효 기간과 요청 내용이 다른데 같은 키를 사용했을 때의 처리도 정한다. 이 설계는 중복 작업을 줄이지만 외부 서비스와 내부 저장소를 묶은 모든 작업을 자동으로 정확히 한 번 실행하게 만드는 마법은 아니다.

타임아웃은 기다리는 시간을 제한한다. 취소 신호는 클라이언트가 더 이상 결과를 필요로 하지 않는다는 뜻을 전달한다. 이미 서버에서 시작된 작업이 반드시 취소된다는 보장은 없다. 사용자가 취소 버튼을 눌렀을 때 화면을 초기화하는 일과 서버 작업을 중단하는 일은 서로 다른 계약이다.

늦게 도착한 응답이 최신 화면을 덮지 않게 한다

검색어를 과학에서 과학 실험으로 빠르게 바꾸면 첫 요청보다 두 번째 요청이 먼저 끝날 수 있다. 응답이 도착하는 순서대로 화면에 표시하면 마지막에 오래된 결과가 나타난다. 해결 방법 중 하나는 요청마다 증가하는 번호를 붙이고 현재 번호와 일치하는 결과만 반영하는 것이다. 가능하면 이전 요청을 취소하되 응답을 반영할 때도 최신 여부를 확인한다.

이 문제는 AI에게 검색 속도를 개선해 달라고 요청할 때 쉽게 놓친다. 디바운스는 요청 횟수를 줄이지만 이미 진행 중인 요청의 도착 순서를 보장하지 않는다. 서로 다른 기법이 해결하는 문제가 무엇인지 구분해야 한다. 요청 억제와 취소 및 최신 결과 선택은 관련되어 있지만 동일한 기능은 아니다.

API 계약에는 성공 응답뿐 아니라 오류의 구조도 포함한다. 사용자가 고칠 수 있는 입력 오류와 권한 부족 및 일시적인 서버 오류를 구분한다. 로그에는 요청을 연결할 식별자를 남기되 비밀과 민감한 본문을 무조건 기록하지 않는다. 운영자는 실패를 추적할 수 있어야 하고 사용자의 데이터는 필요한 범위를 넘어 노출되지 않아야 한다.

안전한 메서드와 멍등인 메서드는 다른 분류다

HTTP에서 안전성은 클라이언트가 요청한 의미가 서버 상태의 변경을 의도하지 않는다는 성질이다. 멍등성은 같은 요청을 반복했을 때 의도한 효과가 한 번 수행한 것과 같다는 성질이다. GET은 안전하고 멍등인 메서드로 정의된다. PUT과 DELETE는 멍등성이 있지만 안전한 메서드로 분류되지는 않는다. 실제 서버 구현이 이 의미를 지켜야 클라이언트와 중간 시스템이 올바르게 동작할 수 있다. [T17]

DELETE를 반복했을 때 첫 응답은 성공이고 다음 응답은 자원이 없다는 상태일 수 있다. 응답 코드가 같아야 멍등인 것은 아니다. 요청의 의도한 최종 효과가 같은지를 본다. 반대로 POST로 항목을 생성하는 API는 별도 중복 방지 설계가 없다면 반복 요청마다 새 항목이 생길 수 있다.

자료 탐색기의 읽기 재시도는 일시적 실패에 도움이 될 수 있다. 요청판의 저장 재시도에서는 서버가 이미 처리했는지 알 수 없는 상황을 고려한다. HTTP 메서드 이름만 보고 무조건 재시도하는 대신 업무 효과와 멍등성 키 및 서버의 보장 범위를 함께 확인한다.

오류 분류는 사용자 행동과 연결되어야 한다

입력 형식이 잘못된 요청은 같은 내용을 그대로 다시 보내도 성공하기 어렵다. 인증이 만료된 요청은 세션을 갱신해야 할 수 있다. 요청량 제한은 서버가 제공한 대기 정보를 따라야 한다. 일시적 서버 오류는 제한된 재시도가 적합할 수 있다. 모든 실패를 네트워크 오류라는 한 문장으로 처리하면 다음 행동을 결정할 정보가 사라진다.

API 응답의 status와 업무 결과를 함께 읽는다. HTTP 200이지만 본문에 실패 상태를 담는 API도 존재할 수 있다. 이때 계약을 명확히 읽고 처리해야 한다. 새 API를 설계한다면 상태 코드와 응답 구조가 일관되도록 하여 클라이언트가 예외 규칙을 과도하게 외우지 않게 한다.

응답을 받지 못했다는 사실로 처리 실패를 확정할 수 없다

공용 요청판에 새 요청을 보내는 상황을 생각해 보자. 서버가 저장을 완료한 뒤 응답이 네트워크에서 사라질 수 있다. 사용자는 시간 초과만 보지만 데이터베이스에는 이미 요청이 존재한다. 이 상태에서 같은 내용을 다시 보내면 중복이 생길 수 있다. 분산된 실행에서는 호출자의 관찰과 실제 처리 상태가 다를 수 있다는 점이 핵심이다.

이를 해결하려면 작업의 정체성을 내용과 구분할 수 있다. 같은 내용의 요청 두 개가 의도된 것인지 한 작업의 재시도인지는 제목만 비교해서 알 수 없다. 클라이언트가 작업 식별자를 생성하고 서버가 그 식별자에 대한 처리 결과를 기록하는 방식이 한 대안이다. 식별자와 처리 결과를 얼마나 오래 보존할지와 다른 사용자의 식별자가 충돌할 수 있는지도 정책으로 정해야 한다.

이 설계에서 어려운 부분은 식별자를 전달하는 코드보다 기록과 실제 변경을 일관되게 수행하는 것이다. 요청은 저장했는데 처리 기록은 남기지 못했다면 재시도가 다시 실행될 수 있다. 반대로 처리 기록만 남고 요청이 저장되지 않았다면 성공을 잘못 보고할 수 있다. 가능한 경우 두 변경을 같은 트랜잭션 경계에 넣거나 실패를 복구할 수 있는 별도 절차를 둔다.

오래된 응답이 새 화면을 덮어쓰는 문제

검색창에 분을 입력한 뒤 곧바로 분수를 입력하면 두 요청이 나갈 수 있다. 첫 요청이 느리게 도착하고 두 번째 요청이 먼저 끝나면 화면에는 새 결과가 잠시 보였다가 옛 결과로 바뀔 수 있다. 요청을 보낸 순서와 응답이 도착한 순서는 같다고 가정할 수 없다.

간단한 해결은 요청마다 순번을 부여하고 현재의 최신 순번과 일치하는 결과만 적용하는 것이다. 이전 요청을 취소하는 기능을 함께 사용할 수도 있지만 취소가 서버의 처리를 반드시 중단한다는 뜻은 아니다. 화면의 최신성 보호와 서버 작업의 취소는 별개의 계약이다. 검색은 최신 결과만 보여 주면 되지만 저장 작업은 취소 표시 이후 실제 상태를 다시 확인해야 할 수도 있다.

```
let latest = 0;
async function searchRemote(query) {
  const current = ++latest;
  const response = await fetch('/search?q=' + encodeURIComponent(
    query));
  if (!response.ok) throw new Error('검색 실패');
  const data = await response.json();
```

```
if (current !== latest) return;  
renderResults(data);  
}
```

이 코드는 순서 문제의 한 측면만 보여 준다. data의 형식 검증과 로딩 상태 및 오류 표시가 생략되어 있다. 이전 요청의 오류가 최신 화면에 표시되지 않게 하려면 오류 처리에서도 같은 순번을 검사해야 한다. 성공 경로에만 규칙을 넣으면 오래된 실패가 새 성공을 가릴 수 있다. 원리는 모든 완료 경로에 일관되게 적용해야 한다.

재시도에는 멈추는 조건이 필요하다

일시적인 연결 문제는 재시도로 해결될 수 있지만 잘못된 입력과 거부된 권한은 같은 요청을 반복해도 해결되지 않는다. 실패를 분류하지 않고 모든 오류를 재시도하면 서버 부하와 사용자의 대기 시간만 늘 수 있다. 재시도 가능한 오류와 최대 횟수 및 총 대기 시간을 정한다.

여러 사용자가 동시에 같은 주기로 다시 요청하면 복구 중인 서버에 부하가 몰릴 수 있다. 대기 시간을 늘리고 약간의 무작위 차이를 두는 방식은 이런 동시 집중을 완화하는 설계 선택이다. 그러나 재시도 횟수를 무한히 늘리거나 성공을 보장하는 장치로 설명해서는 안 된다. 최종 실패를 사용자에게 알리고 나중에 상태를 확인하는 경로도 필요하다.

이 책의 순수 함수 예시는 중복 제거와 상태 판정 규칙을 관찰하기 위한 모형이다. 실제 네트워크의 시간 초과와 프로세스 종료 및 저장소 장애를 재현한 것은 아니다. 공용 시스템으로 확장할 때는 위의 각 시간 지점에서 실패를 주입하고 관찰 결과가 설계와 일치하는지 시험해야 한다. 작은 모형에서 얻은 이해를 실제 경계의 검사로 옮기는 것이 다음 단계다.

저장하지 않는 것과 재사용 전에 확인하는 것은 다르다

HTTP의 no-store와 no-cache는 같은 뜻이 아니다. no-store는 저장하지 않도록 하는 지시이며 no-cache는 저장된 응답을 재사용하기 전에 유효성을 확인하도록 하는 의미를 가진다. private은 공유 캐시의 저장을 제한하는 맥락에서 사용된다. 이름의 인상으로 설정을 고르지 않고 요구하는 동작을 기준으로 선택한다. [T30]

자료 목록처럼 여러 사용자가 같은 값을 읽는 응답과 개인의 비공개 요청 목록은 캐시 정책이 다를 수 있다. 개인 응답이 잘못된 공유 캐시에 들어가면 성능 문제가 아니라 정보 노출 문제가 된다. URL이 같다는 이유만으로 같은 사용자 문맥의 응답이라고 가정하지 않는다. 인증과 캐시 키 및 헤더의 관계를 확인한다.

ETag 같은 검증자는 저장된 표현이 여전히 유효한지 확인하는 데 쓰일 수 있다. 검증자가 무엇을 대표하는지와 강한 검증자 및 약한 검증자의 차이를 API 계약에서

읽는다. 조건부 요청이 성공했다는 사실과 모든 업무 자료가 최신이라는 주장은 범위가 다를 수 있다. 응답 표현과 그 뒤의 여러 자원이 항상 같은 단위는 아니다.

화면 갱신과 데이터 갱신의 순서를 설계한다

사용자가 요청을 저장했는데 목록에 이전 값이 보이면 실제 저장이 실패한 것인지 캐시가 오래된 것인지 구분해야 한다. 저장 응답의 값을 즉시 반영할지 다시 조회할지와 어떤 캐시를 무효화할지 정한다. 같은 데이터를 여러 화면에서 읽는다면 한 화면의 수정이 다른 화면에 어떻게 전달되는지도 설계한다.

오래된 자료를 잠시 보여 주면서 새 자료를 받는 방식은 빠른 반응을 줄 수 있다. 그러나 잔여 수량이나 제출 상태처럼 잘못된 최신성 판단의 비용이 큰 자료에는 다른 정책이 필요할 수 있다. 사용자가 현재 값의 신선도를 알아야 한다면 갱신 시각이나 동기화 상태를 표시한다. 성능 개선이 자료의 의미를 숨기지 않게 한다.

재시도 예산과 시간 제한을 상위 작업에 맞춘다

하나의 사용자 행동이 여러 내부 요청으로 이루어져 있다면 각 요청에 긴 재시도를 적용할 때 전체 시간이 과도하게 늘 수 있다. 상위 작업이 허용하는 총시간을 먼저 정하고 하위 요청에 예산을 배분한다. 어느 단계가 오래 걸렸는지 기록해야 시간 제한의 원인을 찾을 수 있다.

호출자가 포기한 뒤에도 서버의 처리가 계속될 수 있다. 따라서 시간 초과를 취소와 동일하게 표현하지 않는다. 저장 작업에서는 작업 식별자로 상태를 다시 확인하는 경로가 필요할 수 있다. 네트워크의 불확실성을 사용자에게 전달할 때는 실패가 확정된 경우와 처리 여부를 아직 모르는 경우를 구분한다.

상태 변화 / 늦은 응답이 최신 검색을 덮지 않게 한다

현재 입력

과학

요청 번호

1

도착 응답

없음

화면 반영

대기

네트워크 응답은 요청 순서대로 도착하지 않을 수 있다. 디바운스만으로 이 순서 문제를 해결할 수는 없다.

현재 입력

과학 실험

요청 번호

2

도착 응답

요청 1 결과

화면 반영

무시

네트워크 응답은 요청 순서대로 도착하지 않을 수 있다. 디바운스만으로 이 순서 문제를 해결할 수는 없다.

네트워크 응답은 요청 순서대로 도착하지 않을 수 있다. 디바운스만으로 이 순서 문제를 해결할 수는 없다.

- 1. 첫 검색 요청을 보낸다
- 2. 입력이 바뀌어 새 요청을 보낸다
- 3. 최신 요청이 먼저 끝나면 반영한다
- 4. 오래된 응답은 도착해도 반영하지 않는다

사례 | 없는 자료를 읽고도 성공으로 표시

자료 읽기 요청이 404 응답을 받았다. 코드는 응답이 왔다는 이유로 성공 안내를 보냈다. Fetch는 이런 HTTP 오류에서도 Response를 돌려줄 수 있으므로 상태를 확인해야 했다. 통신 예외와 서버의 거절 및 데이터 형식 오류를 나누어 처리했다.

```

async function readResources() {
  const response = await fetch('/api/resources');
  if (!response.ok) {
    throw new Error(`HTTP ${response.status}`);
  }
  return await response.json();
}

```

위 주소는 구조 설명용이다. 이 책의 정적 실행 예제에는 해당 서버가 없다. 실제로 호출하려면 그 주소의 API를 구현해야 한다. 빈 배열과 잘못된 JSON도 화면에서 어떻게 다룰지 정한다.

CORS 오류를 볼 때

다른 출처의 서버 응답을 브라우저 코드가 읽을 수 있는지는 서버의 허용 설정과 요청 방식의 영향을 받는다. 브라우저의 보호를 끄는 일을 서비스 해결책으로 삼지 않는다. CORS는 로그인과 자원별 권한을 대신하는 장치도 아니다.

결제나 신청을 다시 보낼 때

조회 재시도와 쓰기 재시도는 영향이 다르다. 중복 등록이 가능한 요청이라면 서버에서 요청의 중복을 어떻게 다룰지 정한다. 사용자에게 다시 누르라고 안내하기 전에 실제 처리 상태를 조사한다.

적용과 확인 | 구현 전 API 계약 표

1. 자료 목록 읽기와 새 요청 등록을 구분한다.
2. 정상 응답의 가짜 JSON을 만든다.
3. 빈 목록과 거절 응답을 추가한다.
4. 느린 응답 중에 버튼이 어떻게 보일지 정한다.
5. 외부 서비스가 멈췄을 때의 안내를 쓴다.

상황	화면의 기대 행동
읽는 중	진행 상태와 중복 행동 방지
자료 없음	오류와 구분한 빈 목록 안내
권한 없음	접근 조건 안내와 재인증 경로

상황	화면의 기대 행동
통신 실패	입력 보존과 재시도 방법
등록 결과 불명	중복 전송 전 처리 여부 확인

분야별 적용 예

연수 신청은 중복 접수를 검사한다. 자료 번역은 응답 대기과 실패를 알린다. 재고 조치는 자료의 기준 시점을 표시한다. 외부 응답을 무조건 사실로 쓰기보다 형식과 범위를 확인한다.

검토 질문 API 키를 입력창에서 받아 숨김 처리하면 서버 비밀을 보호할까? **해설** 입력 표시를 가리는 것과 실행 위치에서 비밀을 보호하는 것은 다르다. 누가 그 값을 읽고 전송하고 보관하는지까지 확인한다.

기술 해설 / HTTP 상태와 업무 결과를 나눈다

서버가 답했다는 것과 요청이 허용됐다는 것은 다르다. 상태 코드와 응답 본문 및 실제 저장 결과를 확인한다. 성공 범위의 응답이라도 필요한 필드가 없으면 업무 결과를 완성할 수 없다.

신청 버튼 뒤에 성공 문구를 띄우기 전에 서버의 거절과 등록 ID를 확인한다. 결과가 불명확할 때는 중복 재전송을 조심한다.

```
const response = await fetch('/api/resources');
if (!response.ok) throw Error(`HTTP ${response.status}`);
```

기술 해설 / 기다리는 동안의 화면도 설계한다

비동기 요청에는 대기과 성공 및 실패가 있다. 그동안 입력을 보존할지와 중복 실행을 막을지 정한다. 이전 요청의 늦은 응답이 새로운 검색 결과를 덮어쓰지 않는지도 살핀다.

검색어를 빠르게 바꾸었을 때 예전 검색의 응답이 나중에 도착할 수 있다. 요청의 순서나 취소를 관리하는 구조를 검토한다.

```
let requestId = 0;
async function load() {
  const mine = ++requestId;
  const data = await readData();
  if (mine === requestId) show(data);
}
```

실패 응답과 중복 요청

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 `x`가 입력을 받고 `return`이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L30 / HTTP 응답 상태의 안내

성공과 권한 문제 및 서버 문제를 나누어 표시한다. 이 사례의 입력 계약은 `status`는 HTTP 응답 상태 숫자다.

```
if (x.status >= 200 && x.status < 300) return '성공';
if (x.status === 401) return '인증 확인';
if (x.status === 403) return '권한 확인';
if (x.status === 404) return '대상 확인';
if (x.status >= 500 && x.status < 600) return '서버 오류';
return '응답 검토';
```

200은 상태 차원의 성공이고 404는 대상 주소나 자원을 확인할 근거다. 503은 서버 쪽 문제를 조사한다. 같은 상태에서도 응답 본문과 실제 업무 결과는 추가로 읽어야 한다.

입력	기대 결과
<code>{"status": 200}</code>	"성공"
<code>{"status": 404}</code>	"대상 확인"
<code>{"status": 503}</code>	"서버 오류"

해석의 경계. 상태만으로 업무 처리 결과 전체를 단정하지 않는다. 통신 예외와 응답 본문 형식 오류를 별도 안내한다.

분석 사례 L31 / 재시도할 요청을 고르기

조회 요청에 한정해 정한 조건의 재시도 모형을 만든다. 이 사례의 입력 계약은 `method`와 `attempts` 및 `transient / attempts`는 이미 시도한 횟수다.

```
return x.method === 'GET' && x.transient === true &&
  Number.isInteger(x.attempts) && x.attempts >= 0 && x.attempts <
    3;
```

GET이고 일시적 실패로 분류했으며 정한 횟수보다 덜 시도한 경우에만 이 모형은 재시도를 허용한다. POST를 같은 방식으로 반복하면 중복 신청 같은 영향을 만들 수 있다.

입력	기대 결과
<code>{"method": "GET", "attempts": 1, "transient": true}</code>	<code>true</code>
<code>{"method": "POST", "attempts": 1, "transient": true}</code>	<code>false</code>
<code>{"method": "GET", "attempts": 3, "transient": true}</code>	<code>false</code>

해석의 경계. 조회라 해도 모든 실패를 재시도하면 안 된다. 지연과 취소 및 실제 오류 종류에 맞는 정책을 추가한다.

분석 사례 L49 / 같은 요청 키의 재처리 모형

이미 처리한 가짜 요청 키가 다시 오면 같은 기록을 돌려준다. 이 사례의 입력 계약은 done은 키와 결과의 배열 / key와 value는 새 요청다.

```
const found = x.done.find(item => item.key === x.key);
return found ? {reused:true, value:found.value} : {reused:false,value:x.value};
```

같은 키로 처리한 결과가 있으면 그 결과를 다시 돌려주는 모형이다. 이전 결과 r1이 있는데 r2가 새로 들어와도 r1을 반환한다. 실제 서버의 동시 처리와 키의 보관 및 본문 충돌 정책은 추가 구현이다.

입력	기대 결과
<code>{"done": [{"key": "k1", "value": "r1"}], "key": "k1", "value": "r2"}</code>	<code>{"reused": true, "value": "r1"}</code>
<code>{"done": [], "key": "k2", "value": "r2"}</code>	<code>{"reused": false, "value": "r2"}</code>

해석의 경계. 실제 서버의 동시 요청과 키 보관 및 원자적 저장은 구현하지 않은 모형이다. 같은 키에 다른 본문이 오면 거절할지의 정책을 더한다.

관련 근거 [\[R09\]](#) [\[R10\]](#)

11장 신뢰 경계와 접근 제어

인증된 사용자도 모든 자료에 접근할 수는 없다

인증은 요청 주체가 누구인지 확인하는 과정이다. 인가는 그 주체가 특정 자원에 특정 행동을 할 수 있는지 판단하는 과정이다. 로그인 화면을 추가했다고 권한 설계가 끝나는 것은 아니다. 공용 요청판에서 사용자가 자신의 요청을 볼 수 있다는 조건과 팀 전체 요청을 볼 수 있다는 조건은 서로 다르다. 읽기 권한이 있다고 수정 권한까지 주어지는 것도 아니다. [R11] [R12]

신뢰 경계는 어느 쪽의 값을 그대로 믿을 수 없는지를 보여 주는 경계다. 브라우저는 사용자가 제어할 수 있는 환경이다. 화면에서 관리자 버튼을 숨기더라도 직접 요청을 보내는 행동을 막지는 못한다. 클라이언트가 전달한 role이나 ownerId를 그대로 신뢰하면 사용자가 값을 바꾸어 다른 사람의 권한을 주장할 수 있다. 서버는 검증된 세션에서 주체를 얻고 실제 자원의 소유 관계와 권한을 확인해야 한다.

Saltzer와 Schroeder의 고전적인 보안 설계 원칙은 최소 권한과 기본 거부 및 모든 접근의 검사 같은 생각을 제시한다. 이 원칙을 요청판에 적용하면 사용자가 필요한 행동만 수행하도록 허용하고 명시적으로 허용되지 않은 행동은 거부하며 각 요청에서 권한을 확인하는 설계가 된다. 원칙은 특정 프레임워크의 설정 목록이 아니라 설계를 평가하는 질문이다. [S07]

권한 모델을 행렬과 부정 사례로 표현한다

요청판의 역할을 작성자와 담당자 및 관리자로 나누었다고 하자. 역할 이름만으로 허용 범위를 설명하기 어렵다. 작성자는 자신의 접수 상태 요청을 수정할 수 있고 담당자는 자신에게 배정된 요청을 진행으로 옮길 수 있다고 구체화한다. 이때 자원의 소유자와 상태 및 조직 범위가 모두 권한 판단의 입력이다.

권한 테스트에는 허용 사례보다 거부 사례가 더 많은 경우도 있다. 다른 사용자의 ID로 바꾼 요청과 다른 조직의 요청 및 만료된 세션을 사용한 요청을 각각 확인한다. 존재하지 않는 자원과 접근할 수 없는 자원을 어떤 응답으로 표현할지도 정책이다. 내부 오류 내용을 그대로 반환하면 자원의 존재나 시스템 구조를 불필요하게 드러낼 수 있다.

권한 캐시는 성능을 높일 수 있지만 권한 변경이 언제 반영되는지 문제가 생긴다. 관리자가 사용자를 팀에서 제거했는데 이전 캐시가 계속 허용한다면 정책과 실제 동작 사이에 시간 차이가 존재한다. 즉시 철회가 필요한 권한과 일정 지연을 허용할 수 있는 권한을 구분해야 한다. 캐시 만료 시간은 기술 설정이면서 업무 위험의 선택이다.

생성된 보안 코드의 자신감과 실제 안전성을 분리한다

Perry 등의 사용자 연구는 당시 특정 AI 보조 도구를 사용한 참여자들이 보안 관련 과제에서 더 취약한 코드를 작성하면서도 자신의 코드가 안전하다고 믿는 경향을 보고했다. 이 결과는 연구의 모델과 과제 조건에 한정된다. 최신 도구 전체의 취약성 비율을 말해 주지는 않는다. 다만 결과에 대한 자신감과 실제 검증 결과를 분리해야 한다는 문제를 보여 준다. [S08]

실무에서는 AI에게 보안을 점검해 달라는 요청 하나를 완료 기준으로 삼지 않는다. 접근 제어는 실제 서버 요청으로 확인하고 입력 처리는 위험한 문자열과 잘못된 형식으로 확인한다. 민감한 데이터가 로그나 응답 및 정적 번들에 들어갔는지도 조사한다. 기능별로 어떤 경계를 검사했는지 남겨야 검토자가 확인 범위를 이해할 수 있다.

이 책의 권한 순수 함수는 정책의 핵심을 설명하기 위한 실행 모형이다. 실제 인증 서비스와 서버 미들웨어 및 데이터베이스 권한 설정을 구현한 것은 아니다. 모형을 통과했다는 결과를 운영 서비스가 안전하다는 결론으로 확대하지 않는다. 추상 모형의 검증과 실제 배포 환경의 검증은 서로 연결되지만 별도의 증거를 요구한다.

보호할 자산과 경계를 먼저 적는다

위협 모형은 공격 기법의 목록을 길게 외우는 작업으로 시작하지 않는다. 어떤 자료를 보호해야 하며 누가 어떤 작업을 할 수 있고 어느 경계를 통해 입력이 들어오는지 적는다. 공용 요청판에서는 요청 내용과 작성자 정보 및 관리자 권한과 변경 이력이 자산이 된다. 공개된 화면과 API 및 데이터베이스와 배포 자동화는 서로 다른 신뢰 경계를 가진다.

정적 자료 탐색기에는 서버 계정이 없을 수 있지만 안전 문제 자체가 사라지지는 않는다. 외부 주소를 표시하는 방식과 가져온 JSON을 해석하는 방식 및 사용자 입력을 HTML로 넣는 방식이 검토 대상이다. 보호해야 할 자산과 실패 영향이 공용 요청판보다 작을 수 있다는 뜻이지 검토가 불필요하다는 뜻은 아니다.

경계를 그릴 때는 데이터가 이동하는 방향과 권한을 함께 적는다. 브라우저에서 서버로 넘어온 ownerId는 사용자가 보낸 값이다. 서버가 인증된 세션에서 얻은 사용자 식별자와 같은 수준으로 신뢰하면 안 된다. 클라이언트가 소유자를 주장하는 것과 서버가 소유자를 확인하는 것은 다른 행동이다.

역할만으로 모든 권한을 표현할 수 있는가

작성자과 검토자 및 관리자의 역할을 정의하는 것은 시작점이다. 같은 작성자 역할을 가진 두 사람이 서로의 요청을 수정할 수 있는지는 별도의 규칙이다. 권한 판정에는 주체의 역할뿐 아니라 자원의 소유자와 현재 상태 및 수행하려는 행동이 필요할 수 있다. 이 관계를 빠뜨리면 화면에 로그인한 사람만 접근할 수 있어도 다른 사람의 자료를 바꿀 수 있다.

주체와 자원	읽기	상태 변경	삭제
본인 요청의 작성자	정책에 따라 허용	허용된 전이만 가능	보존 정책에 따라 제한
다른 요청의 작성자	공개 범위에 따라 결정	기본적으로 거부	거부
담당 검토자	담당 범위에서 허용	검토 단계만 허용	별도 권한 필요

이 표는 예시 정책이며 보편적인 정답이 아니다. 중요한 것은 빈칸을 구현자의 추측으로 남기지 않는 것이다. 삭제 권한이 모호하면 생성 모델이 편의상 관리자에게 모든 권한을 줄 수 있다. 실제로는 관리자도 감사 이력이 필요한 자료를 즉시 삭제하면 안 될 수 있다. 역할 이름의 권위보다 업무 규칙을 먼저 정의한다.

권한 검사는 읽기와 쓰기 모두에 적용한다

쓰기 요청만 보호하고 조회 요청은 공개하면 민감한 정보가 노출될 수 있다. 목록에서는 항목을 숨겼어도 ID를 직접 지정한 상세 API가 다른 사용자의 자료를 반환할 수 있다. 목록 필터와 상세 조회 및 수정과 삭제를 같은 정책으로 검토해야 한다. 화면의 메뉴가 보이지 않는다는 사실은 서버의 거부를 입증하지 않는다.

검사 순서도 외부에 노출하는 정보에 영향을 준다. 존재하지 않는 자원과 접근할 수 없는 자원에 서로 다른 답을 주면 자원의 존재를 추정할 수 있다. 어떤 서비스에서는 이 차이를 허용하고 어떤 서비스에서는 같은 응답으로 처리한다. 일관된 정책과 사용자 경험을 함께 고려해야 하며 예외를 즉흥적으로 추가하지 않는다.

권한 시험의 핵심은 허용 사례와 거부 사례를 함께 실행하는 것이다. 본인 요청 수정이 성공했는지만 확인하면 다른 사람의 요청이 거부되는지 알 수 없다. 만료된 세션과 삭제된 사용자 및 변경된 역할도 중요한 경계다. 각 시험에서 데이터가 실제로 바뀌지 않았다는 사실까지 확인하면 단순한 오류 메시지 확인보다 강한 증거가 된다.

브라우저가 자동으로 보내는 자격 증명에는 별도의 경계가 있다

쿠키 기반 인증에서는 브라우저가 요청에 자격 증명을 자동으로 붙일 수 있다. 이 때문에 로그인한 사용자의 브라우저에서 요청이 왔다는 사실만으로 사용자가 의도한 변경이라고 판단할 수 없다. CSRF 방어는 이런 경로를 다루며 프레임워크의 검증 기능과 토큰 및 출처 확인 같은 통제를 사용한다. SameSite는 보완 수단이며 대부분의 배포에서 단독으로 모든 방어를 대신한다고 가정하지 않는다. [T36]

요청판에서 상태 변경을 GET 링크로 구현하면 단순한 이동이나 자동 접근이 자료를 바꾸는 경로가 될 수 있다. 읽기와 변경의 의미를 HTTP 메서드 및 서버 정책에 맞게 설계한다. 화면에서 확인창을 띄웠다는 사실은 서버에 직접 들어오는 요청의 검증을 대신하지 않는다.

CORS는 브라우저에서 다른 출처의 응답을 읽는 관계를 통제하는 기능이다. 인증과 권한의 대체물로 사용하지 않는다. 특정 출처만 허용했더라도 서버가 사용자의 권한을 검사해야 한다. 브라우저 정책과 서버 정책이 보호하는 질문을 구분하면 보안 설정의 역할을 더 정확히 읽을 수 있다.

에이전트가 읽은 자료를 새로운 명령으로 승격시키지 않는다

AI가 저장소 문서와 웹페이지 및 이슈 내용을 읽을 때 그 자료에는 작업과 무관한 지시가 들어 있을 수 있다. 검색된 내용과 도구 출력은 분석할 데이터이며 자동으로 사용자의 명령이 되지 않는다. OWASP는 도구 호출을 사용자 권한과 세션 문맥에 맞춰 검증하고 모델 밖에서 최소 권한 통제를 유지하는 방식을 강조한다. [T35]

예를 들어 문서에 환경 변수 전체를 외부 주소로 전송하라는 문장이 있더라도 자료 탐색기 수정이라는 목표에 필요한 행동이 아니다. 모델이 그 문장을 설득력 있게 해석하더라도 실행 계층은 허용된 자원과 행동의 범위를 유지해야 한다. 위험한 단어를 찾아 지우는 필터 하나로 이 문제를 모두 해결한다고 약속하지 않는다.

저장소 자동화에서는 읽기와 수정 및 외부 전송의 권한을 분리하고 실제 수행한 도구 호출을 기록한다. 필요한 범위에서 테스트 환경을 제공하면 생성 코드의 실행 결과를 확인하면서 영향 범위를 제한할 수 있다. 권한 설계와 관찰 가능성은 에이전트의 능력을 실무에 연결하는 기반이다.

상태 변화 / 서버는 주체와 자원과 행동을 함께 검사한다

서버는 주체와 자원과 행동을 함께 검사한다

1 / 4 정상 허용 사례를 확인한다

검증된 주체
사용자 A

요청 자원
A의 요청

하려는 행동
읽기

서버 결정
허용

권한 정책을 설명하는 모형이다. 실제 서비스에서는 브라우저 값이 아닌 검증된 세션과 서버의 자원 관계로 판단한다.

서버는 주체와 자원과 행동을 함께 검사한다

4 / 4 로그인 여부와 자원 권한은 다른 조건이다

검증된 주체
미인증

요청 자원
A의 요청

하려는 행동
수정

서버 결정
거부

권한 정책을 설명하는 모형이다. 실제 서비스에서는 브라우저 값이 아닌 검증된 세션과 서버의 자원 관계로 판단한다.

권한 정책을 설명하는 모형이다. 실제 서비스에서는 브라우저 값이 아닌 검증된 세션과 서버의 자원 관계로 판단한다.

- 1. 정상 허용 사례를 확인한다
- 2. 다른 사용자의 자원을 직접 요청한다
- 3. 조직 범위도 별도로 확인한다
- 4. 로그인 여부와 자원 권한은 다른 조건이다

사례 | 숨겨 둔 관리자 버튼

업무 요청판에서 관리자 버튼을 숨겼다. 그러나 서버의 삭제 API가 로그인한 모든 사용자에게 허용돼 있었다. 버튼을 보이지 않게 한 것은 화면의 선택이었으며 삭제 권한을 보호한 것은 아니었다. 서버 요청의 역할과 자원 검사를 추가해야 했다.

시험 계정과 요청	기대 결과
작성자 A가 자신의 요청 수정	허용
작성자 A가 B의 요청 수정	거절
담당자가 담당 밖 요청 처리	정책에 따라 거절
로그아웃 상태의 삭제 요청	거절
잘못된 역할 문자열 전송	서버가 임의 주장을 믿지 않음

실패를 보여 주는 안내

접근이 거절됐을 때 사용자의 입력을 함부로 지우지 않는다. 다시 로그인할지와 담당자에게 문의할지의 경로를 알려 준다. 구체적인 내부 비밀이나 다른 사람의 존재 여부를 불필요하게 노출하지 않도록 응답 범위도 검토한다.

이 책에서 구현하는 범위

정적 자료 탐색기는 실제 로그인과 서버 권한을 구현하지 않는다. 이 장의 역할 표는 공용 서비스를 설계할 때 사용할 기준이다. 회원가입 화면만 덧붙여 실제 인증이 완성됐다고 설명하지 않는다.

적용과 확인 | 두 계정으로 역할 표 시험

1. 실제 개인정보 대신 가짜 사용자 A와 B를 정한다.
2. 각자의 가짜 요청을 하나씩 만든다고 가정한다.
3. 읽기와 수정 및 삭제 요청을 역할 표로 정리한다.
4. 화면 경로와 직접 요청의 기대 결과를 각각 적는다.
5. 구현한 경우에만 실제 시험 결과를 별도 열에 남긴다.

기록	작성 예
시험 이름	A가 B의 요청 수정
대상	가짜 요청 ID request-b

기록	작성 예
기대	거절하며 원래 자료 유지
실제	아직 구현하지 않음
다음 일	서버의 소유자 검사 구현

다른 예시로 확장

독서 모임에서는 개인 메모와 공용 추천을 구분한다. 상담 신청에서는 신청자와 담당자의 읽기 범위를 다르게 정한다. 선거 공약 자료실에서는 공개 공약과 비공개 관리 자료를 분리한다. 역할 이름이 같아도 자료의 성격에 따라 허용 범위는 달라질 수 있다.

검토 질문 브라우저에서 role 값을 admin으로 바꾸면 관리자가 되어도 될까? **해설** 서버가 브라우저의 역할 주장만 믿어서는 안 된다. 검증된 신원과 서버의 정책으로 허용 여부를 판단해야 한다.

기술 해설 / 권한은 자원과 행동의 관계다

역할만 확인하면 담당 밖 자료까지 열 수 있다. 검증된 주체와 대상 자원 및 요청 행동을 함께 읽어야 한다. 정책에서 허용한 관계가 아니면 거절하는 기준을 세우고 실제 요청에서 적용한다.

작성자 A가 B의 요청 ID를 알아도 수정할 수 없어야 한다. 브라우저의 역할 문자열은 서버가 검증한 신원과 같지 않다.

```
const allowed = user.id === record.ownerId;
if (!allowed) throw Error('거절');
```

주체와 자원 및 역할

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x가 입력을 받고 return이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L21 / 자기 기록 읽기 규칙

인증된 주체와 소유자 관계로 읽기 허용 모형을 만든다. 이 사례의 입력 계약은 user는 ID 또는 null / owner는 자원 소유자 ID다.

```
return typeof x.user === 'string' && x.user.length > 0 && x.user
=== x.owner;
```

A의 신원이 확인됐다고 가정한 뒤 자원의 소유자도 A인지 본다. B의 자료라면 ID를 알고 있어도 거절해야 한다. 이 작은 함수는 로그인 절차와 요청의 신원 검증 자체를 제공하지 않는다.

입력	기대 결과
<code>{"user": "A", "owner": "A"}</code>	true
<code>{"user": "A", "owner": "B"}</code>	false
<code>{"user": null, "owner": "A"}</code>	false

해석의 경계. 이 함수는 실제 인증을 수행하지 않는 판단 모형이다. 실제 API에서는 검증된 세션의 ID로만 주체를 결정한다.

분석 사례 L22 / 허용 역할의 수정 규칙

정한 역할 목록에 있을 때만 수정하는 모형을 만든다. 이 사례의 입력 계약은 role은 서버가 확인했다고 가정한 역할다.

```
return ['editor', 'admin'].includes(x.role);
```

정확히 editor이거나 admin인 역할만 허용한다. superadmin처럼 비슷한 글자가 있어도 목록에 없으면 거절한다. 실제 역할은 브라우저가 써 보낸 값을 그대로 믿는 방식으로 정하면 안 된다.

입력	기대 결과
<code>{"role": "editor"}</code>	true
<code>{"role": "visitor"}</code>	false
<code>{"role": "superadmin"}</code>	false

해석의 경계. 역할 이름에 admin 글자가 들어간다는 이유로 허용하지 않는다. 이 모형과 실제 서버의 자원별 권한을 함께 확인한다.

분석 사례 L23 / 담당 범위 안의 요청

담당자와 프로젝트의 일치 여부를 함께 검사한다. 이 사례의 입력 계약은 `role`과 `assignedProject` 및 `targetProject`다.

```
return x.role === 'reviewer' &&  
    x.assignedProject === x.targetProject;
```

검토자라는 역할과 대상 프로젝트의 일치 여부를 함께 묻는다. p1 담당자는 p2의 자료를 자동으로 읽는 사람이 아니다. 담당 관계를 바꾼 뒤에도 예전 접근이 남는지 실제 요청에서 확인한다.

입력	기대 결과
<code>{"role": "reviewer", "assignedProject": "p1", "targetProject": "p1"}</code>	<code>true</code>
<code>{"role": "reviewer", "assignedProject": "p1", "targetProject": "p2"}</code>	<code>false</code>

해석의 경계. 담당자라는 역할만 보고 모든 프로젝트를 열지 않는다. 담당 관계가 회수됐을 때 접근도 바뀌는지 시험한다.

관련 근거 [\[R11\]](#) [\[R12\]](#)

12장 테스트와 디버깅의 추론 구조

오류를 고치기 전에 설명 가능한 실패를 만든다

디버깅은 코드를 무작정 바꾸는 일이 아니라 관찰된 실패를 설명하는 가설을 좁히는 과정이다. 검색 결과가 없다는 보고에는 자료가 실제로 없는 경우와 필터가 잘못된 경우 및 화면 갱신이 누락된 경우가 섞일 수 있다. 같은 입력과 같은 환경에서 실패를 반복할 수 있게 만들면 가설을 비교할 기준이 생긴다.

최소 재현 사례는 문제를 유지하면서 불필요한 요소를 줄인 사례다. 자료 탐색기 전체에서 오류가 나더라도 자료 두 개와 검색 함수만으로 재현할 수 있다면 조사 범위가 작아진다. 다만 요소를 제거하는 과정에서 실패 원인까지 없애면 다른 문제를 검증하게 된다. 원래 실패와 최소 사례의 실패가 같은 조건에서 발생하는지 확인한다.

시에게는 관찰과 해석을 구분해 전달한다. 과목을 바꾸면 결과 수가 이전 값으로 남는다는 것은 관찰이다. 렌더 함수가 호출되지 않는 것 같다는 것은 가설이다. 두 문장을 분리하면 모델이 사용자의 추측을 사실로 받아들여 잘못된 곳만 수정하는 일을 줄일 수 있다.

테스트의 계층과 비용을 맞춘다

단위 테스트는 작은 계산이나 규칙을 빠르게 확인한다. 통합 테스트는 여러 구성 요소의 연결을 확인한다. 종단 간 테스트는 사용자의 실제 흐름을 통해 시스템을 확인한다. 자료 탐색기의 제목 검색 규칙은 단위 테스트에 적합하고 저장 후 다시 열었을 때의 복원은 브라우저와 저장소를 함께 보는 시험이 필요하다.

모든 검사를 브라우저 자동화로 작성하면 실행 시간이 늘고 환경 변화에 민감해질 수 있다. 반대로 순수 함수 테스트만 있으면 이벤트 연결과 접근성 및 실제 배포 경로의 문제를 놓칠 수 있다. 테스트 계층의 선택은 도구의 유행보다 어떤 종류의 실패를 잡으려는지에 따라 결정한다. Playwright의 권고 역시 사용자가 관찰하는 행동과 안정적인 선택자를 중심으로 시험을 설계하는 데 도움이 된다. [R06]

경계값은 특히 유용하다. 페이지 크기가 10일 때 자료 수가 0과 1 및 10과 11인 경우를 비교하면 페이지 계산의 오류를 찾기 쉽다. 날짜 구간이 맞는 경우에는 끝 시각을 포함하는지 여부가 중요하다. 정렬에서는 같은 제목과 한글 및 숫자가 섞인 자료를 확인한다. 경계값의 의미는 요구에서 나오며 시가 임의로 고른 숫자에서 나오지 않는다.

테스트가 통과했다는 사실의 한계

코드 커버리지는 실행된 코드의 범위를 보여 주지만 기대 결과의 정확성을 보장하지 않는다. 모든 줄을 실행해도 잘못된 값을 정답으로 비교하면 테스트는 통과한다. 반대로 중요한 권한 분기 하나가 검사되지 않았다면 전체 커버리지가 높아도 위험이 남는다. 수치를 단독 목표로 삼기보다 어떤 실패를 탐지할 수 있는지 설명해야 한다.

속성 기반 사고는 특정 사례를 넘어 규칙의 성질을 묻는다. 정렬 결과의 길이는 입력과 같아야 하고 원본 배열이 바뀌지 않아야 한다. 즐겨찾기 토글을 같은 ID에 두 번 적용하면 원래 상태로 돌아와야 한다. 이러한 성질은 많은 입력에 적용할 수 있지만 잘못된 정의의 속성 역시 잘못된 오라클이 된다.

SWE-bench는 실제 저장소의 이슈와 수정 과제를 이용해 코드베이스 변경 능력을 평가하도록 설계되었다. 원 연구는 12개 Python 저장소에서 얻은 2294개 문제를 제시했다. 이 벤치마크의 점수는 정해진 평가 환경에서의 문제 해결 성능이며 새로운 제품의 요구 정의와 배포 운영까지 완수하는 능력을 직접 측정하는 것은 아니다. [S09]

따라서 모델의 공개 점수는 도구 선택의 참고 자료가 될 수 있지만 자신의 작업에 대한 수용 시험을 대신하지 않는다. 자료 탐색기에서는 우리의 데이터와 브라우저 및 실제 사용 행동을 확인한다. 테스트 결과를 보고할 때는 통과 개수뿐 아니라 검사하지 않은 범위도 함께 적는다.

예제 기반 시험과 속성 기반 사고를 결합한다

예제 기반 시험은 입력과 기대 출력을 구체적으로 지정한다. 속성 기반 사고는 여러 입력에서 유지되어야 할 성질을 정한다. 정렬 함수에서는 결과가 정렬되어 있다는 성질과 원소가 빠지지 않았다는 성질 및 원본이 바뀌지 않았다는 성질을 함께 생각할 수 있다. 한 성질만으로 함수 전체를 정의하기는 어렵다.

즐거찾기 토글 함수는 같은 ID를 두 번 적용하면 원래 집합으로 돌아와야 한다. 이 속성은 다양한 초기 집합에서 확인할 수 있다. 다만 존재하지 않는 자료 ID를 넣어도 되는지 같은 도메인 정책은 별도로 정해야 한다. 수학적으로 일관된 함수가 업무 요구에도 맞는지는 다른 질문이다.

변이 테스트의 사고방식은 테스트가 오류를 실제로 잡는지 묻는다. 조건식의 AND를 OR로 바꾸거나 경계의 크거나 같음을 큼으로 바꾼 가상의 결합을 생각해 본다. 기존 테스트가 여전히 통과한다면 그 결합을 구분할 입력이 빠졌을 수 있다. 이 책에서는 자동 변이 도구를 실행했다는 주장을 하지 않으며 테스트 설계의 질문으로 이 방법을 사용한다.

실패의 안정성을 확인한다

Flaky test는 코드가 바뀌지 않았는데도 통과와 실패가 번갈아 나타나는 테스트다. 비동기 타이밍과 공유 상태 및 외부 서비스에 대한 의존이 원인이 될 수 있다. 재실행해서 통과했다는 이유만으로 무시하면 검사 신호에 대한 신뢰가 낮아진다. 실패 입력과 실행 환경을 기록하고 비결정적인 요소를 줄인다.

시간에 의존하는 함수는 현재 시각을 내부에서 직접 읽는 대신 입력으로 받을 수 있다. 그러면 같은 기준 시각으로 마감 여부를 반복 검사할 수 있다. 외부 API도 계약을 확인하는 통합 시험과 고정 응답을 사용하는 계산 시험을 분리할 수 있다. 가짜 응답 시험이 실제 서비스 연결을 검증한 것은 아니라는 한계는 함께 남긴다.

회귀 테스트는 발견한 실패가 다시 나타나지 않도록 기준을 남기는 작업이다. 코드의 현재 형태를 그대로 복제하는 테스트보다 사용자가 관찰한 실패를 재현하는 테스트가 구조 변경에 더 잘 견딜 수 있다. 구현이 바뀌어도 요구가 유지된다면 같은 수용 기준으로 확인할 수 있어야 한다.

같은 오해로 코드와 테스트를 만들 수 있다

테스트 오라클은 관찰한 결과가 맞는지 판정하는 기준을 뜻한다. 코드와 테스트를 같은 설명에서 생성하면 두 결과가 같은 오해를 공유할 수 있다. 예를 들어 과목과 검색어를 모두 만족해야 한다는 요구를 둘 중 하나만 만족해도 된다고 잘못 해석했다면 구현과 기대값이 함께 틀릴 수 있다. 테스트가 통과해도 요구를 충족하지 않는 상황이다.

이를 줄이려면 기대 결과를 구현의 계산식과 다른 방식으로 검토한다. 작은 자료 집합을 사람이 분류하고 각 항목이 포함되어야 하는 이유를 적을 수 있다. 과목만 맞는 항목과 검색어만 맞는 항목 및 둘 다 맞는 항목을 구분한다. 테스트의 기대값을 구현 함수로 다시 계산하면 독립성이 약해진다.

독립성은 반드시 다른 사람이 모든 테스트를 작성해야 한다는 뜻은 아니다. 요구를 다른 표현으로 검토하거나 별도의 간단한 기준 구현과 비교하는 방법도 있다. 다만 복잡한 구현을 그대로 복사한 기준 구현은 같은 오류를 반복할 수 있다. 어떤 가정을 공유하는지 인식하는 것이 중요하다.

반례를 줄이면 원인에 가까워진다

오류를 발견한 입력이 크면 관련 없는 요소가 많아 원인을 찾기 어렵다. 실패를 유지하면서 자료를 줄이면 어떤 조건이 문제를 만드는지 볼 수 있다. 자료 100개에서 검색이 실패한다면 절반을 제거해도 실패하는지 확인하고 필요한 항목만 남긴다. 최종적으로 빈 제목을 가진 한 항목에서 오류가 발생한다면 수정의 초점이 분명해진다.

이 과정에서는 실패의 정의를 유지해야 한다. 화면이 느린 문제를 조사하다가 자료를 줄인 뒤 다른 예외가 발생했는데 같은 실패라고 취급하면 원인을 잘못 찾는다. 오류 메시지와 재현 행동 및 기대 결과를 함께 기록한다. 작은 입력은 원래 문제의 원인을 설명할 수 있을 때 가치가 있다.

AI에게는 전체 로그를 무작정 전달하기보다 재현 가능한 최소 입력과 관찰된 실패를 주는 편이 분석을 검토하기 쉽다. AI가 제안한 원인이 맞다면 어떤 입력 변화에서 오류가 사라지거나 유지되어야 하는지 예측하게 한다. 그 예측을 실행으로 확인하면 설명을 반증 가능한 가설로 바꿀 수 있다.

원인과 우연한 동시 발생을 구분한다

코드 한 줄을 바꾼 뒤 문제가 사라졌다고 그 줄이 유일한 원인이었다고 단정할 수는 없다. 브라우저 캐시가 갱신되었거나 자료가 달라졌거나 실행 순서가 바뀌었을 수 있다. 가능하면 변경 전 조건에서 실패를 재현하고 변경 후 같은 조건에서 성공을 확인한다. 이 차이를 남겨야 회귀 검사로 사용할 수 있다.

실패를 재현하지 못했다면 그 사실을 기록한다. 추정된 원인과 확인한 원인을 구분하는 것은 소극적인 태도가 아니라 진단의 정확성을 높이는 방법이다. 로그를 추가해 다음 발생 시 필요한 정보를 얻거나 오류가 일어나는 조건을 더 좁힐 수 있다. 재현되지 않는 문제에 여러 변경을 한꺼번에 넣으면 무엇이 효과가 있었는지 알기 어려워진다.

테스트의 수보다 탐지하는 오류의 종류를 본다. 정상 입력 열 개가 같은 분기를 지나면 경계 입력 한 개보다 새로운 정보를 적게 줄 수 있다. 빈 입력과 중복 및 잘못된 자료형과 순서 변화가 어떤 분기를 통과하는지 살핀다. 코드 커버리지가 높아도 요구의 중요한 반례가 빠져 있으면 품질의 충분한 근거가 되지 않는다.

코드를 실행한 것과 오류를 발견할 수 있는 것은 다르다

코드 커버리지는 어느 부분이 실행되었는지 보여 준다. 그러나 실행된 줄에서 잘못된 계산을 했을 때 테스트가 실패하는지는 별도의 질문이다. 결과를 관찰하지 않거나 기대값이 너무 약하면 모든 줄을 실행해도 오류를 놓칠 수 있다. 테스트의 품질은 실행 범위와 판정의 강도를 함께 가진다.

변이 테스트는 코드에 작은 변형을 넣고 기존 테스트가 이를 발견하는지 살피는 방법이다. 조건의 AND를 OR로 바꾸거나 경계 비교의 등호를 바꾸는 식의 변형이 예가 된다. Stryker의 지표는 발견된 변이와 살아남은 변이 및 실행되지 않은 변이 등을 구분한다. 도구의 점수에는 시간 초과와 컴파일 오류 같은 상태를 어떻게 처리하는지도 영향을 준다. [T33]

검색 함수의 두 조건을 OR로 바꾸었는데 모든 테스트가 통과한다면 두 조건의 결합을 구별하는 입력이 부족할 수 있다. 과목만 맞는 항목과 검색어만 맞는 항목을 추가하면 그

오류를 발견할 수 있다. 변이 점수를 높이는 행동이 실제 요구의 검증을 강화하는지 연결해야 한다.

살아남은 모든 변이가 반드시 문제인 것은 아니다. 허용된 입력 범위에서 원래 코드와 같은 행동을 하는 변이가 있을 수 있다. 점수 하나를 기계적으로 목표로 삼으면 의미 없는 시험을 늘릴 수 있다. 변이가 바꾼 행동과 현재 명세를 읽고 필요한 검사를 결정한다.

변환 전후의 관계로 결과를 확인할 수 있다

정확한 기대값을 매번 사람이 작성하기 어려울 때 입력을 바꿨을 때 결과 사이에 유지되어야 하는 관계를 사용할 수 있다. 이를 변형 관계를 이용한 시험으로 생각할 수 있다. 예를 들어 검색 결과에 포함되지 않는 항목을 원본에 추가해도 기존 결과의 ID 집합은 같아야 한다는 규칙을 세울 수 있다.

입력 순서를 바꾸면 결과의 순서도 달라질 수 있으므로 어떤 부분을 비교할지 명확히 한다. 명세가 원본 순서를 보존한다고 했다면 배열 전체가 같은 것을 기대하면 안 된다. ID 집합만 같아야 하는지와 정렬 결과까지 같아야 하는지를 구분한다. 관계 자체가 잘못되면 올바른 프로그램을 오류로 판정할 수 있다.

즐거찾기를 두 번 토글하면 원래 상태로 돌아와야 한다는 관계도 작은 시험이 된다. 다만 중간에 다른 변경이 없는 같은 상태 모형이라는 전제가 필요하다. 이런 전제를 밝히면 시험은 구현을 그대로 복제하지 않으면서도 중요한 규칙을 확인하는 도구가 된다.

비결정적 시험은 원인을 분리해 안정화한다

현재 시각과 난수 및 네트워크 상태가 결과에 영향을 주면 같은 코드가 다른 결과를 낼 수 있다. 필요한 경우 시각과 난수 발생기를 외부에서 전달해 시험에서 고정할 수 있다. 외부 의존성을 바꾸는 방법이 코드에 명시되어 있으면 실패를 재현하기 쉽다.

모든 외부 시스템을 가짜로 바꾸면 빠르고 안정적인 시험을 만들 수 있지만 실제 연결의 오류를 놓칠 수 있다. 가짜 응답이 실제 API의 계약과 일치하는지도 유지해야 한다. 계산 규칙은 작은 단위에서 확인하고 실제 통신과 인증 및 저장은 해당 경계의 시험으로 보완한다. 시험의 안정성과 현실성을 한 종류의 시험에서 모두 해결하려 하지 않는다.

상태 변화 / 경계값은 규칙의 빈틈을 드러낸다

경계값은 규칙의 빈틈을 드러낸다

1 / 4 빈 목록의 정책을 정한다

입력

자료 0개

예상 페이지

0개

실제 결과

0개

판정

통과

페이지 크기 10인 가상 계산 사례다. 구현의 출력에서 기대값을 복사하면 이런 오류를 탐지할 수 없다.



경계값은 규칙의 빈틈을 드러낸다

4 / 4 관련 경계도 함께 다시 검사한다

입력

자료 11개

예상 페이지

2개

실제 결과

2개

판정

수정 후 통과

페이지 크기 10인 가상 계산 사례다. 구현의 출력에서 기대값을 복사하면 이런 오류를 탐지할 수 없다.



페이지 크기 10인 가상 계산 사례다. 구현의 출력에서 기대값을 복사하면 이런 오류를 탐지할 수 없다.

1. 빈 목록의 정책을 정한다
2. 페이지 크기와 같은 수를 확인한다
3. 경계를 하나 넘는 입력으로 오류를 찾는다
4. 관련 경계도 함께 다시 검사한다

사례 | 초기화 뒤에도 남아 있는 필터

자료 탐색기에서 검색 초기화 버튼을 눌렀다. 입력칸은 비워졌지만 결과는 여전히 좁혀져 있었다. 화면의 값만 지우고 목록을 계산하는 상태는 갱신하지 않은 것이다. 빈 입력의 기대 결과가 검사 표에 있었다면 바로 찾을 수 있었다.

순서	행동	기대 결과
1	검색어 입력	관련 자료만 표시
2	과목 선택	두 조건의 교집합
3	조건 초기화	전체 자료 표시
4	즐거찾기 확인	검색 초기화로 즐겨찾기는 유지

수정 전후의 차이를 읽는다

수정한 파일과 줄을 확인한다. 단순한 초기화 오류를 고치면서 인증 파일과 배포 설정까지 바뀌었다면 이유를 묻는다. 필요 없는 넓은 수정은 검토와 복구를 어렵게 한다. 자동 포맷과 기능 변경도 나누어 읽는다.

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

명령은 현재 저장소에서 실행한다. diff는 선택 전 변경을 확인하고 staged diff는 다음 커밋에 선택한 변경을 확인하는 데 쓴다. 프로젝트가 Git 저장소가 아니라면 먼저 상태를 확인한다.

적용과 확인 | 실패가 있는 검사 기록

1. 기대 결과가 있는 행동 하나를 정한다.
2. 오류가 나는 상태를 재현한다.
3. 같은 자료와 환경을 유지한 채 한 가지 수정을 적용한다.
4. 실패하던 검사를 다시 한다.
5. 관련된 기존 행동 하나도 확인한다.

항목	기록 예
재현	과학 선택 후 초기화
기대	전체 여섯 개

항목	기록 예
수정 전	과학 한 개만 보임
변경	검색 상태와 과목 상태를 함께 초기화
수정 후	여섯 개와 전체 선택 확인
추가 확인	즐거찾기 상태 유지

분야별 실패 예

행사 신청은 더블클릭으로 중복 접수가 되는지 본다. 타이머는 일시정지 후 다시 시작할 때 남은 시간이 맞는지 본다. 인쇄 자료는 화면에 있던 표가 PDF에서 잘리지 않는지 본다. 결과를 남길 때 실행한 검사와 아직 못 한 검사를 구분한다.

검토 질문 AI가 오류를 찾았다고 말하면 수정이 검증됐을까? **해설** 수정 제안과 실제 재실행의 결과를 나누어 확인한다. 어떤 입력에서 어떤 결과가 나왔는지가 필요하다.

기술 해설 / 검사 자료는 경계에서 힘을 발휘한다

평범한 값 한 개만 확인하면 비교 기호와 빈 값 처리의 오류를 놓치기 쉽다. 범위의 바로 아래와 경계 및 바로 위를 정한다. 상태가 없는 상황과 중복 및 누락도 포함한다.

한도 20명에는 19명과 20명 및 21명을 비교한다. 예약에서는 끝 시각과 다음 시작이 같은 경우를 별도로 확인한다.

정원 20 / 현재 19 / 요청 1 -> 허용
정원 20 / 현재 19 / 요청 2 -> 거절

기술 해설 / 자동 검사는 기대와 실재를 비교한다

검사는 실행 결과를 미리 정한 기대와 대조해야 한다. 구현과 똑같은 계산식을 검사에 다시 쓰면 같은 실수를 반복할 수 있다. 작은 예시의 정답을 손으로 정하고 실제 함수를 실행해 비교한다.

분수 1/2와 2/3의 비교 정답을 먼저 적는다. 구현이 바뀌어도 정답의 의미가 유지되는지 확인한다.

```
const assert = require('node:assert/strict');
assert.deepEqual(actual, expected);
```

경계값과 날짜 및 정밀도

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 *x*가 입력을 받고 *return*이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L10 / 가중치를 적용한 연습 평균

값과 비중을 구분하고 가중치 합이 0인 경우를 처리한다. 이 사례의 입력 계약은 *items*는 *value*와 *weight* / *weight*는 0 이상다.

```
if (x.items.some(i => !Number.isFinite(i.value) ||
    !Number.isFinite(i.weight) || i.weight < 0)) throw Error('값'
);
const weight = x.items.reduce((s, i) => s + i.weight, 0);
return weight === 0 ? null :
    x.items.reduce((s, i) => s + i.value * i.weight, 0) / weight;
```

60에 비중 1을 곱하고 90에 비중 2를 곱해 합하면 240이다. 비중의 합 3으로 나누어 80을 얻는다. 비중이 모두 0이면 계산할 평균이 없으므로 *null*을 돌려주며 실제 점수 0과 구분한다.

입력	기대 결과
<code>{"items": [{ "value": 60, "weight": 1}, { "value": 90, "weight": 2}]}</code>	80
<code>{"items": [{ "value": 10, "weight": 0}]}</code>	<i>null</i>

해석의 경계. 가중치가 모두 0일 때 0점처럼 표시하지 않는다. 실제 평가에 사용하기 전 별도의 정책과 검토 기준을 정한다.

분석 사례 L15 / 마감일 지난 요청 찾기

완료되지 않았고 정한 기준일보다 이전인 요청을 고른다. 이 사례의 입력 계약은 *today*와 *due*는 YYYY-MM-DD / *done*은 완료 여부다.

```
return x.items.filter(i => !i.done && i.due < x.today).map(i => i
.id);
```

기준일 24일과 마감 23일을 비교하면 이미 지난 요청이다. 24일이 마감인 요청은 이 규칙에서 아직 지난 것으로 보지 않는다. 완료한 요청은 같은 날짜 조건을 만족해도 제외한다.

입력	기대 결과
<code>{"today": "2026-10-24", "items": [{"id": "a", "due": "2026-10-23", "done": false}, {"id": "b", "due": "2026-10-24", "done": false}]}</code>	<code>["a"]</code>
<code>{"today": "2026-10-24", "items": [{"id": "a", "due": "2026-10-23", "done": true}]}</code>	<code>[]</code>

해석의 경계. 예제는 유효하고 0을 채운 ISO 날짜만 비교한다. 시간과 시간대가 들어가면 어떤 기준으로 비교할지 다시 설계한다.

분석 사례 L16 / 예약 시간 구간의 겹침

끝 시각과 다음 시작이 같은 경우는 허용하는 구간을 계산한다. 이 사례의 입력 계약은 start와 end는 자정 이후 분 / start는 end보다 작음다.

```
const valid = i => Number.isInteger(i.start) && Number.isInteger(i.end) &&
  i.start >= 0 && i.end <= 1440 && i.start < i.end;
if (!valid(x.a) || !valid(x.b)) throw Error('구간');
return x.a.start < x.b.end && x.b.start < x.a.end;
```

14시부터 15시까지와 14시 30분부터 15시 30분까지는 일부 시간이 겹친다. 반면 다음 예약이 정확히 15시에 시작하면 이 예제의 반열린 구간 정책에서는 겹치지 않는다.

입력	기대 결과
<code>{"a": {"start": 840, "end": 900}, "b": {"start": 870, "end": 930}}</code>	<code>true</code>
<code>{"a": {"start": 840, "end": 900}, "b": {"start": 900, "end": 960}}</code>	<code>false</code>

해석의 경계. 양끝을 모두 포함하는지의 정의에 따라 경계 결과가 달라진다. 겹침 계산과 실제 동시 예약 저장의 원자적 처리를 구분한다.

분석 사례 L18 / 남은 날짜 안내

날짜만 있는 값으로 UTC 기준 일수 차이를 연습한다. 이 사례의 입력 계약은 today와 target은 유효한 YYYY-MM-DD다.

```
const day = d => Date.parse(d + 'T00:00:00Z');
const diff = (day(x.target) - day(x.today)) / 86400000;
if (!Number.isFinite(diff)) throw Error('날짜');
return diff;
```

두 날짜를 UTC의 자정으로 바꾸어 밀리초 차이를 일수로 나눈다. 24일과 27일은 3일 차이다. 지역의 마감 시각과 서머타임까지 처리하는 일반 일정 엔진은 아니라는 점을 남긴다.

입력	기대 결과
<code>{"today": "2026-10-24", "target": "2026-10-27"}</code>	3
<code>{"today": "2026-10-24", "target": "2026-10-23"}</code>	-1

해석의 경계. 이 모형은 시각과 현지 시간대의 마감 정책까지 구현하지 않는다. 현지 날짜 기준과 정확한 마감 시각 기준을 구분해 안내한다.

분석 사례 L20 / 중복 신청의 식별 기준

가짜 주소의 공백과 영문 대소문자를 정리해 중복을 찾는다. 이 사례의 입력 계약은 emails는 가짜 주소 배열 / 이 예제의 동일성 정책을 적용다.

```
const seen = new Set();
return x.emails.map(email => {
  const key = email.trim().toLowerCase();
  const duplicate = seen.has(key); seen.add(key);
  return {key, duplicate};
});
```

앞뒤 공백을 지우고 소문자로 바꾼 주소를 비교 키로 사용한다. 첫 주소는 새 항목이고 두 번째 같은 키는 중복이다. 어떤 주소를 동일하게 볼지의 정책과 실제 서버의 중복 저장 방식은 따로 확인한다.

입력	기대 결과
<pre>{"emails": ["A@example.invalid", " a@example.invalid "]}</pre>	<pre>[{"key": "a@example.invalid", "duplicate": false}, {"key": "a@example.invalid", "duplicate": true}]</pre>

해석의 경계. 이 정규화 정책을 모든 외부 식별자에 임의로 적용하지 않는다. 서버에서 같은 요청을 동시에 받을 때의 중복 방지도 검토한다.

관련 근거 [\[R06\]](#) [\[R14\]](#)

13장 구조 개선과 성능을 함께 판단하기

좋은 구조는 예상되는 변경을 국소화한다

아키텍처는 대규모 시스템의 도형만을 뜻하지 않는다. 작은 자료 탐색기에서도 데이터의 권위와 상태의 소유자 및 외부 의존성의 위치를 정하는 선택이 구조를 만든다. 어떤 변경이 어느 부분에 영향을 줄지 설명할 수 있다면 구조를 읽고 있는 것이다.

검색 대상을 제목에서 설명까지 넓히는 변경을 생각하자. 계산 함수와 입력 안내 및 테스트가 바뀌는 것은 자연스럽다. 이 변경 때문에 저장소 인증과 배포 설정까지 함께 바뀐다면 의존성이 불필요하게 연결되어 있을 수 있다. 변경의 목적과 영향 범위를 비교하면 개선할 경계를 찾을 수 있다.

응집도는 한 구성 요소 안의 책임이 얼마나 관련되어 있는지 생각하는 관점이다. 결합도는 구성 요소들이 서로 얼마나 의존하는지 생각하는 관점이다. 수치를 외우기보다 함께 바뀌는 이유가 같은지 묻는다. 검색과 정렬은 같은 목록 계산에 속할 수 있지만 인증 토큰 갱신은 다른 이유로 바뀐다.

의존성의 방향을 바꾸면 테스트가 쉬워질 수 있다

필터 함수가 내부에서 localStorage를 읽으면 테스트할 때 브라우저 저장소를 준비해야 한다. 저장소에서 읽은 값을 함수의 인자로 전달하면 계산을 독립적으로 확인할 수 있다. 외부 기능을 직접 찾아 쓰는 대신 필요한 값이나 함수를 전달받는 방식은 테스트 경계를 단순하게 만들 수 있다.

```
function selectedItems(items, query, subject) {
  const q = query.trim().toLocaleLowerCase('ko-KR');
  return items.filter(item => {
    const subjectMatches = subject === '전체'
      || item.subject === subject;
    const titleMatches = item.title
      .toLocaleLowerCase('ko-KR').includes(q);
    return subjectMatches && titleMatches;
  });
}
```

이 함수는 유효한 자료 배열과 문자열 입력을 전제로 한다. 저장소와 DOM 및 네트워크를 읽지 않는다. 따라서 실패가 나면 계산 규칙과 입력을 먼저 조사할 수 있다. 실제 화면에서는 상태를 읽어 이 함수를 호출하고 결과를 렌더링하는 연결 코드가 추가된다. 분리된 함수가 정확하다고 연결 코드까지 정확한 것은 아니다.

의존성 주입이라는 이름을 배우기 전에 이 간단한 관계를 이해하면 충분하다. 외부 기능을 인자로 받는 모든 함수가 좋은 설계는 아니다. 매개변수가 지나치게 많아지거나 작은 기능의 호출이 복잡해지면 다른 경계를 검토해야 한다. 패턴은 문제를 해결하기 위한 언어이며 목적 자체가 아니다.

리팩터링 전에 행동의 기준선을 만든다

기존 코드가 복잡하다고 바로 다시 작성하면 정상 동작까지 바뀔 수 있다. 먼저 현재 사용자가 의존하는 행동을 기록한다. 검색과 과목의 결합 및 초기화와 즐겨찾기 복원 같은 흐름을 확인한다. 알려진 오류는 정상 기준과 구분해 적는다. 잘못된 동작을 무조건 보존하는 것이 리팩터링의 목적은 아니다.

구조 변경과 버그 수정을 함께 해야 하는 경우에는 변경 단계를 나눈다. 먼저 테스트로 실패를 재현하고 작은 수정으로 기대 행동을 만든 뒤 구조를 개선할 수 있다. 반대로 현재 구조 때문에 테스트가 불가능하다면 외부 경계를 최소한으로 분리해 관찰 지점을 만든다. 각 단계에서 어떤 행동이 바뀌었는지 설명할 수 있어야 한다.

SI가 대규모 리팩터링을 제안하면 변경 이유와 위험 및 검증 방법을 먼저 읽는다. 파일 이동과 이름 변경이 대부분인 차이 속에 조건식 변경이 숨어 있을 수 있다. 의미 있는 변경과 기계적인 변경을 분리하면 리뷰의 부담이 줄어든다. 수정 범위가 커질수록 되돌릴 기준도 더 중요해진다.

성능 최적화에는 비용을 줄이는 효과가 있다

캐시를 추가하면 반복 계산은 줄지만 무효화 규칙과 메모리 사용이 늘어난다. 모든 자료를 한 번에 내려받으면 이후 필터는 빠르지만 초기 전송량이 커진다. 서버 검색을 사용하면 클라이언트 부담은 줄 수 있지만 네트워크 지연과 요청량 제한을 고려해야 한다. 최적화는 비용을 없애기보다 다른 위치와 시점으로 이동시키는 경우가 많다.

자료 탐색기에서 제목을 소문자로 바꾸는 작업을 매 검색마다 수행한다고 하자. 정규화한 검색용 값을 미리 만들면 반복 비용을 줄일 수 있다. 그러나 원본 제목이 바뀌면 검색용 값도 갱신해야 한다. 원본과 파생 값의 불일치 위험을 성능 이익과 함께 평가한다.

프로파일링은 실제 시간이 어디에 쓰이는지 관찰하는 과정이다. 전체 실행 시간을 재는 것뿐 아니라 계산과 렌더링 및 네트워크를 구분한다. 작은 입력에서 보이지 않던 병목이 큰 입력에서 나타날 수 있으므로 실제 사용 규모를 대표하는 자료를 준비한다. 모든 최적화에는 개선 전후의 같은 측정 조건이 필요하다.

검색 기능을 확장할 때 정확성과 비용을 함께 본다

문자열 포함 검색은 규칙이 단순하고 설명하기 쉽다. 형태소 분석이나 의미 검색을 추가하면 더 다양한 표현을 찾을 수 있지만 결과의 설명과 평가가 어려워질 수 있다. 사용자가 정확한 제목을 찾는지와 관련 주제를 탐색하는지에 따라 적합한 방식이 다르다.

의미 검색의 벡터는 문서 내용을 수치 표현으로 바꾸어 유사도를 계산하는 데 사용된다. 유사도가 높다는 사실이 업무상 정답이라는 뜻은 아니다. 서로 반대되는 문장도 많은 단어와 맥락을 공유할 수 있다. 결과를 평가하려면 사용자의 질의와 관련성 판정 기준 및 실제 자료를 포함한 시험 집합이 필요하다.

검색 결과를 생성형 모델로 요약하는 기능은 또 다른 오류 경로를 추가한다. 검색이 필요한 문서를 놓치거나 요약이 문서에 없는 내용을 만들 수 있다. 검색 단계의 품질과 생성 단계의 품질을 나누어 확인한다. 출처 링크를 제공하더라도 그 출처가 실제 문장을 뒷받침하는지 검토해야 한다.

기술 선택은 되돌릴 비용까지 포함한다

새 라이브러리를 도입하면 구현 시간이 줄 수 있지만 프로젝트의 의존성과 갱신 비용이 늘어난다. 작은 기능을 위해 큰 도구를 넣을 때는 직접 구현의 비용과 비교한다. 직접 구현 역시 테스트와 유지 책임이 생기므로 무조건 더 단순하다고 말할 수는 없다.

외부 서비스에 데이터를 저장하면 편리한 기능을 얻지만 자료를 내보내고 다른 환경으로 옮길 경로를 확인해야 한다. 표준 형식의 내보내기와 데이터 계약을 남기면 전환 비용을 줄일 수 있다. 초기 선택을 영구적인 약속으로 만들지 않는 설계는 작은 프로젝트에서도 가치가 있다.

최종적으로 구조를 평가할 때는 새로운 요구 하나를 적용해 본다. 자료 유형을 추가하거나 저장 방식을 바꾸거나 필터를 확장하는 작업이 어디까지 영향을 주는지 관찰한다. 추상적인 좋은 코드라는 평가보다 구체적인 변경 비용이 설계의 장단점을 더 분명하게 보여 준다.

모듈의 기준은 실행 순서만이 아니다

Parnas는 1972년 논문에서 같은 문제를 서로 다른 기준으로 모듈화하는 사례를 비교했다. 핵심은 처리 단계대로 나누는 방법만이 아니라 변경될 가능성이 있는 설계 결정을 모듈 안에 숨기는 기준이었다. 이는 파일을 작게 쪼개면 자동으로 좋은 구조가 된다는 주장이 아니다. 어떤 정보를 다른 부분이 알아야 하는지 제한하는 설계의 문제다.

[S19]

자료 탐색기에서 읽기와 검색 및 표시 순서에 따라 파일을 나눌 수 있다. 그러나 검색 코드가 저장소의 내부 키와 JSON 형식을 직접 알고 있다면 저장 방식이 바뀔 때 검색도 수정해야 한다. 저장 방식의 결정을 loadFavorites와 saveFavorites라는 경계 안에 숨기면 검색은 ID 집합만 받아 계산할 수 있다. 모듈의 이익은 이름과 위치보다 변경의 파급 범위에서 나타난다.

반대로 경계를 과도하게 추상화하면 작은 변경을 이해하기 위해 여러 파일을 오가야 할 수 있다. 한 곳에서만 사용하는 짧은 계산을 일반화된 플러그인 구조로 바꾸는 것은 현재의 문제보다 큰 복잡성을 만들 수 있다. 정보 은닉은 모든 세부사항을 감추는 것이 아니라 변할 수 있는 결정을 적절한 책임 안에 두는 것이다.

의존성의 방향은 시험 가능성을 바꾼다

검색 함수가 직접 DOM을 읽고 localStorage에 접근하면 시험에 브라우저 환경이 필요해진다. 입력 자료와 상태를 인자로 받도록 바꾸면 작은 배열만으로 규칙을 검사할 수 있다. 외부 환경에 닿는 코드는 별도 경계에서 시험한다. 이렇게 책임을 나누면 시험이 쉬워지는 이유를 추상적인 장점이 아니라 필요한 환경의 감소로 설명할 수 있다.

다만 순수 함수 시험이 많아졌다고 통합 시험이 불필요해지는 것은 아니다. 화면에서 읽은 문자열이 함수에 제대로 전달되는지와 반환 결과를 올바르게 표시하는지는 연결의 문제다. 함수의 정확성과 연결의 정확성을 각각 확인한다. 좋은 구조는 시험 범위를 나누기 쉽게 만들지만 전체 행동의 검증을 없애 주지는 않는다.

AI에게 구조 개선을 맡길 때는 파일 수를 늘리라는 요청보다 변경 시나리오를 제시한다. 저장 방식만 바꾸어도 검색 규칙을 다시 작성하지 않게 하거나 표시 순서를 바꾸어도 원본 자료가 변하지 않게 하라는 요구가 더 평가 가능하다. 수정 후 그 시나리오를 실제로 수행하면 구조의 효과를 확인할 수 있다.

기술 부채를 선택과 망각으로 구분한다

첫 버전에서 간단한 구조를 선택하는 것은 합리적일 수 있다. 문제는 제약을 알고 선택한 것인지 제약을 모른 채 남겨 둔 것인지다. 서버 없이 로컬 저장을 쓰는 선택은 개인 사용이라는 전제에서는 적절할 수 있다. 여러 기기에서 공유해야 한다는 요구가 생겼는데 같은 구조를 유지하면 부채의 비용이 나타난다.

부채를 기록할 때는 막연히 나중에 리팩터링이라고 쓰기보다 현재의 제약과 영향을 받는 변화 및 재검토 조건을 적는다. 자료량이 늘어 검색 시간이 목표를 넘으면 색인을 검토한다는 식이다. 숫자를 정한다면 실제 측정 가능한 기준과 사용자 요구에 연결한다. 근거 없이 임의의 규모를 보편적인 한계처럼 제시하지 않는다.

모든 부채를 즉시 제거하려 하면 제품의 가치와 무관한 구조 개선에 시간을 쓸 수 있다. 실제 변경을 막거나 오류를 반복해서 만드는 부분을 우선한다. 전문적인 설계는 완벽한

추상 구조를 만드는 작업보다 현재의 제약에서 변경 비용과 실패 비용을 관리하는 작업에 가깝다. AI가 생성한 코드도 같은 기준으로 평가한다.

결합도는 다른 부분을 함께 알아야 하는 정도다

함수 하나를 수정할 때 여러 파일의 숨은 전제를 알아야 한다면 결합이 크다고 볼 수 있다. 전역 상태와 암묵적인 초기화 순서 및 공유되는 가변 객체는 이런 결합을 만들 수 있다. 파일이 나뉘어 있어도 내부 자료 구조를 서로 직접 수정한다면 책임이 분리되었다고 보기 어렵다.

자료 탐색기의 저장 모듈이 ID 목록을 반환한다는 계약을 제공하면 검색 모듈은 저장 키와 JSON 형식을 몰라도 된다. 그러나 저장 모듈의 내부 배열을 그대로 넘기고 호출자가 수정한다면 소유권이 흐려진다. 인터페이스를 줄이는 것뿐 아니라 값의 변경 가능성과 책임을 정하는 것도 중요하다.

응집도는 한 부분 안의 요소들이 같은 목적과 변경 이유로 묶여 있는지를 살피는 관점이다. 이름이 비슷하다는 이유로 모든 유틸리티를 한 파일에 모으면 관계없는 책임이 섞일 수 있다. 현재의 변경 시나리오에서 함께 움직이는 규칙을 중심으로 경계를 정한다. 용어를 외우기보다 예상 변경을 실제로 수행해 보는 편이 구조를 평가하기 쉽다.

복잡도 표기는 실제 병목을 찾는 출발점이다

자료 n 개를 한 번 읽는 계산은 입력 크기에 대체로 선형으로 증가하는 작업으로 볼 수 있다. 하지만 문자열 비교의 비용과 DOM 생성 및 메모리 할당 같은 요소도 실제 시간에 영향을 준다. $O(n)$ 이라는 표기만으로 두 구현의 속도가 같다고 말할 수 없다. 성장률과 상수 비용 및 실행 환경을 구분한다.

즐거찾기 ID m 개를 각 자료마다 배열에서 찾으면 대략 n 과 m 의 곱에 비례하는 비교가 필요할 수 있다. ID 집합을 만들어 membership 검사를 수행하면 반복 비교를 줄이는 선택이 된다. 그러나 자료가 아주 작다면 구조를 만드는 비용이 더 눈에 될 수 있다. 측정된 문제와 예상 규모에 맞춰 최적화를 선택한다.

성능 개선 전후에는 결과의 의미가 유지되는지도 시험한다. 빠른 정렬이 원본 배열을 바꾸어 다른 화면을 망가뜨리면 올바른 최적화라고 할 수 없다. 정확성과 성능은 각각의 검증 질문이며 어느 하나의 개선이 다른 하나를 자동으로 보장하지 않는다.

추상화에는 되돌릴 수 있는 시점을 둔다

새 요구가 들어올 때마다 플러그인과 팩토리 및 설정 계층을 늘리면 작은 프로그램의 이해 비용이 커질 수 있다. 여러 구현이 실제로 같은 계약을 공유하는지 확인한 뒤 공통 경계를 도입한다. 현재 존재하지 않는 모든 미래 요구를 미리 수용하려는 설계는 검증할 수 없는 가정을 늘린다.

실험적인 구조를 도입했다면 어떤 변경에서 이익이 있었는지와 어떤 비용이 추가되었는지 기록한다. 원래보다 파일 탐색과 테스트 설정이 어려워졌다면 단순한 구조로 돌아갈 수 있다. 구조 개선의 가치는 패턴 이름이 아니라 이후의 작업에서 관찰되는 변화로 판단한다.

14장 접근성과 보안을 제품 품질로 통합하기

접근성은 사용자의 입력과 인지 조건을 넓히는 설계다

검색창의 placeholder는 입력 예시를 보여 줄 수 있지만 지속적으로 읽을 수 있는 라벨을 완전히 대신하지 않는다. 버튼의 이름은 수행할 행동을 전달해야 한다. 즐겨찾기 아이콘만 있으면 스크린 리더 사용자가 목적과 현재 상태를 알기 어렵다. 의미 있는 이름과 놀림 상태를 제공하고 변화가 필요한 범위에서 전달되도록 한다. [R13]

키보드 초점은 현재 조작 위치다. 목록을 다시 그릴 때 모든 요소를 교체하면 사용자가 누르던 버튼이 사라져 초점이 엉뚱한 곳으로 이동할 수 있다. 자료 탐색기의 즐겨찾기만 보기에서 항목을 해제하는 경우가 좋은 시험 장면이다. 사라진 항목 뒤에 남은 자료나 필터로 초점을 옮기는 정책을 정해야 한다. 이는 단순한 기술 오류가 아니라 사용자의 작업 연속성 문제다.

색만으로 상태를 전달하지 않는다. 성공과 실패를 초록과 빨강으로만 표시하면 색을 구별하기 어려운 사용자나 흑백 인쇄에서 의미가 사라진다. 텍스트와 형태 및 위치를 함께 사용한다. 반응형 배치는 화면을 좁혔을 때 카드가 줄바꿈되는지를 넘어 핵심 조작이 가려지지 않는지와 긴 제목이 읽히는지를 확인해야 한다.

데이터가 문서나 명령으로 해석되는 순간을 찾는다

외부 문자열을 화면에 넣을 때 `textContent`는 문자열을 텍스트로 다룬다. `innerHTML`은 문자열을 HTML로 해석할 수 있다. 자료 제목을 표시하는 데 HTML 해석이 필요하지 않다면 텍스트로 삽입하는 편이 경계를 명확히 한다. 그러나 이것만으로 앱의 모든 보안 문제가 해결되지는 않는다. 링크의 프로토콜과 파일 업로드 및 서버 질의는 각각 다른 검증이 필요하다. [R07]

같은 원리는 SQL과 셸 명령에도 적용된다. 데이터와 명령의 경계가 섞이면 입력이 원래 의도와 다른 실행 의미를 가질 수 있다. 문자열을 이어 붙이는 방식보다 해당 환경이 제공하는 매개변수화된 인터페이스를 사용한다. AI가 만든 코드를 볼 때는 입력이 어디에서 들어와 어떤 해석기를 통과하는지 추적하는 습관이 유용하다.

프롬프트 인젝션도 신뢰 경계 문제로 볼 수 있다. 에이전트가 읽는 외부 문서에 작업 지시처럼 보이는 문장이 들어 있어도 그 문서가 사용자나 운영자의 권한을 갖는 것은 아니다. 검색된 내용은 작업 대상 데이터로 취급하고 도구 실행 권한과 비밀 접근 범위를 별도로 제한해야 한다. 문장으로 무시하라고 요청하는 조치만으로 실행 권한 통제가 대체되지는 않는다.

품질 검토는 기능 완성 뒤의 장식이 아니다

초기에 접근성을 고려하면 네이티브 버튼과 입력 요소를 사용하는 것만으로도 많은 기본 동작을 얻을 수 있다. 나중에 클릭 가능한 div로 만든 화면을 수정하려면 키보드 처리와 의미 및 초점 관리를 새로 구현해야 한다. 보안 역시 같은 이유로 구조 설계에 포함해야 한다. 브라우저에 비밀을 넣는 구조를 만든 뒤 배포 직전에 숨기는 것은 근본적인 해결이 되기 어렵다.

NIST SSDF는 안전한 소프트웨어 개발을 조직 준비와 소프트웨어 보호 및 안전한 제작과 취약점 대응의 활동으로 다룬다. 이 책은 해당 프레임워크의 인증을 제공하지 않는다. 개발 전 과정에서 증거와 책임을 남겨야 한다는 관점을 작은 프로젝트에 맞게 적용한다. [S10]

HTML에 넣은 GIF도 같은 기준을 따른다. 움직임은 상태 변화를 설명할 때 유용하지만 읽는 사람에게 방해가 될 수 있다. 이 개정판은 각 GIF에 재생과 정지 및 정지 프레임 보기를 제공하고 움직임 감소 설정을 존중한다. 정보는 애니메이션을 보지 않아도 캡션과 본문으로 이해할 수 있게 한다. [T12]

생성된 프로그램과 사용자의 행동은 다른 연구 대상이다

Pearce와 동료들은 보안 취약점과 관련된 89개 시나리오에서 1689개 프로그램을 생성해 분석했다. 해당 실험에서 약 40퍼센트가 취약하다고 판정되었다. 이 수치는 선택된 시나리오와 당시 도구의 결과이며 모든 AI 코드의 취약 비율이나 현재 제품의 성능으로 일반화할 수 없다. 연구의 가치는 위험한 조건에서 생성 결과를 별도로 검증해야 한다는 문제를 구체화한 데 있다. [S20]

앞에서 다룬 Perry와 동료들의 사용자 연구는 사람이 AI 도움을 받아 작성한 결과와 안전성에 대한 인식을 관찰했다. 생성된 후보 코드를 직접 검사하는 실험과 사람이 후보를 선택하고 수정하는 실험은 질문이 다르다. 두 결과를 같은 비율로 합치거나 어느 하나로 현재 도구의 전체 위험을 계산하면 연구 설계의 차이를 무시하게 된다. [S08]

이 책이 도출하는 실무적 해석은 생성 단계와 사용 단계의 검토가 모두 필요하다는 것이다. 위험한 코드가 제안되는지 확인하는 것과 사람이 그 코드를 알아보고 고칠 수 있는지 확인하는 것은 별개의 문제다. 도구의 출력만 좋아져도 충분하지 않을 수 있고 사용자 교육만으로 모든 위험이 없어지지도 않는다. 구체적인 신뢰 경계에서 실행 결과를 확인해야 한다.

안전한 표시 방식도 입력의 의미에 따라 달라진다

자료의 제목을 화면에 표시할 때는 문자열을 HTML로 해석할 필요가 없다. 텍스트로 삽입하면 제목에 꺾쇠괄호가 들어 있어도 일반 글자로 보여 줄 수 있다. 반면 제목을 innerHTML에 넣으면 브라우저가 마크업으로 해석할 수 있다. 같은 문자열이라도 어떤 문맥에 삽입하느냐에 따라 위험이 달라진다.

외부 링크는 텍스트 표시와 다른 검증이 필요하다. 주소처럼 보이는 문자열을 그대로 href에 넣기 전에 허용할 프로토콜을 정할 수 있다. 업무 자료 링크라면 HTTPS만 허용하거나 승인한 도메인으로 제한하는 정책을 선택할 수 있다. 이때 URL 파서로 구조를 읽고 문자열 포함 여부만으로 신뢰할 도메인을 판정하지 않는다.

정제와 검증도 구분한다. 위험해 보이는 문자를 무조건 지우면 원래 자료를 훼손하거나 예상하지 못한 입력을 통과시킬 수 있다. 제목은 텍스트로 출력하고 URL은 URL 규칙으로 검증하며 SQL 값은 매개변수로 전달하는 식으로 사용 문맥에 맞는 방법을 적용한다. 하나의 만능 정제 함수로 모든 경계를 보호한다고 생각하지 않는다.

보안 확인은 변화에 따라 다시 수행한다

정적 자료 탐색기에서 안전했던 입력 처리가 서버와 파일 업로드를 추가한 뒤에도 충분하다고 보장할 수 없다. 새로운 기능은 새로운 경로와 권한을 만든다. 특히 외부 자료를 읽거나 명령을 실행하거나 파일을 저장하는 기능은 입력이 실제 행동으로 바뀌는 경계다. 어떤 경로가 추가되었는지 먼저 확인한다.

자동 분석 도구는 일정한 패턴의 문제를 빠르게 찾는 데 도움이 되지만 업무의 권한 정책을 모두 알 수는 없다. 다른 사용자의 요청을 조회해도 되는지와 완료된 요청을 다시 열 수 있는지는 도구가 코드만 보고 정하기 어렵다. 자동 검사와 사람의 정책 검토 및 실제 거부 시험을 함께 사용한다.

검토 결과는 문제 없음이라는 포괄적 문장보다 확인한 범위로 남긴다. 제목을 텍스트로 표시하는 경로를 확인했고 HTTPS 외의 링크는 거부했지만 실제 서버의 파일 업로드는 구현하지 않았다고 적는 식이다. 이렇게 기록하면 다음 변경에서 어떤 검사를 추가해야 하는지 알 수 있다. 전문적인 보고는 확신의 크기보다 근거의 범위를 정확하게 표현한다.

시각적 표시와 프로그램이 전달하는 의미를 맞춘다

즐거찾기 버튼의 색이 바뀌어도 보조 기술에 선택 상태가 전달되지 않으면 일부 사용자는 변화를 알 수 없다. 대화상자가 열렸는데 키보드 초점이 뒤의 화면에 남아 있으면 보이는 화면과 조작 위치가 달라진다. 접근성 검토는 색상과 크기만이 아니라 상태와 의미 및 조작 순서를 함께 다룬다.

자료 탐색기에서 결과가 바뀌었을 때 모든 항목을 한꺼번에 읽게 하면 오히려 방해가 될 수 있다. 결과 개수나 갱신의 핵심 정보를 적절한 시점에 전달하는 방식을 선택한다. 자동 알림의 양과 사용자의 현재 작업을 고려한다. 더 많이 알리는 것이 항상 더 이해하기 쉬운 것은 아니다.

키보드만으로 검색과 필터 및 즐겨찾기와 초기화를 수행해 본다. 초점이 어디에 있는지 보이는지와 동작 후 예상 가능한 위치로 이동하는지 확인한다. 클릭이 된다는 사실은 키보드 조작과 의미 전달까지 검증했다는 뜻이 아니다. 자동 접근성 검사와 실제 과제 수행을 함께 사용한다.

오류 메시지는 원인과 수정 경로를 연결한다

입력이 잘못되었다는 말만으로는 사용자가 무엇을 바꿔야 하는지 알기 어렵다. 어느 필드에 어떤 제약이 적용되었는지와 입력을 보존했는지를 알려 준다. 서버 오류와 사용자 입력 오류를 같은 문구로 표현하면 사용자가 해결할 수 없는 문제를 계속 수정하려 할 수 있다.

권한 오류에서는 민감한 내부 정보를 노출하지 않으면서 가능한 다음 행동을 설명한다. 다시 로그인해야 하는지와 담당자에게 요청해야 하는지 및 현재 자료를 읽을 수만 있는지를 구분한다. 보안과 사용성은 별개의 장식이 아니라 같은 실패 경로를 다른 관점에서 설계하는 작업이다.

사례 | 이름 없는 검색창

검색창의 안내가 입력칸 안에만 있었다. 사용자가 글자를 쓰자 안내가 사라졌고 무엇을 검색하는지 혼동했다. 화면의 라벨을 따로 두고 입력과 연결했다. 같은 화면에서 결과 수를 갱신하고 빈 결과도 문장으로 알리자 상태를 읽기 쉬워졌다.

```
<label for="query">자료 제목 검색</label>
<input id="query" type="search">
<p id="count" role="status" aria-live="polite"></p>
```

문자열을 보여 주는 방법

제목에 꺾쇠 기호가 들어가도 글자로 보여야 한다면 `textContent` 같은 텍스트 삽입 방식을 쓴다. HTML을 해석해야 하는 특별한 요구는 별도로 검토한다. 아무 자료나 `innerHTML`로 넣는 습관은 입력이 코드처럼 해석되는 문제로 이어질 수 있다.

교과와 업무의 다른 예

영어 학습 카드에서는 재생 버튼의 이름이 어떤 항목인지 알려야 한다. 과학 관찰 기록에서는 오차와 경고를 색뿐 아니라 글로 설명한다. 업무 대시보드에서는 작은 숫자의 색보다 상태 이름과 기준 시점을 함께 표시한다.

적용과 확인 | 기기를 바꾸어 같은 행동 확인

1. 마우스를 쓰지 않고 Tab과 Enter로 검색과 즐겨찾기를 조작한다.
2. 375px 정도의 좁은 화면에서 가로 넘침을 확인한다.
3. 글자를 확대해 버튼 이름과 결과가 잘리는지 본다.
4. 결과가 없는 검색어를 입력한다.
5. 가짜 제목에 꺾쇠 기호를 넣고 글자로 보이는지 시험한다.

발견 예	수정 방향
초점이 보이지 않음	초점 스타일과 색 대비 검토
카드 버튼의 이름이 모두 같음	항목 제목과 연결한 이름
작은 화면에서 표가 잘림	읽을 수 있는 스크롤이나 배치
저장 실패를 알 수 없음	실제 결과에 맞는 상태 안내

도구 검사 이후에도 관찰

자동 검사는 유용하지만 실제 사용 장면을 모두 대신하지 않는다. 처음 쓰는 사람에게 목표 행동만 알려 주고 어디에서 멈추는지 관찰한다. 사용자가 틀렸다고 말하기 전에 화면이 어떤 선택을 요구했는지 살핀다.

검토 질문 색 대비 검사만 통과하면 접근성 검토가 끝날까? **해설** 라벨과 키보드 및 구조와 상태 전달 등 다른 관점도 확인해야 한다. 검사 범위만큼의 결과로 설명한다.

기술 해설 / 입력 검증과 출력 처리는 다른 일이다

값의 형식과 범위를 확인하는 일과 그 값을 화면이나 저장소에 넣는 방식은 서로 다른 보호다. 형식을 통과했다고 모든 위치에서 안전하게 사용할 수 있는 것은 아니다.

제목은 길이를 검사하고 화면에는 텍스트로 넣는다. 서버에서는 권한과 필요한 자료 형식도 따로 확인한다.

```
const title = raw.trim();
if (!title) throw Error('빈 제목');
element.textContent = title;
```

의미와 조작 및 표시

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x 가 입력을 받고 `return`이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L32 / 버튼 상태를 글로 설명

현재 선택 상태에 맞는 버튼 이름과 상태값을 만든다. 이 사례의 입력 계약은 `title`은 항목 제목 / `selected`는 선택 여부다.

```
return {label: x.title + ' 즐겨찾기',
  text: x.selected ? '즐거찾기 해제' : '즐거찾기',
  pressed: String(x.selected === true)};
```

현재 선택이 참이면 버튼의 표시를 즐겨찾기 해제로 만들고 눌린 상태 문자열도 `true`로 만든다. 이름에는 자료 제목을 붙인다. 실제 키보드 조작과 초점은 화면에서 별도로 시험한다.

입력	기대 결과
<code>{"title": "분수 카드", "selected": true}</code>	<code>{"label": "분수 카드 즐겨찾기", "text": "즐거찾기 해제", "pressed": "true"}</code>

해석의 경계. 이 문자열 모형만으로 키보드 조작까지 검증한 것은 아니다. 실제 `button`에 이름과 `aria-pressed`를 적용해 조작을 시험한다.

분석 사례 L33 / 화면 너비와 카드 열 수

정한 디자인 기준에 따라 열 개수 모형을 선택한다. 이 사례의 입력 계약은 `width`는 뷰포트 너비 `px`다.

```
if (!Number.isFinite(x.width) || x.width < 0) throw Error('너비');
return x.width < 600 ? 1 : x.width < 1000 ? 2 : 3;
```

375px에서는 한 열을 고른다. 정확히 600px이면 두 열이고 1000px 이상은 세 열이다. 경계를 어디에 둘지는 이 예제의 디자인 선택이며 글자 길이와 확대 상태에 맞춰 확인한다.

입력	기대 결과
<code>{"width": 375}</code>	1
<code>{"width": 600}</code>	2
<code>{"width": 1200}</code>	3

해석의 경계. 이 숫자는 웹의 보편 규칙이 아니라 이 예제의 디자인 선택이다. 실제 CSS와 확대 상태에서 글자가 넘치지 않는지 확인한다.

분석 사례 L34 / 긴 제목을 짧게 표시

원본 제목을 유지하고 짧은 표시값을 별도로 만든다. 이 사례의 입력 계약은 text는 문자열 / limit은 1 이상 정수다.

```
if (!Number.isInteger(x.limit) || x.limit < 1) throw Error('길이')
;
const chars = [...x.text];
return chars.length <= x.limit ? x.text : chars.slice(0, x.limit)
  .join('') + '...';
```

원본 문자열을 바꾸지 않고 표시용 문자열만 만든다. 정한 개수를 넘으면 앞부분과 생략 기호를 돌려준다. 화면에서 줄인 제목을 쓰더라도 전체 제목을 읽을 수 있는 경로가 필요하다.

입력	기대 결과
<code>{"text": "가나다라마바사", "limit": 4}</code>	"가나다라..."
<code>{"text": "책", "limit": 4}</code>	"책"

해석의 경계. 원본을 잘라 저장하지 않으며 코드 포인트 단위 예제임을 이해한다. 전체 제목을 접근 가능한 방식으로 함께 제공한다.

분석 사례 L35 / 적용한 검색 조건 안내

현재 선택이 무엇인지 문장으로 설명한다. 이 사례의 입력 계약은 subject와 query는 문자열 / only는 선택 자료만 보기다.

```
const parts = [];
if (x.subject) parts.push('과목 ' + x.subject);
if (x.query.trim()) parts.push('검색 ' + x.query.trim());
if (x.only) parts.push('즐거찾기만');
return parts.length ? parts.join(' / ') : '전체 자료';
```

과목과 검색어 및 즐거찾기 조건 중 실제 적용된 것만 모은다. 아무 조건도 없으면 전체 자료라고 설명한다. 이 문장과 실제 필터의 상태가 같은 시점에 갱신되는지 살핀다.

입력	기대 결과
<code>{"subject": "과학", "query": "실험", "only": true}</code>	"과목 과학 / 검색 실험 / 즐거찾기만"
<code>{"subject": "", "query": "", "only": false}</code>	"전체 자료"

해석의 경계. 결과가 좁혀진 원인을 사용자에게 설명한다. 초기화 뒤 안내와 실제 결과가 동시에 갱신되는지 확인한다.

관련 근거 [\[R07\]](#) [\[R12\]](#) [\[R13\]](#)

15장 AI가 바꾼 코드를 이력으로 통제하기

AI가 파일을 여러 개 수정할수록 현재 작업과 검토한 결과를 구분하는 일이 중요해진다. 대화에서 이전 상태로 돌아가 달라고 요청하는 것과 정확한 커밋으로 복원하는 것은 다르다. 이 장은 자료 탐색기의 변경을 Git의 상태와 이력으로 읽는 데 집중한다.

자료 탐색기의 검색 기능과 즐겨찾기를 동시에 수정했다면 두 변화가 어떤 목적을 갖는지 먼저 분리한다. 실제 diff에서 의도하지 않은 파일이 섞였는지 확인하고 검사 가능한 단위로 커밋한다. 아래의 Git 원리는 AI 도구가 만든 변경에도 동일하게 적용된다.

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.

resource-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.

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. 기존 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.

resource-lab에서 한 사람은 자료 필드명을 subject에서 category로 바꾸고 다른 사람은 subject를 사용하는 필터를 추가했다고 하자. 파일이 다르면 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.

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.

add와 commit과 diff를 정확히 사용

커밋할 내용을 미리 고른다

Git의 작업 폴더는 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.

적용과 확인 | 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 역지로 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. 해당 Git 주제에서 복구 명령의 차이를 다시 다룬다.

관련 근거 [\[G10\]](#) [\[G65\]](#) [\[G70\]](#) [\[G72\]](#)

Fetch와 Pull과 Push와 추적 브랜치

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
git push -u origin 브랜치	첫 Push와 upstream 설정
git branch -vv	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-only가 실패하면 로컬과 원격이 갈라졌을 수 있다. 오류 문장과 log를 읽고 해당 Git 주제과 해당 Git 주제의 통합 절차를 따른다.

ahead와 behind를 보고 바로 판단하지 않기

비교 대상은 마지막으로 가져온 원격 정보다

git status -sb와 git branch -vv에 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.

적용과 확인 | 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.

관련 근거 [G10] [G11] [G68]

Branch와 Merge와 충돌 해결

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 표식 제거
검증	실행과 diff 확인	실제 동작 확인

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

위 표식 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 표식을 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.

적용과 확인 | 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.

관련 근거 [\[G06\]](#) [\[G10\]](#) [\[G69\]](#)

실수 복구의 지도

되돌리기 전에 무엇을 보존할지 정한다

작업 파일을 버릴지와 스테이징만 풀지와 커밋을 취소할지는 서로 다른 결정이다. 먼저 git status와 git diff와 git diff --staged를 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`다.

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

원하는 결과	선택할 예	작업 파일의 영향
공유한 수정의 취소 기록	git revert COMMIT-ID	반대 변경을 적용

특히 놓치기 쉬운 Restore의 기준

HEAD가 v1이며 Index가 v2이고 작업 파일이 v3이라고 하자. 이때 git restore README.md는 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/name은 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.

적용과 확인 | 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 `force`로 지우는 것이 항상 좋은가?
공유 브랜치라면 보통 `revert`로 취소 기록을 남기는 편이 협업에 유리하다. 비밀 유출은 `revert`로 비밀이 지워지지 않으므로 해당 Git 주제의 별도 대응이 필요하다.

관련 근거 [\[G14\]](#) [\[G15\]](#) [\[G16\]](#) [\[G17\]](#)

로그와 Diff와 Git의 내부 구조

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
<code>git log --graph --oneline --all</code>	이력이 어떻게 갈라졌는가
<code>git show COMMIT-ID</code>	그 커밋에서 무엇이 달라졌는가
<code>git log -- README.md</code>	이 파일을 바꾼 기록은 무엇인가
<code>git blame README.md</code>	현재 줄은 어느 커밋에서 왔는가
<code>git diff A B</code>	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 탓하는 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 B는 두 상태의 파일 차이를 물을 때 쓴다. git log A..B는 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.

적용과 확인 | 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.

관련 근거 [\[G10\]](#) [\[G21\]](#) [\[G65\]](#) [\[G66\]](#)

16장 AI와 사람의 결과를 함께 검토하는 협업

AI가 설명을 잘 작성해도 동료의 변경의 의미를 이해했다는 증거는 아니다. 리뷰는 코드와 요구 및 테스트를 함께 읽고 이해의 차이를 줄이는 과정이다. 이 장에서는 개인의 채팅 맥락을 팀이 읽을 수 있는 이슈와 PR 및 규칙으로 옮긴다.

자료 탐색기의 필터를 수정하는 PR에는 실패 입력과 기대 결과 및 실제 검사 기록을 넣는다. AI가 바꾸지 않은 영역과 아직 확인하지 않은 환경도 설명한다. 검토자는 코드가 그럴듯한지보다 요구와 증거가 일치하는지 판단한다.

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]

resource-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 다르다면 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.

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 아니다. 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.

이슈와 Projects로 일을 설계한다

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.

적용과 확인 | 수업 도구 개선 보드

1. resource-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.

관련 근거 [\[G22\]](#) [\[G23\]](#)

좋은 PR과 좋은 코드 리뷰

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

PR 항목	답을 내용
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 바뀐 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.

적용과 확인 | 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.

관련 근거 [\[G06\]](#) [\[G24\]](#)

CODEOWNERS와 협업 양식

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 설정되지 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.

적용과 확인 | 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.

관련 근거 [\[G29\]](#) [\[G24\]](#)

17장 자동 검사를 반복 가능한 피드백으로 만들기

로컬에서 한 번 통과한 검사를 지속적인 기준으로 만들려면 실행 환경과 명령 및 실패 판정을 기록해야 한다. CI는 AI가 수정할 때마다 같은 질문을 다시 확인하는 장치가 될 수 있다. 이 장의 워크플로는 설명과 검토를 위한 예시이며 실제 계정에 실행하거나 배포한 결과를 의미하지 않는다.

자료 탐색기의 제목 검색과 필터 및 입력 보존 검사를 자동화하면 이후 구조 개선에서도 같은 계약을 확인할 수 있다. 그러나 브라우저의 초점과 실제 배포 주소 및 서버 권한은 별도 검사 범위를 갖는다. 자동화의 편리함과 검증의 범위를 혼동하지 않는다.

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]

resource-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 넣은 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 건너뛸을 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.

CI의 원리와 첫 GitHub Actions

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.yaml	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

해당 Git 주제의 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.

적용과 확인 | 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.

관련 근거 [\[G37\]](#) [\[G38\]](#) [\[G39\]](#)

Actions 실패 진단과 Matrix와 재사용

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 설계가 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 갱신됐다면 오래된 실행을 보고 있지 않은지도 확인한다.

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

적용과 확인 | 두 Python 버전으로 검사

해당 Git 주제 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.

관련 근거 [\[G37\]](#) [\[G40\]](#) [\[G41\]](#)

재사용 가능한 CI를 실제로 분리

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

위치	확인할 질문
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.

적용과 확인 | 호출 Workflow 만들기

아래 파일은 기존 해당 Git 주제 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.

관련 근거 [\[G78\]](#) [\[G37\]](#)

18장 버전 관리와 배포의 재현성

코드는 실행 환경과 연결되어야 한다

커밋은 소스의 특정 상태를 식별한다. 배포는 그 상태와 빌드 환경 및 설정을 사용해 실행 가능한 결과를 제공하는 과정이다. 같은 커밋을 사용하더라도 환경 변수나 외부 데이터베이스가 다르면 동작이 달라질 수 있다. 따라서 배포를 설명할 때는 커밋 ID와 빌드 결과 및 실행 환경을 함께 기록한다.

자료 탐색기는 정적 파일로 배포할 수 있다. 반면 공용 요청판은 서버와 저장소 및 인증 설정을 포함한다. 화면 URL만 보관하면 전체 서비스를 재현하기 어렵다. 정적 배포와 서버 운영의 차이를 이해해야 첫 프로젝트에서 선택한 방법을 다음 프로젝트에 무조건 적용하지 않게 된다.

Git의 diff는 변경을 검토하는 기본 자료다. AI가 수정한 파일 이름만 확인하지 말고 실제 차이를 읽는다. 의도하지 않은 설정 변경이나 테스트 삭제가 섞였는지 살핀다. 변경이 너무 커서 설명하기 어렵다면 기능 단위로 나누어 검토한다. 작은 커밋은 그 자체가 품질 보증은 아니지만 원인을 좁히고 되돌릴 범위를 정하기 쉽게 한다. [R14]

환경의 분리는 이름보다 연결 자원의 분리다

개발과 미리보기 및 운영 환경을 다른 이름으로 표시해도 같은 데이터베이스에 쓰면 데이터 위험은 분리되지 않는다. 테스트에서 삭제한 자료가 운영에서 사라질 수 있다. 실제 분리 대상은 URL뿐 아니라 자격 증명과 저장소 및 외부 서비스와 로그다. 환경별 설정이 어디에서 주입되는지 문서화한다.

배포 플랫폼의 동작은 변할 수 있다. Vercel의 현재 문서는 새 프로젝트의 첫 배포가 Production이 된다는 조건을 설명한다. 익숙한 명령 이름만으로 미리보기라고 가정하지 말고 결과 화면의 환경과 연결 자원을 확인해야 한다. 이 책에서 제품별 안내는 2026년 10월 24일 확인한 문서를 기준으로 읽는다. [R16]

롤백은 이전 애플리케이션 판본으로 돌아가는 일이다. 이미 바뀐 데이터 구조까지 자동으로 되돌아가는 것은 아니다. 새 버전이 필드를 삭제하거나 데이터를 변환했다면 이전 코드가 새 데이터를 읽지 못할 수 있다. 배포 전에 호환성을 유지하는 기간과 데이터 마이그레이션의 복구 방식을 설계해야 한다.

재현 가능한 변경 기록을 남긴다

변경 기록에는 무엇을 바꿨는지와 왜 바꿨는지 및 어떻게 확인했는지가 필요하다. 검색 필터를 수정했다면 실패했던 입력과 수정 뒤 기대 결과를 남긴다. 사용한 모델의 전체 대화를 보관하는 것이 항상 필요한 것은 아니다. 개인정보와 비밀을 제외하면서 설계 판단과 검증 증거를 재현할 수 있는 수준으로 기록한다.

AI가 생성한 코드라는 사실은 검토 의무를 줄이지 않는다. 동시에 모든 줄의 생성 과정을 설명해야 한다는 뜻도 아니다. 운영자가 이해해야 할 핵심은 입력과 출력 및 의존성과 위험 경계다. 다른 사람이 이 정보를 읽고 실행과 수정 및 복구를 할 수 있다면 기록의 실용적 가치가 생긴다.

새 코드와 이전 코드가 함께 존재할 수 있다

배포 시점에 모든 사용자의 브라우저가 동시에 새 코드로 바뀌는 것은 아니다. 오래 열린 탭과 캐시된 자원 및 순차 배포된 서버가 함께 존재할 수 있다. API의 필드를 갑자기 삭제하면 이전 클라이언트가 실패할 수 있다. 호환성은 소스 저장소의 한 판본만 보고 판단하기 어렵다.

필드를 추가하고 새 코드가 이를 읽게 한 뒤 충분한 전환 기간을 거쳐 이전 필드를 제거하는 방식은 이런 문제를 줄이는 선택이다. 단계마다 어떤 버전 조합을 지원하는지와 이전 상태로 되돌릴 수 있는지 확인한다. 이 과정은 배포 횟수를 늘릴 수 있지만 한 번의 변경에 모든 위험을 집중시키지 않게 한다.

자료 마이그레이션도 같은 관점으로 읽는다. 새 코드가 새 형식으로만 저장하기 시작하면 이전 코드로 되돌려도 이전 자료를 읽을 수 없을 수 있다. 되돌림 계획에는 코드와 데이터 및 설정을 함께 포함한다. 버튼 하나로 이전 배포를 선택할 수 있다는 사실만으로 업무가 복구된다고 약속하지 않는다.

배포 판정은 검사한 대상과 공개한 대상을 연결한다

테스트한 소스에서 만든 파일과 실제로 올린 파일이 같은지 확인한다. 테스트 이후 다시 빌드하면서 환경과 외부 자료가 달라지면 검사 대상이 바뀔 수 있다. 검증한 산출물을 식별하고 그 산출물을 공개하는 흐름을 만들면 이런 불일치를 줄일 수 있다.

배포 후에는 최소한의 실제 사용자 경로를 확인한다. 시작 화면만 열렸다는 사실로 검색과 저장 및 권한이 모두 정상이라고 판단하지 않는다. 배포 환경에서 중요한 경로와 실패 응답을 확인하고 판본을 기록한다. 로컬 검사와 배포 후 확인은 서로 다른 환경의 증거다.

상태 변화 / 배포 판본과 실행 환경을 함께 확인한다

배포 판본과 실행 환경을 함께 확인한다

1 / 4 로컬 성공은 첫 번째 증거다

소스

C1

검사

로컬 통과

배포 환경

미배포

확인

준비

정적 앱의 게시와 서버 및 데이터베이스를 포함한 서비스 운영은 요구가 다르다. 환경 이름만으로 자원 분리를 추정하지 않는다.

배포 판본과 실행 환경을 함께 확인한다

4 / 4 운영 주소의 판본과 실제 행동을 확인한다

소스

C1

검사

동일 결과

배포 환경

운영

확인

사용 행동 확인

정적 앱의 게시와 서버 및 데이터베이스를 포함한 서비스 운영은 요구가 다르다. 환경 이름만으로 자원 분리를 추정하지 않는다.

정적 앱의 게시와 서버 및 데이터베이스를 포함한 서비스 운영은 요구가 다르다. 환경 이름만으로 자원 분리를 추정하지 않는다.

- 1. 로컬 성공은 첫 번째 증거다
- 2. 배포 경로에서 링크와 설정을 확인한다
- 3. 데이터와 비밀의 연결 대상을 확인한다
- 4. 운영 주소의 판본과 실제 행동을 확인한다

사례 | 첫 배포를 미리보기로 착각

개발자는 미리보기용 명령이라고 생각하고 새 프로젝트를 배포했다. 그러나 사용한 서비스의 첫 배포 규칙은 이후 배포 규칙과 달랐다. 현재 Vercel 문서는 새 프로젝트의 첫 배포를 Production으로 처리한다고 설명한다. 이름과 명령의 인상보다 실제 환경 표시와 연결된 자원을 확인해야 한다.

158

설정을 복사하기 전에

미리보기 화면이 운영 데이터베이스를 읽는다면 시험도 실제 자료에 영향을 줄 수 있다. 화면 주소만 분리하지 말고 사용하는 저장소와 비밀 및 외부 서비스도 구분한다. 환경 변수를 바꾼 뒤 새 배포가 필요한지는 서비스의 실행 방식을 확인한다.

배포 뒤 현상	먼저 확인할 것
전체 404	주소와 시작 파일 및 결과 폴더
그림만 없음	경로와 파일 대소문자
저장만 실패	서버 주소와 권한 및 환경 설정
옛 내용 표시	적용 커밋과 배포 시점
로그인 반복	도메인과 세션 및 리디렉션 설정

검토하는 배포와 사용자에게 공개하는 배포

둘의 목적과 접근 조건을 문서로 남긴다. 공개 링크를 전달하기 전에 실제로 접근할 대상과 자료를 확인한다. 미리보기라는 단어만으로 비공개를 보장하지 않는다.

적용과 확인 | 작은 정적 도구 공개 점검

1. 가짜 자료만 들어 있는 실행 예제를 준비한다.
2. 선택한 정적 호스팅의 공식 안내로 배포한다.
3. 기본 배포 주소에서 검색과 필터 및 즐겨찾기를 확인한다.
4. 다른 기기의 브라우저에서 같은 주소를 연다.
5. 배포한 파일과 확인한 날짜 및 주소를 기록한다.

실제 배포는 계정과 서비스 설정이 필요하다. 책의 HTML을 여는 것만으로 배포되지는 않는다. 도메인을 사용한다면 기본 배포 주소가 정상인 뒤 연결을 진행한다.

기록 항목	작성 예
판본	자료 탐색기 v0.1
자료 범위	가짜 자료 여섯 개

기록 항목	작성 예
확인	모바일 필터와 빈 결과
알려진 한계	브라우저별 즐겨찾기
복구	이전 배포 파일 보관 위치

검토 질문 Push에 성공하면 사용자가 보는 화면도 반드시 바뀔까? **해설** 배포 연결과 실행 성공 및 적용 판본을 따로 확인한다. 저장소 전달과 서비스 반영은 같은 결과가 아니다.

기술 해설 / 로컬 주소와 공개 주소를 구분한다

내 PC에서 파일을 여는 것과 로컬 서버를 실행하는 것 및 외부에 배포하는 것은 다른 단계다. 주소와 실행 과정을 함께 읽는다. 브라우저 저장과 경로 및 API의 동작도 어떤 출처에서 실행하는지 영향을 받는다.

같은 파일을 file 주소와 로컬 서버 주소에서 열어 비교한다. 저장이 이어지지 않으면 다른 저장 공간인지 확인한다.

```
python -m http.server 8000 --bind 127.0.0.1
```

기술 해설 / 변경을 비교하는 순서

먼저 파일 목록을 보고 그다음 줄의 차이를 읽는다. 요청한 기능과 관련 없는 파일이 함께 바뀌었다면 이유를 찾는다. 새 의존성과 외부 서비스 및 권한 변경은 운영의 영향도 함께 읽는다.

초기화 버튼을 고쳤는데 로그인 파일과 배포 설정이 함께 바뀌었다. 필수 변경인지와 더 작은 수정으로 해결할 수 있는지 검토한다.

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

기술 해설 / 배포 결과물과 설정을 연결한다

소스 파일과 빌드 결과 폴더 및 실제 실행 환경을 구분한다. 환경 설정을 바꾸어도 이미 만들어진 결과에 자동 반영되지 않는 경우가 있다. 서비스를 실제 주소에서 다시 확인한다.

로컬에서는 그림이 보이지만 배포에서는 없으면 파일 경로와 대소문자 및 결과 폴더를 읽는다. 도메인 연결 전에 기본 주소에서 검증한다.

소스 커밋 -> 빌드 결과 -> 배포 판본 -> 실제 주소

관련 근거 [\[R14\]](#) [\[R15\]](#) [\[R16\]](#)

19장 의존성과 실행 권한의 공급망을 읽기

AI는 직접 작성한 코드뿐 아니라 패키지와 Action 및 외부 서비스를 제안한다. 이들은 실행되는 프로그램의 일부가 되므로 출처와 버전 및 필요한 권한을 확인해야 한다. 도구가 추천했다는 사실은 신뢰의 근거가 아니라 검토를 시작할 입력이다.

자료 탐색기에 작은 기능을 추가하려고 대형 패키지를 도입한다면 직접 구현의 비용과 의존성 관리 비용을 비교한다. 공용 요청판의 배포 자동화에는 비밀과 운영 권한이 연결되므로 외부 입력과 실행 코드의 경계를 더 엄격하게 확인한다.

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.

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 증명이 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]

resource-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.

Secrets와 GITHUB_TOKEN과 안전한 실행

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

검토 질문	찾아볼 지점
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.

적용과 확인 | Workflow 권한 리뷰

실제 비밀 값을 만들 필요는 없다. 해당 Git 주제 Workflow와 조직에서 쓰는 Workflow 하나를 읽고 아래 표를 채운다.

질문	기록할 내용
누가 이벤트를 만들 수 있는가	내부 작성자 또는 외부 기여자
어떤 코드를 실행하는가	기준 브랜치 또는 PR 코드
어떤 토큰이 있는가	GITHUB_TOKEN과 외부 비밀

질문	기록할 내용
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.

관련 근거 [\[G42\]](#) [\[G43\]](#)

Environment와 OIDC와 배포 통제

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 적용과 확인 | 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.

관련 근거 [G44] [G45]

Dependabot과 Code scanning과 공급망

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.

적용과 확인 | GitHub Actions 갱신 PR 받기

아래 파일은 Action 버전 갱신 감시를 위한 완결된 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.

관련 근거 [\[G46\]](#) [\[G47\]](#) [\[G48\]](#)

서명과 출처 검증과 신뢰

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
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 적용과 확인 | 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.

관련 근거 [\[G60\]](#) [\[G61\]](#)

20장 운영 관찰과 비용 및 복구 설계

로그와 지표와 추적은 서로 다른 질문에 답한다

로그는 개별 사건을 설명한다. 지표는 사건의 수나 시간 같은 값을 집계한다. 추적은 하나의 작업이 여러 구성 요소를 통과한 경로를 연결한다. 요청판에서 저장에 실패했다면 오류 로그로 구체적인 사건을 보고 실패율 지표로 범위를 판단하며 요청 식별자로 관련 서버 작업을 연결할 수 있다.

관찰 자료는 많을수록 무조건 좋은 것이 아니다. 사용자의 입력 본문을 모두 기록하면 문제 재현에는 편리할 수 있지만 민감한 자료의 복제본이 늘어난다. 운영에 필요한 필드와 보존 기간 및 접근 권한을 정한다. 비밀 키의 일부 이름만 가리는 단순 함수는 임의의 중첩 데이터와 모든 민감값을 안전하게 제거하지 못한다.

서비스 수준 목표는 중요한 사용자 행동이 어느 정도로 성공해야 하는지 정한 운영 목표다. 작은 프로젝트라면 자료 목록을 정상적으로 열 수 있는지와 요청 저장에 완료되는지를 먼저 관찰한다. 추상적인 서버 정상 상태보다 사용자의 실제 성공을 대표하는 신호를 고른다. 목표 수치는 사업과 사용 맥락의 판단이며 이 책의 예시 수치를 그대로 채택할 이유는 없다.

비용은 단가와 호출 구조가 결합된 결과다

AI 기능을 포함한 앱에서는 요청 수와 입력 및 출력 토큰 수가 비용에 영향을 줄 수 있다. 그러나 모델 요금만 보면 저장소와 네트워크 및 재시도와 운영 시간의 비용을 놓친다. 실제 비용 모델은 고정 비용과 사용량에 비례하는 비용 및 실패로 반복되는 비용을 구분해야 한다.

자료를 검색할 때마다 같은 요약을 생성하는 구조라면 캐시를 통해 호출을 줄일 수 있다. 이때 캐시 키에는 결과에 영향을 주는 입력과 버전이 포함되어야 한다. 서로 다른 사용자의 권한 범위가 같은 키로 묶이면 다른 사람의 결과를 보여 줄 수 있다. 비용 최적화 역시 데이터 경계와 함께 검토해야 한다.

재시도는 일시적 실패를 회복시키지만 동시에 부하를 키울 수 있다. 실패할수록 모든 클라이언트가 즉시 다시 요청하면 서버가 회복하기 어려워진다. 지수 백오프와 무작위 지연은 재시도 시점을 분산하는 기법이다. 재시도 횟수와 총 대기 시간에는 상한을 두고 쓰기 작업의 중복 방지 조건을 함께 확인한다.

복구 목표는 실제 복원 시험으로 확인한다

RPO는 장애 후 허용할 수 있는 데이터 손실의 시간 범위를 뜻한다. RTO는 서비스를 복구하기까지 허용하는 시간 목표다. 하루에 한 번 백업한다는 사실만으로 RPO가 언제나 24시간 이하라고 보장되지는 않는다. 백업 실패와 복원 불가능 및 장애 시점을 함께 고려해야 한다.

백업 파일이 존재하는지 확인하는 검사는 복원 성공을 증명하지 않는다. 분리된 환경에서 파일을 읽고 데이터 개수와 중요한 관계를 확인한 뒤 실제 애플리케이션으로 조회해야 한다. 암호화된 백업이라면 복호화에 필요한 키의 접근 가능성도 복구 계획의 일부다. 백업과 키가 같은 장애로 함께 사라지면 사본의 의미가 줄어든다.

운영 문서에는 장애를 판단하는 신호와 초기 조치 및 되돌리는 방법을 적는다. 복구가 끝났다는 판단에는 화면이 열리는 것뿐 아니라 데이터가 의도한 상태인지 확인하는 절차가 필요하다. 전문성은 장애가 전혀 생기지 않는다는 약속보다 장애를 발견하고 범위를 줄이며 복원할 수 있는 준비에서 드러난다.

복잡도는 입력 증가에 대한 성장률을 설명한다

자료 N개를 한 번씩 확인하는 선형 검색은 입력 수가 늘 때 검사 횟수도 비례해 늘어난다. 제목 길이까지 고려하면 문자열 처리 비용도 포함해야 한다. $O(N)$ 이라는 표현은 정확히 몇 밀리초가 걸리는지를 말하지 않는다. 기기와 데이터 길이 및 구현 상수와 브라우저 상태가 실제 시간에 영향을 준다.

검색 결과를 얻을 때마다 전체 배열을 다시 정렬하면 필터 비용 외에 정렬 비용이 추가된다. 원본 순서가 고정되어 있다면 미리 정렬해 두는 선택을 검토할 수 있다. 그러나 자료가 자주 갱신되면 캐시를 무효화하는 규칙이 필요하다. 성능 개선은 계산을 없애기보다 어느 시점에 수행하고 언제 다시 계산할지를 바꾸는 일이기도 하다.

데이터를 모두 DOM 요소로 만들면 계산보다 화면 갱신이 병목이 될 수 있다. 페이지네이션과 가상 목록은 한 번에 표시할 요소 수를 줄이는 전략이다. 먼저 성능 기록을 보고 병목이 계산인지 네트워크인지 화면인지 구분한다. 측정 없이 복잡한 캐시를 추가하면 오류 가능성과 유지 비용만 늘어날 수 있다.

평균 하나로 사용자 경험을 설명하지 않는다

응답 시간의 평균은 긴 지연 일부를 숨길 수 있다. 중앙값과 상위 백분위 및 실패 비율을 함께 보면 경험의 분포를 더 잘 이해할 수 있다. 표본 수가 적으면 백분위도 불안정하므로 측정 기간과 요청 수를 함께 적는다. 숫자의 소수점을 늘리는 것이 측정의 신뢰성을 높이지는 않는다.

비용의 간단한 모형을 $C = F + N \times u$ 로 둘 수 있다. F 는 고정 비용이며 N 은 호출 수이고 u 는 호출당 비용이다. 실제 AI 요청은 입력과 출력 길이에 따라 비용이 달라질 수 있으므로 u 를 상수로 두는 것은 근사다. 실패와 재시도 및 캐시 적중률까지 고려하면 모델을 확장해야 한다.

이 식의 용도는 예산을 확정하는 데 있지 않다. 어떤 변수가 비용에 영향을 주는지 질문하게 하는 데 있다. 사용자가 열 명에서 백 명으로 늘 때 모든 사용자가 같은 요청을 같은 빈도로 보내는지부터 확인한다. 실제 단가와 제한은 도입 시점의 공식 요금과 사용량 기록으로 대체해야 한다.

로그와 지표는 서로 다른 질문에 답한다

로그는 특정 사건의 맥락을 남기고 지표는 여러 사건의 수치적 경향을 보여 준다. 요청 실패율이 늘었다는 지표는 문제가 있다는 신호를 줄 수 있지만 어느 입력과 경로에서 실패했는지는 추가 정보가 필요하다. 반대로 개별 오류 로그 하나만으로 전체 사용자의 영향을 추정하기 어렵다. 사건의 세부와 집계된 규모를 연결해 읽는다.

공용 요청판에서는 요청 식별자와 처리 단계 및 결과와 소요 시간을 남길 수 있다. 비밀번호와 인증 토큰 및 요청 본문의 민감한 내용을 그대로 기록할 필요는 없다. 원인 분석에 필요한 최소한의 정보를 정하고 보존 기간과 접근 권한을 관리한다. 로그가 많을수록 무조건 좋은 것이 아니라 필요한 질문에 답할 수 있어야 한다.

자료 탐색기의 작은 정적 버전에서는 서버 로그가 없을 수 있다. 브라우저 콘솔과 저장 실패 메시지 및 재현 입력이 주요 진단 자료가 된다. 사용 환경이 단순하다는 이유로 운영 개념이 없어지는 것은 아니다. 관찰 가능한 경계를 현재 구조에 맞게 선택하면 된다.

시간의 분포와 처리량을 구분한다

요청 하나의 응답 시간과 일정 시간 동안 처리한 요청 수는 다른 값이다. 동시 사용자가 늘어나면 처리량이 증가하다가 자원 한계에 도달하고 대기 시간이 급격히 길어질 수 있다. 평균 응답 시간이 낮아도 일부 사용자는 매우 오래 기다릴 수 있으므로 분포와 느린 요청의 조건을 살핀다.

가상의 측정에서 대부분의 요청이 100밀리초 안에 끝났지만 일부 요청이 수 초 걸렸다면 평균만 보고 문제를 숨길 수 있다. 백분위수는 일정 비율의 요청이 어떤 시간 이하였는지 보여 준다. 다만 표본이 적으면 높은 백분위수는 몇 개의 관찰에 크게 좌우된다. 측정 구간과 표본 수 및 부하 조건을 함께 적어야 한다.

성능 개선 전후에는 같은 자료량과 비슷한 실행 조건을 사용한다. 개발자 도구를 열었는지와 캐시가 채워졌는지 및 기기의 부하가 어떤지도 결과에 영향을 줄 수 있다.

브라우저에서 한 번 측정된 값을 모든 사용자의 속도로 표현하지 않는다. 반복 측정에서 변화의 방향이 유지되는지와 사용자가 체감할 행동이 개선되는지를 함께 본다.

장애 대응은 복구와 원인 분석을 구분한다

서비스가 중단되었을 때 첫 목표는 영향을 줄이는 것이다. 문제가 된 배포를 되돌리거나 위험한 기능을 잠시 제한할 수 있다. 근본 원인을 완전히 이해할 때까지 사용자가 계속 피해를 보게 둘 필요는 없다. 다만 임시 복구와 원인 제거를 같은 완료 상태로 기록하지 않는다.

복구 뒤에는 사건의 시간 순서를 구성한다. 어떤 변경이 있었고 언제 지표가 변했으며 어떤 대응 후 회복되었는지 기록한다. 시간적으로 가까운 변경이 원인 후보가 될 수 있지만 자동으로 원인이 되는 것은 아니다. 다른 자료와 대조하고 재현 가능한 조건을 찾는다. 사람의 실수라는 설명만으로 끝내면 같은 실수가 다시 영향을 주지 않게 할 구조적 개선을 놓친다.

후속 작업은 원인과 연결되어야 한다. 입력 검증이 빠졌다면 검증과 회귀 검사를 추가하고 배포 전 검사가 실행되지 않았다면 자동화 경로를 수정한다. 모든 장애 뒤에 문서를 더 쓰겠다고만 하면 실제 실패 경계가 바뀌지 않을 수 있다. 개선 조치가 완료되었는지 관찰할 증거도 정한다.

실패율의 분모를 먼저 정한다

실패 요청 수를 전체 요청 수로 나눈 비율은 계산하기 쉽지만 무엇을 요청으로 썼는지에 따라 의미가 달라진다. 사용자가 취소한 요청과 잘못된 입력 및 서버 오류를 같은 실패로 셀지 정해야 한다. 봇의 트래픽과 내부 상태 확인 요청이 분모를 늘리면 실제 사용자 경험이 가려질 수도 있다.

SLO는 선택한 서비스 수준 지표에 대한 목표다. 오류 예산은 그 목표가 허용하는 실패의 범위를 운영 판단에 연결하는 관점이다. Google SRE Workbook은 예산 소모 속도를 이용한 알림과 여러 시간 구간의 결합을 설명한다. 수치만 복사하기보다 자신의 사용자 행동을 대표하는 지표와 관찰 기간을 정해야 한다. [T37]

가상의 서비스가 요청 성공률 99.9퍼센트를 목표로 한다면 허용 실패 비율은 0.1퍼센트다. 관찰된 실패 비율이 1퍼센트라면 같은 정의에서 예산을 기준 속도의 10배로 소모하는 상태로 볼 수 있다. 이 계산은 예산 소모의 의미를 설명하는 예이며 모든 서비스에 99.9퍼센트가 적절하다는 권고가 아니다.

요청 수가 매우 적으면 한 번의 실패가 비율을 크게 바꾼다. 한 시간에 요청 두 개가 있고 하나가 실패한 상황과 수만 개 중 절반이 실패한 상황은 같은 50퍼센트라도 운영 판단이 다를 수 있다. 표본 수와 영향받은 사용자 및 반복 여부를 함께 본다.

분산 추적은 한 작업의 여러 구간을 연결한다

로그는 개별 사건을 기록하고 지표는 집계된 추세를 보여 준다. 분산 추적은 요청이 여러 구성 요소를 통과하는 경로를 span이라는 작업 구간들로 연결하는 데 사용된다. 각 구간의 시작과 종료 및 부모 관계와 속성을 통해 지연이 어디에서 발생했는지 조사할 수 있다. OpenTelemetry는 이런 신호를 표현하고 수집하는 공통 체계를 제공한다. [T34]

요청판의 저장 시간이 길다면 인증과 데이터 조회 및 갱신과 알림 준비를 나누어 관찰할 수 있다. 전체 시간만 알면 데이터베이스가 느린지 외부 서비스가 느린지 구분하기 어렵다. 그러나 모든 구간에 민감한 본문을 기록할 필요는 없다. 진단에 필요한 속성과 개인정보의 경계를 함께 정한다.

추적 자료를 일부만 수집하면 비용을 줄일 수 있지만 드문 실패를 놓칠 수 있다. 표본 선택 방식과 보존 기간을 알고 해석해야 한다. 관찰되지 않았다는 사실을 발생하지 않았다는 증거로 취급하지 않는다. 측정 체계의 누락도 운영 분석의 한계다.

알림은 행동할 수 있는 조건이어야 한다

오류 로그 한 개마다 알림을 보내면 작은 문제와 큰 장애가 같은 소음으로 섞일 수 있다. 현재 누가 무엇을 해야 하는지 결정할 수 있는 조건을 정한다. 일시적인 변화와 지속적인 악화를 구분하고 복구 후 알림이 종료되는 조건도 확인한다.

알림 문구에는 영향 범위와 관찰 시각 및 관련 판본과 조사 시작점을 넣는다. 원인을 확정하지 못했는데 특정 구성 요소의 고장이라고 단정하면 대응이 잘못된 방향으로 갈 수 있다. 관찰한 사실과 원인 가설을 구분한다. 장애 중에는 설명의 확신보다 다음 행동의 정확성이 중요하다.

사례 | 자동 새로고침이 만든 요청 폭증

대시보드가 짧은 간격으로 모든 자료를 다시 읽었다. 사용자가 늘자 외부 API 요청도 크게 늘었다. 개발자는 호출 간격과 실제 필요한 자료를 다시 정하고 중복 요청을 줄였다. 화면이 편리해 보인다는 이유로 호출 비용까지 적절하다고 볼 수는 없었다.

가상의 비교	계산 예
한 화면이 1분에 한 번 요청	시간당 60회
화면 20개가 같은 방식으로 동작	시간당 1200회
실패 때 재시도 세 번 추가	추가 호출 가능성 조사

위 숫자는 개념을 설명하는 가정이다. 실제 비용은 서비스의 집계 방식과 단가 및 실행 조건에 따라 계산한다. 호출 수만으로 최종 청구를 단정하지 않는다.

복원 시험을 별도 환경에서

내려받은 파일이나 데이터 사본을 실제로 다시 읽어 본다. 자료 ID와 최신 기록 및 실행 설정을 확인한다. 원래 PC의 설정이 없으면 무엇을 준비해야 하는지도 적는다. 백업 성공 메시지와 복원 성공은 서로 다른 근거다.

서비스 종료도 운영의 일부

앱을 더 쓰지 않을 때 외부 실행 자원과 도메인 및 토큰과 청구가 남는지 확인한다. 소스 저장소를 보관했다는 사실만으로 연결 자원이 모두 종료되는 것은 아니다.

적용과 확인 | 운영 카드 한 장

1. 서비스 이름과 현재 담당 역할을 적는다.
2. 사용하는 외부 자원과 청구 주체를 목록으로 만든다.
3. 실패를 확인할 로그 위치를 적는다.
4. 이전 판본과 자료 사본의 위치를 적는다.
5. 새 폴더나 별도 환경에서 복원 과정을 따라가 본다.

운영 카드	내 문서에 넣을 내용
현재 상태	적용 판본과 확인 날짜
사용 자원	배포와 저장 및 외부 요청
장애 관찰	주소와 오류 및 관련 실행
복구 기준	돌아갈 판본과 자료 처리
인계 대상	담당 역할과 필요한 접근

생활과 업무의 적용

개인 독서 기록은 내려받은 메모를 다시 불러오는 시험을 한다. 장비 예약은 복원 뒤 중복 예약이 생기지 않는지 본다. 연수 신청은 이미 접수한 사람의 자료가 누락되지 않는지 확인한다. 복구 목표는 서비스가 다루는 자료에 맞춰 정한다.

검토 질문 백업 파일만 있으면 복구 계획은 끝날까? **해설** 실행 조건과 접근 경로 및 실제 복원 결과도 필요하다. 사용할 수 있는 상태로 돌아오는 과정을 시험한다.

기술 해설 / 관찰과 비용의 기준을 정한다

실패율과 응답 시간 및 호출 수를 무엇으로 셀지 정한다. 사용자가 느끼는 문제와 운영 수치의 연결을 읽는다. 경보와 실제 제한의 차이도 확인한다.

대시보드 20개가 분마다 요청하면 한 화면과 비용 구조가 다르다. 중복 요청과 초기 호출 및 재시도를 나누어 조사한다.

관찰 시점 / 적용 판본 / 요청 종류 / 결과 / 필요한 식별값

집계와 비용 및 복구

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x 가 입력을 받고 `return`이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L09 / 준비물 수량과 금액 합계

품목별 단가와 수량으로 가상의 합계를 계산한다. 이 사례의 입력 계약은 `items`는 `price`와 `qty`를 가진 배열 / 모두 0 이상다.

```
for (const item of x.items) {
  if (!Number.isFinite(item.price) || item.price < 0 ||
      !Number.isInteger(item.qty) || item.qty < 0) throw Error('
    값');
}
return x.items.reduce((sum, item) => sum + item.price * item.qty,
  0);
```

단가 1000에 수량 3을 곱하면 3000이다. 다른 품목의 500과 2는 1000이므로 합계는 4000이다. 화면에서 통화 기호를 붙이는 일과 계산할 숫자를 보관하는 일을 나누어 본다.

입력	기대 결과
<code>{"items": [{ "price": 1000, "qty": 3}, {"price": 500, "qty": 2}]}</code>	4000
<code>{"items": []}</code>	0

해석의 경계. 통화와 세금은 예제 범위 밖이며 실물 청구와 구분한다. 단가 변경 전후의 합계 차이를 품목별로 보여 준다.

분석 사례 L13 / 예산의 남은 범위

가상의 정한 한도와 지출 합계를 비교한다. 이 사례의 입력 계약은 limit은 한도 / expenses는 0 이상 숫자 배열다.

```
if (!Number.isFinite(x.limit) || x.limit < 0 ||
    x.expenses.some(v => !Number.isFinite(v) || v < 0)) throw Error('금액');
const spent = x.expenses.reduce((a, b) => a + b, 0);
return {spent, remaining: x.limit - spent, exceeded: spent > x.limit};
```

한도 10000에서 지출 3000과 2000의 합 5000을 빼면 5000이 남는다. 지출이 한도보다 크면 음수와 초과 표시를 함께 만든다. 이것은 가상의 계산이며 실제 정산 항목을 자동으로 반영하지 않는다.

입력	기대 결과
<code>{"limit": 10000, "expenses": [3000, 2000]}</code>	<code>{"spent": 5000, "remaining": 5000, "exceeded": false}</code>
<code>{"limit": 1000, "expenses": [1200]}</code>	<code>{"spent": 1200, "remaining": -200, "exceeded": true}</code>

해석의 경계. 초과했을 때 남은 금액을 0으로만 가리지 않는다. 실제 사용 전 금액 단위와 정산 규칙을 별도로 정한다.

분석 사례 L37 / 호출 수의 가상 추정

사용 화면 수와 호출 간격으로 요청 수를 추정한다. 이 사례의 입력 계약은 screens와 minutes는 0 이상 / interval은 분 단위의 양수다.

```
if (![x.screens,x.minutes,x.interval].every(Number.isFinite) ||
    x.screens < 0 || x.minutes < 0 || x.interval <= 0) throw Error(
    '범위');
return x.screens * Math.floor(x.minutes / x.interval);
```

20개 화면이 60분 동안 매분 요청하면 주기 호출은 1200번이다. 실제로는 처음의 즉시 호출과 실패 재시도 및 캐시도 영향을 줄 수 있으므로 이 계산을 청구 금액으로 단정하지 않는다.

입력	기대 결과
<code>{"screens": 20, "minutes": 60, "interval": 1}</code>	1200
<code>{"screens": 0, "minutes": 60, "interval": 1}</code>	0

해석의 경계. 초기 호출은 제외한 주기 호출 모형이며 실제 청구와 다르다. 초기 호출과 재시도 및 캐시의 영향을 항목으로 더한다.

분석 사례 L38 / 오류 종류별 발생 수

필요한 오류 종류만 모아 개수를 계산한다. 이 사례의 입력 계약은 events는 kind와 ok를 가진 가짜 로그 배열다.

```
const counts = new Map();
for (const event of x.events) {
  if (event.ok) continue;
  counts.set(event.kind, (counts.get(event.kind) || 0) + 1);
}
return Object.fromEntries(counts);
```

성공한 이벤트를 제외하고 실패한 종류를 Map으로 센다. network가 두 번이면 2가 된다. 전체 본문을 남기지 않고도 필요한 오류 종류와 시점 및 판본을 연결할 수 있는지 검토한다.

입력	기대 결과
<code>{"events": [{"kind": "network", "ok": false}, {"kind": "network", "ok": false}, {"kind": "read", "ok": true}]}</code>	<code>{"network": 2}</code>

해석의 경계. 모든 본문과 비밀을 로그에 넣지 않는다. 시간 구간과 판본 정보를 더해 원인을 좁힌다.

분석 사례 L39 / 알려진 비밀 필드 가리기

정한 키 이름에 한정해 값을 가리는 모형을 만든다. 이 사례의 입력 계약은 record는 평평한 객체 / password와 token은 제외할 키다.

```
return Object.fromEntries(Object.entries(x.record).map(([key, value]) =>
  [key, ['password', 'token'].includes(key.toLowerCase()) ? '[가림]' : value]
));
```

키 이름이 token이나 password인 평평한 필드만 가린다. 다른 이름과 중첩 구조 및 문자열 안의 비밀은 그대로 남을 수 있다. 실습 결과를 완전한 로그 보안 도구로 오해하지 않는다.

입력	기대 결과
<pre>{"record": {"user": "A", "token": "FAKE", "password": "FAKE"}}</pre>	<pre>{"user": "A", "token": "[가림]", "password": "[가림]"}</pre>

해석의 경계. 중첩 객체와 다른 키 및 문자열 속 비밀까지 탐지하는 보안 도구가 아니다. 민감 정보가 애초에 로그로 들어오지 않도록 기록 범위를 줄인다.

분석 사례 L40 / 백업에 빠진 자료 찾기

기대 ID와 복원한 ID를 비교해 누락을 찾는다. 이 사례의 입력 계약은 expected와 restored는 ID 배열다.

```
const restored = new Set(x.restored);
return x.expected.filter(id => !restored.has(id));
```

복원한 ID의 집합에 기대한 각 ID가 있는지 본다. b가 없으면 누락 목록에 남는다. ID가 모두 있다는 사실은 제목과 메모 및 첨부 내용까지 일치한다는 증거가 아니므로 다음 검사를 연결한다.

입력	기대 결과
<pre>{"expected": ["a", "b", "c"], "restored": ["a", "c"]}</pre>	<pre>["b"]</pre>
<pre>{"expected": ["a"], "restored": ["a"]}</pre>	<pre>[]</pre>

해석의 경계. ID가 같아도 내용까지 같은지는 추가 검사다. 필드 값과 연결 자료 및 실행 설정의 복원을 함께 점검한다.

관련 근거 [\[R08\]](#) [\[R16\]](#)

21장 복원과 인수인계를 실제 절차로 만들기

AI가 다시 만들 수 있다는 기대는 백업 전략이 아니다. 같은 프롬프트가 같은 코드와 같은 데이터를 재생성한다는 보장은 없기 때문이다. 실행 가능한 소스와 필요한 자료 및 설정과 권한의 복구 경로를 별도로 보존한다.

자료 탐색기의 소스 파일과 개인 즐겨찾기 데이터는 다른 자산이다. 공용 요청판에서는 서버 코드와 데이터베이스 및 인증 설정과 운영 문서가 추가된다. Git 이력의 복원에 성공해도 이 자산들이 빠지면 서비스가 복구되지 않을 수 있다.

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.

resource-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 결제 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 resource-lab.bundle --all
git bundle verify resource-lab.bundle
git clone resource-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.

백업과 비용과 인수인계

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 중단을 뜻하지는 않으므로 실제 중단 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 남는 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.

적용과 확인 | 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/resource-lab.git resource-lab-backup.git
git -C resource-lab-backup.git fsck --full
git clone resource-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.

관련 근거 [\[G62\]](#) [\[G35\]](#) [\[G63\]](#)

Bundle로 오프라인 이력 보관과 복원

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 ../resource-lab.bundle --all
git bundle verify ../resource-lab.bundle
git bundle list-heads ../resource-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.

적용과 확인 | 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 ../resource-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 기존 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.

관련 근거 [\[G83\]](#) [\[G62\]](#)

22장 자료 탐색기를 설계 근거와 함께 완성하기

작은 프로그램도 아키텍처를 가진다

자료 탐색기는 한 파일로 제공되지만 논리적으로는 데이터와 상태 및 계산과 표시의 경계를 갖는다. 자료 배열은 원본이다. 검색어와 선택한 과목 및 즐겨찾기 ID는 상태다. 필터 함수는 이 입력을 사용해 결과를 계산한다. 렌더 함수는 결과를 DOM에 반영한다. 파일을 여러 개로 나누지 않아도 이 책임을 구분할 수 있다.

처음부터 서버를 넣지 않은 이유는 사용자 간 공유 요구가 없기 때문이다. 저장하는 값도 개인의 즐겨찾기 ID에 한정한다. 이 선택은 개인정보와 계정 운영의 복잡성을 줄인다. 이후 공유 기능이 필요해지면 서버를 추가할 수 있지만 그 시점에는 인증과 데이터 소유 관계 및 동시성 문제를 새로 설계해야 한다.

이 프로젝트의 제목 검색은 자연어 의미 검색이 아니다. 정규화된 제목에 검색어가 포함되는지를 확인하는 문자열 검색이다. 학생 활동이라는 검색어가 학습 활동을 자동으로 찾지는 않는다. 기능을 설명할 때 이 한계를 명확히 해야 사용자가 잘못된 기대를 갖지 않는다. AI를 사용해 만들었다는 사실과 앱 자체가 AI 검색을 제공한다는 사실은 다르다.

검색 규칙을 코드와 테스트로 추적한다

과목 필터와 제목 검색을 동시에 적용하려면 두 조건을 논리곱으로 결합한다. 한 조건을 만족하면 된다는 논리합으로 구현하면 결과가 넓어진다. 국어 과목을 선택하고 토론을 검색하는 사례에서 영어 토론 자료가 나타나면 이 오류를 의심할 수 있다. 테스트 데이터에 조건 하나만 만족하는 항목을 의도적으로 넣어야 결합 규칙을 검증할 수 있다.

원본 자료를 정렬하거나 필터링할 때 입력을 직접 변경하는지도 중요하다. 원본 배열의 순서가 바뀌면 초기화 이후의 목록이 달라질 수 있다. 입력 보존을 요구한다면 복사본을 정렬한다. 불변성을 무조건 종교처럼 적용할 필요는 없지만 변경 가능한 데이터의 소유권을 명확히 해야 한다.

즐거찾기만 보기에서는 항목을 해제하면 화면에서 즉시 사라진다. 이 변화는 정확한 동작이면서 초점 관리의 문제를 만든다. 사용자가 다음에 어디에서 이어서 작업할지 결정해야 한다. 제공한 실행 예시는 가능한 경우 같은 자료의 조작 위치를 유지하고 대상이 사라진 경우 필터로 돌아가도록 처리한다. 이 동작은 브라우저에서 직접 확인해야 한다.

리팩터링은 관찰 가능한 행동을 보존하며 구조를 바꾸는 일이다

자료 개수가 늘거나 필터가 복잡해지면 계산 함수와 화면 함수를 나누고 싶어질 수 있다. 리팩터링의 목표는 기능을 추가하는 것과 다르다. 외부 행동을 유지하면서 내부 구조를 개선한다. 이때 기존 수용 기준과 테스트가 기준점이 된다. 구조 변경과 기능 변경을 동시에 진행하면 실패의 원인을 구분하기 어려워진다.

리팩터링을 AI에게 맡길 때는 보존할 행동과 공개 인터페이스를 명시한다. 코드가 짧아졌다는 사실만으로 성공을 판단하지 않는다. 읽기 쉬운 이름과 의존성의 방향 및 테스트 가능성이 개선되었는지 살핀다. 추상화가 늘면서 오히려 작은 기능을 이해하는데 여러 파일을 따라가야 한다면 개선의 이익을 다시 평가한다.

이 장의 실행 파일과 핵심 함수 실험은 본문의 주장을 관찰할 수 있는 자료다. 독자는 코드를 복사하는 데서 멈추지 않고 요구 한 가지를 바꿔 어떤 부분이 영향을 받는지 확인해야 한다. 예를 들어 검색 대상을 설명 필드까지 넓히면 검색 함수와 라벨 및 테스트 데이터와 사용자 안내가 함께 바뀐다.

검색 결과는 원본과 상태에서 계산되는 값이다

자료 탐색기의 핵심은 원본 자료와 사용자 상태 및 화면을 구분하는 데 있다. 원본에는 ID와 제목 및 과목과 주소가 있다. 사용자 상태에는 검색어와 선택한 과목 및 즐겨찾기 ID 집합이 있다. 화면에 보이는 목록은 이 둘에서 계산한 결과다. 결과 목록을 별도로 저장하고 모든 사건에서 직접 수정하면 원본과 상태 사이의 불일치가 생기기 쉽다.

사용자가 과목을 선택한 뒤 검색어를 입력하는 경우와 검색어를 입력한 뒤 과목을 선택하는 경우를 비교해 보자. 최종 상태가 같다면 결과 목록도 같아야 한다. 이 성질은 구현이 사건의 순서에 불필요하게 의존하는지 확인하는 기준이 된다. 이전 화면에서 일부 항목을 숨기는 방식만 누적하면 조작 순서에 따라 결과가 달라질 수 있다.

따라서 각 입력 사건에서는 상태를 갱신하고 같은 선택 함수를 다시 실행하는 구조를 사용할 수 있다. 작은 자료에서는 이 단순함이 충분한 성능을 제공할 수 있다. 자료량이 커져 최적화가 필요해져도 무엇을 계산해야 하는지 명확한 기준이 남는다. 처음부터 복잡한 캐시를 넣기보다 올바른 계산 경로를 먼저 확보하는 이유다.

즐거찾기는 선택 상태와 저장 결과를 연결한다

즐거찾기 버튼을 누르면 화면의 선택 상태와 저장소의 값이 함께 관련된다. 화면은 바뀌었지만 저장에 실패할 수 있으므로 사용자에게 무엇이 유지되는지 설명해야 한다. 현재 탭에서만 선택이 유지되는지와 다시 열어도 복원되는지를 구분한다. 저장 실패를 알리지 않으면 사용자는 기억될 것이라고 기대하고 나중에 결과를 잃을 수 있다.

즐거찾기에는 제목보다 안정적인 ID를 저장하는 편이 유리하다. 제목이 바뀌어도 같은 자료를 가리킬 수 있기 때문이다. 다만 ID가 재사용되면 이전 즐거찾기가 다른 자료를 가리킬 수 있다. ID는 단순한 문자열이 아니라 자원의 동일성을 유지하는 계약이다. 자료를 삭제한 뒤 같은 ID를 다른 항목에 부여하지 않는 정책이 필요할 수 있다.

원본 자료에서 사라진 ID가 저장소에 남아 있을 때의 동작도 정한다. 표시 단계에서 무시하면서 원본 저장을 보존할지와 정리할지 선택할 수 있다. 잠시 자료를 불러오지 못한 상태를 삭제로 오인해 즐거찾기를 모두 지우지 않도록 주의한다. 데이터가 없다는 관찰과 데이터가 영구적으로 삭제되었다는 판단은 다르다.

빈 결과는 기능의 일부다

검색 결과가 없을 때 빈 화면만 보이면 사용자는 로딩 중인지 오류인지 알기 어렵다. 입력 조건과 결과 없음의 의미를 설명하고 초기화 경로를 제공하면 다음 행동이 분명해진다. 이 상태는 예외적인 장식이 아니라 검색 기능의 정상적인 결과 중 하나다. 검색어와 과목의 조합이 실제 자료를 제외하는지 확인해야 한다.

빈 자료 집합으로 앱을 시작하는 상황도 시험한다. 검색 결과 없음과 자료를 아직 불러오지 못한 상태를 같은 문구로 표현하면 원인 분석이 어려워진다. 정적 앱에서는 초기 자료가 코드에 포함되어 있어 이 구분이 단순할 수 있지만 원격 자료로 확장하면 로딩과 성공 및 실패 상태가 필요하다. 기능의 구조가 바뀔 때 상태 모형도 함께 바뀐다.

접근성 측면에서는 결과 개수의 변화가 보조 기술 사용자에게 전달되는지와 버튼의 상태가 의미 있게 표현되는지 확인한다. 색상만 바꾸는 즐거찾기는 선택 여부를 알기 어려울 수 있다. 화면이 아름다운지와 조작의 의미가 전달되는지는 별도의 품질 기준이다. 같은 사례에서 데이터의 정확성과 사용 가능성을 함께 평가할 수 있다.

완료 판단은 실제 파일의 행동과 연결한다

이 책에 포함된 실행 HTML은 검색과 과목 선택 및 즐거찾기 조작을 직접 확인할 수 있게 만든 정적 사례다. 여기에서 확인한 결과를 로그인과 서버 저장 및 기기 간 동기화까지 구현했다는 주장으로 확대하지 않는다. 파일 하나의 실용성을 인정하면서도 제공하는 계약의 범위를 정확히 설명한다.

완성 후 작은 변경을 하나 수행해 본다. 새 과목을 추가하고 자료 제목 하나를 바꾸며 기존 즐거찾기가 어떻게 보이는지 확인한다. 초기 시연만 통과하는 코드와 수정 가능한 코드는 다르다. 변경 지점을 찾을 수 있고 기대 결과를 예측하며 기존 동작을 다시 확인할 수 있다면 구조를 이해했다는 증거가 된다.

선형 검색의 장점은 명세와 직접 연결된다는 데 있다

작은 자료 목록에서는 각 항목에 같은 조건을 적용하는 검색이 이해하기 쉽다. 제목의 포함 여부와 과목 및 즐겨찾기 조건이 코드에서 직접 드러난다. 원본 자료가 바뀌면 같은 규칙으로 다시 계산할 수 있다. 복잡한 색인을 도입하지 않아도 요구의 정확성을 검증하기 좋은 구조다.

자료가 커지면 정규화된 제목을 미리 계산하거나 색인을 만드는 선택이 가능하다. 그러나 제목이 바뀔 때 파생 필드와 색인도 갱신해야 한다. 검색 속도를 얻는 대신 갱신 일관성의 책임이 생긴다. 최적화를 제안받았을 때는 빨라지는 경로와 새로 생기는 실패 경로를 함께 설명하게 한다.

검색 규칙 자체를 의미 검색으로 바꾸면 평가 문제도 달라진다. 문자열 포함 여부는 명세로 판정할 수 있지만 의미적 관련성은 사용 목적과 사람의 판단에 영향을 받는다. 화면이 같은 검색창이라고 해서 같은 테스트로 검증할 수 있는 것은 아니다. 기술 선택은 평가 계약의 선택이기도 하다.

같은 기능을 다른 저장 방식으로 옮길 때 보존할 것을 정한다

로컬 즐겨찾기를 서버 저장으로 바꾸어도 검색과 선택의 의미는 보존할 수 있다. 반면 실패와 지연 및 계정 경계는 새로 생긴다. 저장 함수의 이름만 유지한다고 같은 계약이 되는 것은 아니다. 완료 시점과 재시도 및 다른 기기에서의 관찰을 추가로 정의해야 한다.

이식 과정에서 먼저 보존할 행동을 적는다. 선택한 자료가 어떤 순서로 보이는지와 삭제된 자료를 어떻게 처리하는지 및 저장 실패 시 현재 화면이 어떻게 반응하는지를 확인한다. 새 기술의 장점만 설명하는 대신 기존 사용자의 기대가 어떻게 유지되거나 바뀌는지 연결한다.

작은 프로그램의 완성도는 큰 기술을 사용했는지보다 이런 계약을 일관되게 지키는지에 달려 있다. 현재의 정적 버전을 이해한 뒤 공용 시스템으로 확장하면 추가된 복잡성의 이유를 설명할 수 있다. 이것이 예시를 따라 만든 경험을 설계 지식으로 바꾸는 과정이다.

사례 | 검색과 즐겨찾기를 결합한다

과학을 선택하면 실험 변수 기록 한 개가 남는다. 그 항목을 즐겨찾기한 뒤 검색 조건을 초기화한다. 전체 여섯 개가 다시 보이면서 그 항목의 선택 상태는 유지돼야 한다. 즐겨찾기만 보기까지 켜면 선택한 항목만 남는다.

입력과 행동	기대 결과
처음 열기	여섯 개
과학 선택	한 개
실험 입력과 과학	한 개
분수 입력과 과학	결과 없음
조건 초기화	여섯 개와 즐겨찾기 유지
즐겨찾기만 보기	선택한 항목만 표시

예제 코드를 읽는 순서

자료 목록을 찾는다. 선택 상태를 보관하는 Set을 찾는다. 필터 함수가 모든 조건을 함께 읽는지 본다. 카드 제목을 `textContent`로 넣는지와 버튼이 누른 상태를 전달하는지 확인한다. 저장 실패 안내도 실제 보관 결과에 맞게 연결되는지 읽는다.

고치기 쉬운 다음 변경

새 과목을 추가하려면 자료와 선택 목록을 함께 확인한다. 자료를 여섯 개에서 열두 개로 늘린다면 기존 검사 표의 개수도 조정한다. 제목만 바꿀 때는 ID를 유지해 즐겨찾기 연결이 남는지 본다.

적용과 확인 | 스스로 바꾸고 검사한다

1. 자료 제목 두 개를 자신의 수업에 맞게 바꾼다.
2. 기존 ID는 유지한다.
3. 제목에 함께 들어가는 검색어 하나를 정한다.
4. 전체 과목과 특정 과목에서 결과 수를 비교한다.
5. 좁은 화면과 키보드에서 같은 행동을 확인한다.

개선 과제	완료 기준
학년 필터 추가	과목과 검색어와 학년을 모두 만족

개선 과제	완료 기준
설명 문장 추가	제목 검색의 범위를 안내와 일치
즐거찾기 내보내기	검증 가능한 ID 파일 생성
손상된 파일 가져오기	기존 선택을 잃지 않고 거절

위 개선 과제는 기본 예제에 모두 들어 있는 기능을 뜻하지 않는다. 한 가지씩 추가하며 기준을 세우는 연습이다. 공유 서버와 사용자 계정이 필요해지면 10장과 11장의 설계로 돌아간다.

검토 질문 브라우저 자료실을 만들었으니 학교 공용 자료실로 바로 쓸 수 있을까? **해설** 공용 자료의 저장과 관리 권한 및 자료 보관과 접근 조건을 추가로 설계해야 한다. 이번 예제는 개인 도구의 출발점이다.

같은 규칙을 다른 자료에 적용

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x 가 입력을 받고 `return`이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L41 / 분수의 크기 비교

작은 정수의 분자를 교차 곱해 두 분수를 비교한다. 이 사례의 입력 계약은 a 와 b 는 n 과 d / 양의 분모와 안전한 작은 정수 사용다.

```
for (const f of [x.a,x.b]) {
  if (!Number.isInteger(f.n) || !Number.isInteger(f.d) || f.d <=
    0) throw Error('분수');
}
const left = x.a.n * x.b.d, right = x.b.n * x.a.d;
if (!Number.isSafeInteger(left) || !Number.isSafeInteger(right))
  throw Error('범위');
return left === right ? '=' : left < right ? '<' : '>';
```

1/2와 2/3을 비교할 때 1 곱하기 3과 2 곱하기 2를 비교한다. 3이 4보다 작으므로 앞 분수가 작다. 분모가 양수이고 계산한 정수가 안전한 범위라는 조건을 함께 확인한다.

입력	기대 결과
<code>{"a": {"n": 1, "d": 2}, "b": {"n": 2, "d": 3}}</code>	<code>"<"</code>
<code>{"a": {"n": 2, "d": 4}, "b": {"n": 1, "d": 2}}</code>	<code>"="</code>

해석의 경계. 분모 0과 계산 정수 범위를 먼저 검사한다. 기호뿐 아니라 교차 곱의 두 값을 학습 설명으로 보여 준다.

분석 사례 L42 / 사료의 연도 범위 찾기

시작 연도와 끝 연도 사이 자료만 찾아 시간순으로 읽는다. 이 사례의 입력 계약은 items는 year와 title / start와 end는 포함 경계다.

```
if (x.start > x.end) throw Error('연도 범위');
return x.items.filter(i => i.year >= x.start && i.year <= x.end)
    .sort((a,b) => a.year - b.year);
```

연도가 시작과 끝 경계를 모두 포함하는 자료만 고른다. 그 뒤 연도를 숫자로 비교해 정렬한다. 이 함수는 사료의 내용과 해석을 판정하지 않으므로 출처와 주장을 별도로 읽는다.

입력	기대 결과
<code>{"items": [{"year": 1950, "title": "A"}, {"year": 1945, "title": "B"}, {"year": 1960, "title": "C"}], "start": 1945, "end": 1950}</code>	<code>[{"year": 1945, "title": "B"}, {"year": 1950, "title": "A"}]</code>

해석의 경계. 이 예제는 연도 정수이며 사료 해석의 정확성까지 판정하지 않는다. 원문 링크와 관점 및 작성 시점을 별도 필드로 연결한다.

분석 사례 L43 / 다시 읽을 영어 표현

학습자가 정한 다시 읽기 날짜에 도달한 카드를 고른다. 이 사례의 입력 계약은 items는 id와 next / today는 같은 형식의 유효 날짜다.

```
return x.items.filter(i => i.next <= x.today).map(i => i.id);
```

다시 읽기 날짜가 오늘보다 이전이거나 오늘인 카드만 고른다. 이 결과는 사용자가 정한 일정의 목록이다. 학습 효과를 자동으로 판단하는 시스템이나 검증된 간격 처방을 뜻하지 않는다.

입력	기대 결과
<pre>{ "items": [{ "id": "a", "next": "2026-10-23", }, { "id": "b", "next": "2026-10-25" }], "today": "2026-10-24" }</pre>	["a"]

해석의 경계. 선택한 학습 간격 모형을 효과가 검증된 처방처럼 설명하지 않는다. 다시 읽은 뒤 날짜를 어떻게 갱신할지 사용자가 정하게 한다.

분석 사례 L44 / 한 가지 조건만 다른 관찰

두 기록에서 다른 필드를 찾아 비교 조건을 읽는다. 이 사례의 입력 계약은 a와 b는 같은 필드를 가진 가짜 관찰 조건 객체다.

```
const keys = [...new Set([...Object.keys(x.a), ...Object.keys(x.b)])];
return keys.filter(key => x.a[key] !== x.b[key]);
```

같은 필드의 값을 나란히 비교해 다른 이름을 모은다. light만 다르면 하나가 남고 water도 다르면 둘이 남는다. 통제 조건의 목록을 확인하는 도구이며 과학적 타당성 전체를 판정하지 않는다.

입력	기대 결과
<pre>{ "a": { "light": 3, "water": 10 }, "b": { "light": 5, "water": 10 } }</pre>	["light"]
<pre>{ "a": { "light": 3, "water": 10 }, "b": { "light": 5, "water": 20 } }</pre>	["light", "water"]

해석의 경계. 조건이 하나 다르다는 사실만으로 실험의 타당성이 보장되지는 않는다. 측정 방법과 반복 및 통제 조건을 함께 기록한다.

분석 사례 L45 / 바로 전 질문을 피한 선택

후보 목록에서 직전에 쓴 질문을 제외한다. 이 사례의 입력 계약은 ids는 중복 없는 질문 ID / previous는 직전 ID다.

```
return x.ids.filter(id => id !== x.previous);
```

직전 질문 ID가 b이면 a와 c를 후보로 남긴다. 후보가 a 하나뿐이고 직전도 a라면 빈 목록이 된다. 이때 계속 돌리는 대신 후보가 없다는 안내나 새 정책을 정한다.

입력	기대 결과
<code>{"ids": ["a", "b", "c"], "previous": "b"}</code>	<code>["a", "c"]</code>
<code>{"ids": ["a"], "previous": "a"}</code>	<code>[]</code>

해석의 경계. 남은 후보가 없을 때의 화면 안내를 정한다. 실제 무작위 선택을 붙이면 재현 가능한 검사 방법을 더한다.

관련 근거 [\[R07\]](#) [\[R08\]](#) [\[R13\]](#)

23장 공용 업무 요청판의 시스템 설계

개인 도구에서 공용 시스템으로 넘어가는 순간

개인 자료 탐색기는 한 브라우저의 상태를 관리한다. 공용 요청판은 여러 사용자가 같은 자료를 읽고 수정한다. 사용자 식별과 권한 및 데이터 일관성이 핵심 요구로 등장한다. 이 차이는 화면에 로그인 버튼을 추가하는 정도의 변화가 아니다. 데이터의 권위가 브라우저에서 서버로 이동하고 서버가 업무 규칙을 강제해야 한다.

요청 데이터에는 ID와 제목 및 작성자와 담당자 및 상태와 버전이 필요할 수 있다. 각 필드가 필요한 이유를 먼저 설명한다. 버전은 동시 수정 충돌을 찾기 위한 값이며 화면 표시 순서가 아니다. 작성자는 소유 관계와 감사 기록의 기준이며 사용자가 임의로 다른 사람으로 바꿀 수 있는 입력이 아니다. 이 의미가 정해져야 데이터베이스 제약과 API가 일관되게 설계된다.

감사 기록은 누가 언제 어떤 행동을 했는지 남긴다. 그러나 모든 이전 데이터의 복제본을 무기한 보관하는 것과 같지는 않다. 필요한 사건과 보존 범위 및 열람 권한을 정한다. 감사 로그를 수정할 수 있는 권한이 일반 업무 데이터와 같으면 조사 근거의 신뢰성이 약해질 수 있다.

하나의 상태 변경에 여러 조건이 결합된다

담당자가 요청을 완료하려는 상황을 생각하자. 서버는 사용자를 인증하고 대상 요청의 조직 범위를 확인하며 담당자 관계와 현재 상태 및 버전을 검사해야 한다. 어느 조건 하나라도 맞지 않으면 수정하지 않는다. 허용된 경우에는 상태 변경과 버전 증가 및 감사 사건 기록이 일관된 단위로 처리되어야 한다.

조건을 코드 여러 곳에 흩어 놓으면 일부 경로에서 검사가 빠질 수 있다. 웹 화면에서는 검증하지만 일괄 가져오기 API에서는 빠지는 식이다. 업무 규칙을 하나의 서비스 경계에서 적용하고 모든 쓰기 경로가 그 경계를 거치게 하는 설계가 필요하다. 화면 검사는 사용자에게 빠른 피드백을 주는 보조 수단이며 서버 검사의 대체물이 아니다.

동시 요청에 대한 설계는 예상 상태를 사용한 조건부 갱신으로 표현할 수 있다. 요청 ID와 버전이 일치하는 행만 수정하고 영향을 받은 행 수가 1인지 확인하는 방식이 그 예다. 실제 SQL과 트랜잭션 격리 수준 및 권한 정책은 선택한 데이터베이스의 공식 문서와 함께 검토해야 한다. 이 책은 특정 운영 데이터베이스를 연결한 완제품을 제공하지 않는다.

인수 시험은 역할과 실패 경로를 교차한다

작성자과 담당자 및 외부 사용자의 역할을 각각 준비한다. 접수와 진행 및 완료 상태를 교차해 허용과 거부를 확인한다. 오래된 버전과 존재하지 않는 ID 및 다른 조직의 ID도 사용한다. 이 조합은 단순히 테스트 개수를 늘리기 위한 것이 아니라 권한과 상태와 동시성 규칙이 함께 적용되는지 보기 위한 것이다.

인수 조건에는 기능뿐 아니라 운영 가능한 증거가 포함된다. 실패 요청을 로그에서 찾을 수 있는지와 데이터 복원이 가능한지 및 담당자가 바뀌었을 때 권한이 갱신되는지를 확인한다. 개발 환경에서 한 번 성공한 화면은 이 조건들을 모두 만족했다는 증거가 아니다.

이 장은 설계와 검증 계획을 완성하는 종합 사례다. 앞 장의 실제 정적 앱과 달리 인증 서버와 공용 데이터베이스는 구현 범위에 포함되지 않는다. 독자는 무엇이 실행되었고 무엇이 설계되었으며 무엇이 아직 검증되지 않았는지 구분해 읽어야 한다. 전문적인 문서는 이 경계를 감추지 않는다.

순수 정책 함수에서 원자적인 쓰기로 넘어간다

이 사례는 로컬 SQLite에서 버전 충돌을 관찰한다. 인증 서버와 웹 API를 구현하는 예제가 아니다. 이미 인증되었다고 가정한 사용자 ID를 함수의 인자로 주며 이 값의 진위를 확인하는 기능은 포함하지 않는다. 목적은 버전 검사와 상태 변경을 한 UPDATE의 조건으로 결합했을 때 오래된 요청을 거부할 수 있음을 보는 것이다.

SQLite의 트랜잭션은 여러 변경을 함께 확정하거나 취소하는 경계를 제공한다. 별도 연결 사이의 가시성과 쓰기 경쟁의 동작은 저널 모드와 트랜잭션 방식 등에 영향을 받는다. 다른 데이터베이스에 코드를 옮길 때는 해당 시스템의 격리와 잠금 규칙을 다시 확인해야 한다. [T13] [T14]

```
CREATE TABLE requests (  
  id TEXT PRIMARY KEY,  
  owner_id TEXT NOT NULL,  
  title TEXT NOT NULL,  
  state TEXT NOT NULL CHECK (state IN ('open', 'done')),  
  version INTEGER NOT NULL CHECK (version >= 1)  
);
```

테이블의 CHECK는 상태 이름과 버전 범위 같은 저장 제약을 보조한다. 누가 상태를 바꿀 수 있는지와 어떤 전이가 허용되는지는 별도 규칙이다. 데이터베이스 제약과 애플리케이션 규칙이 서로 보완해야 한다. 여기서는 open인 요청을 소유자가 같은 버전으로 완료하는 경우만 허용한다.

```
UPDATE requests  
SET state = 'done', version = version + 1
```

```
WHERE id = ?  
  AND owner_id = ?  
  AND state = 'open'  
  AND version = ?;
```

영향을 받은 행이 한 개라면 조건을 만족해 갱신한 것이다. 영 개라면 ID와 소유자 및 상태와 버전 중 적어도 하나가 맞지 않는다. 이 결과를 사용자에게 어떤 오류로 보여 줄지는 API 정책에서 정한다. 내부 조건을 모두 자세히 노출하면 접근할 수 없는 자원의 정보가 드러날 수 있으므로 외부 응답과 내부 진단을 분리할 수 있다.

두 사용 흐름이 모두 버전 7을 읽었다고 하자. 첫 번째 갱신이 성공하면 저장된 버전은 8이 된다. 두 번째 갱신의 WHERE version = 7 조건은 더 이상 맞지 않으므로 행을 바꾸지 않는다. 이 검사를 SELECT와 무조건 UPDATE 두 단계로 분리하면 그 사이에 다른 쓰기가 들어올 수 있다. 조건을 저장소의 쓰기와 결합하는 이유가 여기에 있다.

개정판의 실행 검증에서는 임시 데이터베이스와 두 연결로 이 순서를 확인했다. 첫 갱신과 오래된 버전의 거부 및 다른 사용자 ID의 거부와 트랜잭션 롤백을 검사한다. 이 결과는 해당 로컬 모형의 증거이며 인터넷에 배포된 요청판의 인증과 보안 전체를 검증했다는 뜻은 아니다.

상태 전이는 순서와 권한을 함께 가진다

공용 요청판의 상태를 접수와 검토 중 및 완료로 정의했다고 하자. 이 이름만으로는 허용된 변화가 정해지지 않는다. 접수에서 바로 완료할 수 있는지와 완료된 요청을 다시 열 수 있는지 및 누가 각 변화를 수행할 수 있는지를 정해야 한다. 상태 모형은 화면의 선택 목록이 아니라 업무의 진행 규칙이다.

현재 상태를 읽은 뒤 허용 여부를 확인하고 나중에 저장하는 과정 사이에 다른 사용자가 자료를 바꿀 수 있다. 읽을 때 접수였다고 저장할 때도 접수라고 가정하면 오래된 판단을 적용하게 된다. 따라서 실제 변경 시점에도 상태와 버전 및 권한의 조건이 만족하는지 확인하는 방식이 필요하다. 앞의 조건부 갱신은 이 요구를 한 문장에 묶는 예다.

버전 충돌을 발견했다고 사용자에게 다시 시도하라는 말만 보여 주면 입력한 내용을 잃을 수 있다. 최신 자료와 사용자의 수정 내용을 함께 보여 주고 다시 판단하게 할 수 있다. 어떤 필드는 자동으로 합칠 수 있지만 상태나 담당자처럼 업무 의미가 큰 값은 사람의 선택이 필요할 수 있다. 충돌 해결은 기술적 덮어쓰기보다 의미의 합의에 가깝다.

트랜잭션 밖으로 나간 효과는 별도로 다룬다

요청 상태를 완료로 바꾸면서 이메일 알림을 보낸다고 생각해 보자. 데이터베이스 트랜잭션이 실패했는데 이메일만 발송되면 사용자는 완료되지 않은 요청의 완료 알림을 받는다. 반대로 상태는 저장되었는데 알림 전송이 실패하면 알림이 누락된다. 데이터베이스의 원자성이 외부 서비스의 행동까지 자동으로 묶어 주지는 않는다.

한 설계는 상태 변경과 알림이 필요하다는 기록을 같은 트랜잭션에 저장하는 것이다. 별도 처리자가 기록을 읽고 알림을 보내며 성공 여부를 남긴다. 이때도 전송 직후 처리자가 종료되면 같은 알림을 다시 보낼 수 있으므로 중복을 고려해야 한다. 이런 구조는 문제를 없애는 마법이 아니라 실패를 추적하고 재시도할 수 있는 경계로 옮기는 방법이다.

작은 개인 도구에는 이런 구조가 과할 수 있다. 그러나 공용 업무에서 알림 누락이 큰 영향을 준다면 검토할 이유가 있다. 설계의 전문성은 복잡한 패턴을 많이 넣는 데 있지 않다. 실제 실패 비용과 운영 능력에 맞게 일관성과 복구 수준을 선택하고 남은 위험을 설명하는 데 있다.

격리 수준과 업무 불변식의 관계

트랜잭션을 사용한다는 사실만으로 모든 동시성 문제가 해결되는 것은 아니다. 저장소와 격리 수준에 따라 여러 읽기와 쓰기가 어떻게 상호작용하는지가 달라진다. SQLite의 직렬화된 쓰기와 스냅샷 동작에 관한 설명을 다른 데이터베이스의 모든 설정에 그대로 적용하면 안 된다. 실제 사용하는 저장소의 계약을 확인해야 한다. [T14]

예를 들어 담당자가 동시에 처리할 수 있는 요청을 다섯 개로 제한한다면 요청 한 행의 버전 검사만으로 충분하지 않을 수 있다. 서로 다른 요청 두 개가 같은 담당자의 현재 개수를 읽고 각각 배정되면 합계 제한을 넘을 수 있다. 보존해야 하는 불변식이 여러 행에 걸쳐 있으면 그 범위에 맞는 제약과 잠금 또는 원자적 갱신 전략을 선택해야 한다.

시험도 같은 범위를 가져야 한다. 한 요청을 두 번 갱신하는 충돌 시험과 여러 요청이 공유된 한도를 초과하지 않는 시험은 다른 문제다. 테스트가 통과한 조건을 구체적으로 적으면 어떤 동시성 규칙이 아직 검증되지 않았는지 볼 수 있다. 모든 동시성에 안전하다는 표현보다 확인한 불변식과 실패 시점을 명시하는 편이 정확하다.

같은 행의 충돌과 여러 행의 불변식은 다르다

버전 번호를 이용한 조건부 갱신은 같은 요청을 두 사람이 동시에 수정하는 충돌을 발견하는 데 유용하다. 그러나 여러 요청에 걸친 규칙은 한 행의 버전만으로 보호되지 않을 수 있다. 담당자가 말할 수 있는 진행 중 요청의 최대 개수를 제한한다면 여러 행의 합계를 포함하는 조건이 된다.

두 트랜잭션이 현재 개수가 네 개라고 읽고 각각 다른 요청을 추가한다고 하자. 두 작업이 각자의 행만 수정하면 직접적인 행 충돌 없이 최종 개수가 여섯 개가 될 수 있다. 각자의 읽기에서는 규칙을 지킨다고 생각했지만 함께 실행한 결과는 규칙을 위반한다. 이런 사례는 읽은 집합과 쓰는 범위의 관계를 이해해야 설명할 수 있다.

시점	작업 A	작업 B
읽기	담당자의 진행 건수 4 확인	같은 건수 4 확인
판단	한 건 추가 가능	한 건 추가 가능
쓰기	요청 A를 담당자에게 배정	요청 B를 담당자에게 배정
결과	두 작업을 합치면 6건	업무 한도 5건 위반

PostgreSQL 18 문서는 Read Committed와 Repeatable Read 및 Serializable의 보장과 재시도 필요성을 구분한다. 특히 반복 읽기의 안정성과 모든 트랜잭션이 어떤 직렬 순서로 실행된 것과 같은 효과를 갖는다는 보장은 다르다. 저장소 이름이 같아도 설정에 따라 판단이 달라지므로 실제 격리 수준을 확인한다. [\[T32\]](#)

직렬화 실패는 전체 작업을 다시 판단하게 한다

Serializable에서 충돌을 발견해 트랜잭션이 중단되면 단순히 마지막 SQL만 다시 실행하는 것으로 충분하지 않을 수 있다. 처음 읽은 값에 기초한 판단 전체를 다시 수행해야 한다. 재시도되는 단위 안에 외부 이메일 발송처럼 되돌리기 어려운 효과가 섞여 있으면 중복 문제가 생길 수 있다. 트랜잭션과 외부 효과의 경계를 함께 설계한다.

재시도에는 최대 횟수와 대기 정책 및 최종 실패 처리가 필요하다. 충돌이 계속되는 상황에서 무한 반복하면 자원을 소모하고 사용자는 이유를 알 수 없다. 어떤 작업이 반복되었는지와 최종 결과가 무엇인지 식별할 수 있는 기록을 남긴다. 원자적 갱신과 역등성 및 재시도는 서로 다른 문제를 보완하는 수단이다.

규칙을 데이터베이스 제약으로 표현할 수 있다면 여러 코드 경로에서 같은 규칙을 지키는 데 도움이 된다. 그러나 모든 업무 규칙을 단순한 제약 하나로 표현할 수 있는 것은 아니다. 제약과 트랜잭션 및 응용 프로그램의 정책을 적절히 나누고 어느 층이 어떤 불변식을 책임지는지 문서화한다.

상태 변경의 이력은 현재 값만으로 대체되지 않는다

현재 담당자가 누구인지 아는 것과 누가 언제 담당자를 바꾸었는지 아는 것은 다른 요구다. 감사 기록이 필요하다면 변경 주체와 이전 상태 및 새 상태와 이유를 함께 남기는 구조를 검토한다. 기록 자체의 수정 권한과 보존 기간도 정해야 한다. 로그 파일에 우연히 남아 있는 내용을 공식 이력으로 간주하지 않는다.

업무 기록에는 민감한 자료가 포함될 수 있으므로 목적에 필요한 범위를 정한다. 추적 가능성을 이유로 모든 요청 본문과 자격 증명을 저장할 필요는 없다. 감사의 질문에 답할 수 있는 정보와 접근 통제를 함께 설계한다. 자료를 오래 보관하는 능력과 적절하게 보관하는 능력은 다르다.

상태 변화 / 동시 수정에서 오래된 버전을 거부한다

동시 수정에서 오래된 버전을 거부한다

1 / 4 두 사용자가 같은 상태를 읽는다

저장된 버전	사용자 A	사용자 B	결과
7	7 읽음	7 읽음	같은 시작

숫자 비교 함수만으로는 데이터베이스의 원자성이 구현되지 않는다. 실제 저장소의 조건부 갱신과 트랜잭션이 필요하다.

● ● ● ●

동시 수정에서 오래된 버전을 거부한다

4 / 4 사용자에게 최신 내용과 재시도 경로를 제공한다

저장된 버전	사용자 A	사용자 B	결과
8	완료	최신 자료 다시 읽음	재조정

숫자 비교 함수만으로는 데이터베이스의 원자성이 구현되지 않는다. 실제 저장소의 조건부 갱신과 트랜잭션이 필요하다.

● ● ● ●

숫자 비교 함수만으로는 데이터베이스의 원자성이 구현되지 않는다. 실제 저장소의 조건부 갱신과 트랜잭션이 필요하다.

1. 두 사용자가 같은 상태를 읽는다
2. 비교와 수정은 원자적으로 처리해야 한다
3. A의 변경 뒤 B의 오래된 요청은 거부한다
4. 사용자에게 최신 내용과 재시도 경로를 제공한다

사례 | 두 담당자가 같은 요청을 바꾸다

담당자 A는 요청을 완료로 바꾸고 B는 조금 전 화면에서 다시 처리 중으로 저장했다. 마지막 저장을 무조건 적용하면 A의 결과가 사라질 수 있다. 자료의 버전이나 변경 시점을 비교하고 충돌 때 사용자가 최신 상태를 읽도록 하는 방식을 검토한다.

시험	기대하는 처리
작성자가 자신의 초안 수정	허용
작성자가 타인의 요청 수정	거절
담당자가 허용된 상태 이동	허용과 기록
오래된 화면으로 다시 저장	충돌 또는 정한 병합 처리
같은 등록 요청을 재전송	중복 정책에 따른 처리

권한 판단 모형

요청한 사람의 신원을 확인한다.
 대상 요청이 실제로 존재하는지 확인한다.
 그 사람에게 그 요청의 변경을 허용하는지 검사한다.
 현재 상태에서 요청한 상태로 이동 가능한지 검사한다.
 버전이 맞으면 저장하고 필요한 변경 기록을 남긴다.

이 목록은 서버 구현의 완성 코드가 아니다. AI에게 구현을 맡길 때 빠진 판단이 있는지 확인하는 설계 모형이다. 거절할 경우 실제 자료가 유지되는지도 시험한다.

적용과 확인 | 시연 대신 인수 조건을 만든다

1. 가짜 요청 다섯 개와 가짜 계정 두 개를 정한다.
2. 역할별 읽기와 수정 및 상태 이동 표를 만든다.
3. 정상 처리와 거절 및 동시 변경을 시험 항목으로 적는다.

- 구현한 범위와 아직 설계만 한 범위를 표시한다.
- 운영 전에 필요한 자료 보관과 복원 및 담당 인계를 적는다.

남길 산출물	확인 기준
문제 정의	누가 어떤 요청을 처리하는가
역할 표	직접 요청에서도 권한 검사
상태 표	허용과 거절 이동의 근거
검사 기록	실제 실행과 미실행 구분
운영 문서	로그와 복구 및 책임 주체

다른 분야로 옮기기

시설 수리 요청은 위치와 처리 상태를 더한다. 학술 모임의 검토 요청은 문헌과 담당 범위를 연결한다. 소규모 고객 문의는 외부 공개 답변과 내부 메모의 읽기 범위를 구분한다. 공용 자료가 생긴다는 공통점부터 읽는다.

검토 질문 예쁜 시연 화면과 처리 완료 애니메이션이면 공용 서비스가 완성일까? **해설** 실제 저장과 권한 및 동시 변경과 복구의 근거도 필요하다. 구현하지 않은 범위는 남은 일로 표시한다.

기술 해설 / 동시에 저장하는 문제를 나눈다

각자 화면에서 조건을 확인해도 서버에서 동시에 처리하면 충돌할 수 있다. 정원과 예약 및 재고처럼 함께 지켜야 하는 조건은 저장 시점에도 일관되게 적용돼야 한다.

정원이 한 자리 남았을 때 두 신청을 동시에 받는다. 브라우저에서 둘 다 가능이라고 계산한 결과를 최종 승인으로 사용하지 않는다.

현재 버전을 검사한다.
같은 저장 작업 안에서 변경을 반영한다.
충돌이면 최신 상태를 읽게 안내한다.

상태와 동시 수정의 정책

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x 가 입력을 받고 `return`이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L24 / 허용한 상태 이동

접수에서 처리 중 및 완료로 이동하는 규칙을 명시한다. 이 사례의 입력 계약은 from과 to는 상태 이름다.

```
const allowed = {new:['working'], working:['done','new'], done:[]};
return Object.hasOwn(allowed, x.from) && allowed[x.from].includes(x.to);
```

접수 new에서는 working만 허용한다고 명시했다. new에서 done으로 바로 가는 입력은 거절한다. 업무 정책이 달라지면 허용 표와 시험을 함께 바꾸어야 한다.

입력	기대 결과
{"from": "new", "to": "working"}	true
{"from": "new", "to": "done"}	false
{"from": "done", "to": "working"}	false

해석의 경계. 완료 후 재개를 허용할지의 정책을 별도로 정한다. 상태 이동과 그 이동을 실행할 역할을 함께 검사한다.

분석 사례 L25 / 오래된 화면의 저장 감지

요청한 버전과 현재 버전이 같을 때만 저장을 제안한다. 이 사례의 입력 계약은 expected와 current는 버전 정수다.

```
if (![x.expected, x.current].every(Number.isInteger)) throw Error('버전');
return x.expected === x.current;
```

사용자가 마지막으로 본 버전이 3이고 서버도 3이면 모형의 조건은 맞는다. 서버가 이미 4라면 오래된 화면이므로 거절한다. 실제 저장에서는 비교와 갱신 사이에 다른 변경이 끼어들지 않도록 해야 한다.

입력	기대 결과
{"expected": 3, "current": 3}	true
{"expected": 3, "current": 4}	false

해석의 경계. 비교와 저장을 별도 요청으로 나누면 사이에 값이 바뀔 수 있다. 실제 저장에서는 비교와 갱신을 원자적으로 수행하도록 설계한다.

분석 사례 L47 / 업무 요청판의 집계

항목의 상태에서 대기와 처리 중 및 완료 수를 계산한다. 이 사례의 입력 계약은 items는 state / new와 working과 done만 허용다.

```
const result = {new:0, working:0, done:0};
for (const item of x.items) {
  if (!Object.hasOwn(result,item.state)) throw Error('상태');
  result[item.state]++;
}
return result;
```

각 요청의 상태를 읽어 정해 둔 세 집계에 더한다. new 두 개와 done 한 개면 working은 0이다. 모르는 상태가 있으면 조용히 버리지 않고 오류로 처리한다.

입력	기대 결과
<code>{"items": [{"state": "new"}, {"state": "done"}, {"state": "new"}]}</code>	<code>{"new": 2, "working": 0, "done": 1}</code>

해석의 경계. 화면 카드와 집계가 같은 원본 상태를 읽는지 확인한다. 상태 이동 뒤 전체 합계가 항목 수와 같은지 검사한다.

분석 사례 L48 / 정해진 순서로 담당 나누기

간단한 순환 방식으로 담당을 배정한다. 이 사례의 입력 계약은 people은 비어 있지 않은 담당 ID 배열 / tasks는 요청 ID 배열다.

```
if (!x.people.length) throw Error('담당자');
return x.tasks.map((id,i) => ({id, owner:x.people[i % x.people.length]}));
```

요청 순서를 담당자 수로 나눈 나머지로 담당을 고른다. A와 B가 번갈아 배정되고 세 번째는 A다. 작업의 난도와 시간까지 반영한 공정한 배정이라는 주장으로 넓히지 않는다.

입력	기대 결과
<code>{"people": ["A", "B"], "tasks": ["t1", "t2", "t3"]}</code>	<code>[{"id": "t1", "owner": "A"}, {"id": "t2", "owner": "B"}, {"id": "t3", "owner": "A"}]</code>

해석의 경계. 업무 난도와 휴무 및 실제 가용성을 반영한 공정성 보장은 아니다. 담당 범위와 현재 업무량을 고려한 추가 규칙을 비교한다.

분석 사례 L50 / 권한과 버전을 함께 확인

소유 관계와 버전이 맞을 때만 다음 상태 모형을 만든다. 이 사례의 입력 계약은 `user`와 `owner / expected`와 `current / state`다.

```
if (typeof x.user !== 'string' || !x.user || x.user !== x.owner)
  return {ok:false, reason:'권한'};
if (![x.expected,x.current].every(Number.isInteger) || x.expected
  < 0 || x.current < 0)
  return {ok:false, reason:'버전'};
if (x.expected !== x.current) return {ok:false, reason:'버전'};
if (x.state !== 'new') return {ok:false, reason:'상태'};
return {ok:true, state:'working', version:x.current + 1};
```

먼저 소유 관계를 보고 그다음 버전과 상태를 본다. 하나라도 어긋나면 이유를 정해 거절한다. 모두 맞으면 다음 상태와 새 버전을 제안하지만 실제 저장까지 완료한 함수는 아니다.

입력	기대 결과
<code>{"user": "A", "owner": "A", "expected": 3, "current": 3, "state": "new"}</code>	<code>{"ok": true, "state": "working", "version": 4}</code>
<code>{"user": "A", "owner": "B", "expected": 3, "current": 3, "state": "new"}</code>	<code>{"ok": false, "reason": "권한"}</code>
<code>{"user": "A", "owner": "A", "expected": 3, "current": 4, "state": "new"}</code>	<code>{"ok": false, "reason": "버전"}</code>

해석의 경계. 판단 함수의 통과와 실제 인증과 트랜잭션 검증을 대신하지 않는다. 허용과 거절 시 실제 자료가 어떻게 유지되는지 서버 검사로 연결한다.

관련 근거 [\[R11\]](#) [\[R12\]](#)

24장 품질 평가와 지식의 인계

결과를 설명할 수 있어야 유지할 수 있다

AI와 작업한 결과를 인계할 때 프롬프트만 남기면 부족하다. 프롬프트는 의도를 보여 주지만 실제 산출물이 그 의도를 만족하는지 증명하지 않는다. 실행 방법과 데이터 구조 및 주요 의존성과 확인한 테스트를 함께 남긴다. 운영자가 수정할 가능성이 높은 설정과 자료 위치를 명확히 한다.

좋은 인계 문서는 처음 읽는 사람이 작은 변경을 수행할 수 있게 한다. 자료 탐색기에서는 자료 한 개를 추가하고 과목 필터에 나타나는지 확인하는 과제가 적합하다. 이 작업이 설명 없이 가능하다면 문서가 어느 정도 기능하는지 관찰할 수 있다. 작성자가 옆에서 모든 단계를 설명해야 한다면 숨은 지식이 아직 문서 밖에 남아 있다.

기술 부채는 단순히 코드가 지저분하다는 인상과 다르다. 현재의 선택이 이후 변경 비용을 늘리는 구체적인 경로를 설명해야 한다. 임시 데이터 구조 때문에 서버 연결 시 변환이 필요하다거나 테스트가 없어 작은 수정에도 수동 검사가 반복된다는 식이다. 부채를 기록할 때는 영향과 상환 조건 및 당장 유지하는 이유를 함께 적는다.

활동량과 가치 및 학습을 분리해 측정한다

SPACE 프레임워크는 개발자 생산성을 단일 활동 지표로 판단하지 말고 여러 차원에서 보도록 제안한다. 커밋 수나 생성된 코드 줄 수만으로 좋은 결과를 평가하기 어렵다는 관점이다. 이 책에서는 작업의 성공과 소요 시간 및 협업의 이해 비용과 사용자의 경험을 함께 관찰하는 데 이 생각을 적용한다. [S11]

자료 탐색기의 개선 전후를 비교할 때 기능 개수보다 사용자가 원하는 자료를 찾았는지를 본다. 개발 과정에서는 오류 수정에 걸린 시간과 같은 오류의 재발 여부를 기록할 수 있다. 학습자는 자신이 코드의 입력과 출력력을 설명할 수 있는지도 확인한다. AI가 더 많은 코드를 만들었다는 사실은 이해가 늘었다는 직접적인 증거가 아니다.

작은 평가 실험에서도 비교 조건을 기록해야 한다. 같은 사용자가 두 화면을 연속으로 쓰면 두 번째 화면에서는 과제에 익숙해지는 효과가 생긴다. 자료의 난이도가 다르면 시간 차이를 화면 설계의 효과로 해석하기 어렵다. 이러한 한계를 적는 것은 결과를 무력화하는 일이 아니라 어떤 결론까지 말할 수 있는지 정하는 일이다.

근거를 읽는 태도가 전문가의 습관이다

학술적인 글쓰기는 어려운 용어를 많이 쓰는 방식이 아니다. 용어를 정의하고 주장과 증거를 연결하며 적용 범위와 반례를 밝히는 방식이다. 논문은 특정 질문에 특정 방법으로 답한다. 공식 문서는 특정 제품의 계약과 현재 동작을 설명한다. 둘은 서로를 대체하지 않으며 자신의 설계 판단은 그 자료에서 도출한 해석으로 구분해야 한다.

이 책의 마지막 점검은 개념을 외웠는지를 묻는 시험이 아니다. 요구를 설명하고 실제 결과를 확인하며 실패 범위와 복구 방법을 말할 수 있는지를 묻는다. 실행하지 않은 검사를 실행했다고 표시하지 않는다. 확인하지 못한 항목은 남은 작업으로 남긴다. 이 정직한 구분이 다른 사람이 결과를 신뢰하고 이어서 작업할 수 있게 한다.

바이브 코딩의 출발점은 자연어로 빠르게 만들어 보는 경험일 수 있다. 학습의 도착점은 만들어진 결과를 읽고 판단하며 책임 있게 바꾸는 능력이다. AI가 초안을 제공할수록 사람의 역할은 사라지기보다 요구와 증거 및 경계를 다루는 쪽으로 이동한다. 마지막까지 중요한 것은 그럴듯한 설명이 아니라 설명과 실제 동작이 일치하는지 확인하는 습관이다.

같은 변화도 무엇을 측정하느냐에 따라 다르게 보인다

AI를 사용한 작업의 성과를 코드 줄 수로 측정하면 더 긴 구현이 유리해질 수 있다. 오류 수정 시간을 측정하면 처음 생성한 코드의 품질이 중요해진다. 사용자 성공률을 측정하면 코드 생성 속도보다 요구 이해와 화면 설계가 중요해질 수 있다. 측정 지표는 중립적인 장식이 아니라 어떤 결과를 좋은 것으로 볼지 정하는 선택이다.

기술 평가에서는 내적 타당도와 외적 타당도를 구분해 생각할 수 있다. 관찰한 차이가 의도한 원인에서 왔는지와 다른 환경에도 적용할 수 있는지가 각각 다른 질문이다. 같은 개발자가 먼저 수동으로 작업하고 다음에 AI로 같은 과제를 수행하면 학습 효과가 섞일 수 있다. 익숙하지 않은 과제만 선택하면 실제 업무와 다를 수 있다.

개인 학습에서는 엄격한 연구 실험을 흉내 내기보다 관찰 조건을 솔직하게 기록하는 편이 유용하다. 과제의 크기와 경험 수준 및 사용 도구와 검토 범위를 남긴다. 결과가 좋았던 사례뿐 아니라 실패한 사례도 보관한다. 선택한 성공 사례만으로 전체 능력을 판단하는 편향을 줄일 수 있다.

주장마다 필요한 증거가 다르다

함수가 특정 입력에서 올바른 값을 반환한다는 주장에는 실행 결과가 필요하다. 공용 서비스가 권한을 지킨다는 주장에는 실제 서버 경계의 시험이 필요하다. 팀의 생산성이 개선되었다는 주장에는 작업 구성과 품질을 고려한 측정이 필요하다. 하나의 테스트 통과 결과를 이 세 주장에 공통으로 사용할 수는 없다.

학술 자료를 읽을 때는 출판 상태와 방법 및 표본과 한계를 함께 본다. 프리프린트는 빠른 연구 공유에 유용하지만 동료 심사 여부가 확정되지 않았을 수 있다. 공식 문서는 제품의 계약을 확인하는 데 적합하지만 제품의 효과를 독립적으로 입증하는 실험은 아닐 수 있다. 자료의 권위는 질문과의 적합성 속에서 판단한다.

이 책의 사례와 수식은 원리를 설명하기 위해 구성한 모형이다. 논문에서 직접 보고한 수치에는 해당 출처를 붙였고 가상 숫자는 예시라고 표시했다. 독자는 그 구분을 자신의 프로젝트 문서에도 적용할 수 있다. 관찰과 추정 및 권고를 분리하면 문서는 더 신뢰할 수 있고 수정하기 쉬워진다.

작동하는 결과와 이해한 지식은 같은 산출물이 아니다

AI가 만든 앱이 작동해도 사용자가 구조를 설명하지 못할 수 있다. 반대로 원리를 이해했지만 제품의 마감이 부족할 수도 있다. 학습을 평가하려면 완성 화면만 보지 않고 작은 변경과 오류 진단 및 설계 이유의 설명을 함께 확인해야 한다. 결과물의 품질과 학습자의 이해를 구분하면 평가가 더 공정해진다.

자료 탐색기를 배운 뒤 같은 자료로 다시 따라 만들게 하면 기억한 절차만 재현할 수 있다. 자료 형식이나 필터 규칙을 조금 바꾸고 무엇을 수정해야 하는지 설명하게 하면 이해의 이동을 관찰할 수 있다. 예를 들어 과목 한 개를 고르는 규칙을 여러 과목 선택으로 바꾸면 상태의 자료형과 포함 조건 및 화면 의미가 함께 달라진다. 단순한 문구 변경보다 깊은 이해가 필요하다.

평가를 어렵게 만들기 위해 낯선 오류를 무작위로 넣을 필요는 없다. 본문에서 배운 원리와 연결된 변화가 적절하다. 원본 배열 보존을 배웠다면 정렬 뒤 초기 순서가 바뀌는 문제를 제시할 수 있다. 권한 경계를 배웠다면 다른 사용자의 ID로 요청했을 때의 결과를 설명하게 할 수 있다. 평가 과제와 학습 목표의 연결을 유지한다.

인계 문서는 다음 사람의 첫 행동을 돕는다

인계 문서에는 프로젝트의 목적과 실행 방법 및 자료의 위치와 주요 규칙이 필요하다. 여기에 현재 확인한 범위와 남은 문제 및 되돌릴 방법을 연결한다. 파일 목록만 길게 적으면 다음 사람이 무엇부터 해야 하는지 알기 어렵다. 처음 실행하고 작은 변경을 확인하는 경로를 중심으로 문서를 구성한다.

AI와의 전체 대화는 참고 자료가 될 수 있지만 인계 문서를 대신하기 어렵다. 과거의 추측과 폐기된 설계가 섞여 있기 때문이다. 현재 유효한 결정을 추려 쓰고 필요한 경우 변경 이유를 이슈나 커밋으로 연결한다. 다음 사람이 대화의 맥락을 모두 알아야만 수정할 수 있다면 프로젝트의 지식이 코드 밖에 과도하게 묶여 있는 것이다.

인계 시험은 실제로 다른 사람이 문서만 보고 실행해 보는 것이다. 작성자가 옆에서 계속 설명해야 했다면 그 지점을 문서의 결함으로 기록할 수 있다. 환경 차이로 실패했다면

필요한 버전과 설정을 보완한다. 문서의 길이보다 독립적으로 행동할 수 있게 만드는지가 중요한 기준이다.

완료 보고에는 주장과 근거의 크기를 맞추다

검색 함수의 입력 103개가 통과했다는 결과는 해당 입력에 대한 동작의 증거다. 앱의 모든 행동과 모든 브라우저 및 서버의 안전성을 증명하지 않는다. 완료 보고에는 실행한 검사와 환경 및 관찰 결과를 적고 범위를 넘어서는 주장을 피한다. 검사를 많이 실행했더라도 어떤 요구를 확인했는지 연결되지 않으면 이해하기 어렵다.

수정 후 남은 제약을 적는 일은 결과의 가치를 낮추지 않는다. 로컬 저장이 같은 브라우저에 한정된다는 사실과 공용 요청판이 설계 모형이라는 사실을 알면 독자는 적절한 용도로 활용할 수 있다. 제약을 숨긴 완성 선언보다 다음 단계의 판단을 돕는 설명이 실무적으로 유용하다.

마지막 질문은 이 결과를 어떻게 더 크게 만들 것인가에만 있지 않다. 지금의 요구를 충분히 만족하며 유지할 수 있는가도 중요하다. 필요 이상의 기능은 검증과 운영의 부담을 만든다. 바이브 코딩을 배운다는 것은 코드를 빨리 얻는 능력에 더해 무엇을 만들고 어디에서 멈추며 어떤 근거로 완료를 선언할지 판단하는 능력을 기르는 과정이다.

한 개의 점수보다 주장과 반증 조건이 중요하다

결과물의 품질을 평가할 때 각 주장에 필요한 증거와 그 증거를 무효화할 조건을 함께 적는다. 검색 규칙이 맞다는 주장에는 입력 계약과 경계 시험이 필요하다. 운영에서 권한이 보장된다는 주장에는 실제 서버의 허용과 거부 시험이 필요하다. 함수 모형의 성공으로 서버의 주장을 대신하지 않는다.

보증 논증은 어려운 형식 언어를 반드시 요구하지 않는다. 무엇을 주장하며 어떤 자료가 이를 뒷받침하고 아직 어떤 가정이 남았는지 연결하는 방법으로 시작할 수 있다. 프로젝트의 복잡성이 커질수록 이 연결을 명확히 해 두면 검토자가 추가 확인이 필요한 곳을 찾기 쉽다.

평가 질문	제출할 근거	반증할 수 있는 관찰
입력 계약을 지키는가	명세와 경계 입력 결과	허용하지 않은 타입을 정상 저장함
기존 행동을 보존했는가	변경 전후 동일 조건 비교	수정하지 않은 기능의 결과가 달라짐

평가 질문	제출할 근거	반증할 수 있는 관찰
다른 사람이 유지할 수 있는가	독립 실행과 작은 변경 기록	작성자의 설명 없이는 실행할 수 없음
실패 이후 복구할 수 있는가	분리 환경의 복원 시험	필요한 설정이나 자료를 찾지 못함

불확실성을 보고하는 문장도 검증 가능해야 한다

대체로 안전하다거나 거의 완성되었다는 문장은 판단의 근거를 알기 어렵다. 확인한 환경과 입력 및 관찰 결과를 적고 미확인 범위를 구체적으로 남긴다. 특정 브라우저에서 검색과 저장을 확인했지만 기기 간 동기화는 구현하지 않았다는 설명은 활용 범위를 분명히 한다.

학술 자료를 인용할 때도 같은 태도가 필요하다. 통제 실험과 관찰 연구 및 사례 연구는 서로 다른 질문에 답한다. 연구의 표본과 과제 및 도구의 시점을 자신의 상황과 비교한다. 논문의 권위를 빌려 모든 선택을 정당화하기보다 어떤 개념과 근거를 현재 설계에 적용했는지 밝힌다.

이 책의 사례를 완성한 뒤에는 새로운 요구 하나를 정하고 수정 계획을 작성해 본다. 바뀔 계약과 영향을 받는 코드 및 필요한 검사와 복구 방법을 설명한다. 아직 코드를 생성하지 않은 상태에서도 이 계획을 말할 수 있다면 결과를 통제할 기반이 생긴 것이다. 전문성은 생성된 코드의 양보다 그 결과를 설명하고 수정하며 검증하는 능력에 있다.

상태 변화 / 초안 시간보다 전체 작업 시간을 본다



시간은 측정 방법을 설명하는 가상 값이며 실험 결과가 아니다. 학술 연구의 수치와 자신의 관찰 및 예시 수치를 구분한다.

- 1. 문제를 이해하는 시간도 작업이다
- 2. 초안 생성 시간만 보고 결론 내리지 않는다
- 3. 오류 수정과 확인 시간을 더한다
- 4. 품질과 재작업 여부도 함께 기록한다

사례 | 다음 담당자의 첫 수정

후임자가 자료 한 개를 추가했는데 과목 선택 목록에는 새 과목이 없었다. 자료만 바꾸면 되는 줄 알았기 때문이다. 인계 문서에 데이터와 필터 및 검사 표의 연결을 덧붙이고 같은 수정을 다시 해 보게 했다. 문서를 읽기만 할 때보다 실제 수정에서 누락이 드러났다.

처음 맡길 작은 작업	관찰할 근거
자료 한 개 추가	데이터와 표시 및 검사 수 갱신
안내 문구 수정	실행 위치와 변경 비교
오류 한 개 재현	재현 순서와 같은 결과
이전 판본 실행	필요한 설정과 복원 안내

개선을 우선순위로 남긴다

사용자가 막힌 지점과 반복 오류 및 운영 비용을 기록한다. 다음 기능을 고르기 전에 어떤 문제를 줄일지 판단한다. 실제 쓰지 않는 기능을 늘리는 것보다 자주 실패하는 저장이나 불명확한 안내를 먼저 고칠 수 있다.

전문가에게 넘길 때 필요한 자료

문제의 재현 절차와 현재 구조 및 변경 기록을 제공한다. 민감한 실제 자료 대신 필요한 범위의 가짜 예시를 준비한다. 보안과 공용 데이터 및 복잡한 운영 문제가 커지면 전문 검토를 연결하고 무엇을 확인받았는지 남긴다.

적용과 확인 | 첫 판본의 인계와 회고

1. 다른 사람에게 실행 안내만 주고 도구를 열게 한다.
2. 자료 한 개를 추가하는 작은 수정을 요청한다.
3. 재질문한 지점을 기록한다.
4. 안내와 검사 표를 보완한다.
5. 다음 판본의 개선 한 가지를 고른다.

회고 질문	적을 근거
무엇이 쉬워졌는가	실제 행동의 변화
어디에서 다시 물었는가	빠진 설명이나 조건
어떤 오류가 반복됐는가	재현과 원인

회고 질문

적을 근거

무엇을 미룰 것인가

우선순위와 영향

다음에 무엇을 검증할 것인가

관찰 가능한 결과

이 책을 마친 뒤

자료 탐색기를 자신의 분야로 바꾸어 본다. 새로운 기능을 하나 추가하고 검사 표도 함께 바꾼다. 그다음 공용 저장에 필요한지 판단한다. 할 수 있는 범위를 넓히되 현재 결과를 설명할 수 있는 능력도 함께 넓힌다.

검토 질문 실력이 늘었다는 것을 코드 줄 수로 판단할 수 있을까? **해설** 요구와 변경과 검증 및 복구를 설명하는 능력도 함께 본다. 남이 만든 코드와 자신의 결과를 읽고 판단할 수 있는지가 중요하다.

기술 해설 / 인계는 실제 작은 수정으로 시험한다

다른 사람이 문서만으로 실행하고 자료 한 개를 바꾸어 보게 한다. 질문이 반복되는 부분이 안내의 빈칸일 수 있다. 정상 작업뿐 아니라 오류 확인과 복구 경로도 이어받을 수 있어야 한다.

자료 ID와 제목 및 과목을 바꿀 때 어떤 파일과 검사 표를 함께 수정하는지 실제로 해 본다. 기존 즐겨찾기의 영향을 확인한다.

목적 / 실행 방법 / 구조 / 최신 판본
대표 검사 / 알려진 한계 / 다음 작업
담당 역할 / 로그 / 복구 경로

완료 기준의 기록

다음 계산은 이 장의 규칙을 작은 입력에서 관찰하는 모형이다. 변수 x 가 입력을 받고 `return`이 계산 결과를 돌려준다. 화면과 저장소 및 서버 권한의 검증은 해당 실행 경계에서 별도로 수행한다.

분석 사례 L36 / 단계별 준비 상태 설명

숫자와 문장으로 완료 상태를 함께 알린다. 이 사례의 입력 계약은 `items`는 Boolean 배열다.

```
if (x.items.some(v => typeof v !== 'boolean')) throw Error('상태')  
;  
const done = x.items.filter(Boolean).length;
```

```
return `${x.items.length}개 중 ${done}개 완료`;
```

Boolean 배열에서 true인 개수를 센다. 전체 3개 중 2개가 완료됐다는 문장은 색만 바꾸는 표시보다 상태를 이해하는 데 도움을 준다. 자료가 0개일 때도 그 사실을 그대로 알린다.

입력	기대 결과
<code>{"items": [true, false, true]}</code>	"3개 중 2개 완료"
<code>{"items": []}</code>	"0개 중 0개 완료"

해석의 경계. 색깔만으로 완료 여부를 전달하지 않는다. 체크박스 라벨과 집계의 갱신을 실제 화면에서 확인한다.

관련 근거 [\[R02\]](#) [\[R06\]](#) [\[R14\]](#)

원리를 다른 분야로 옮기는 방법

아래 사례에서는 기능 이름보다 데이터의 의미와 판단 규칙 및 검증 경계를 비교한다. 같은 화면 구조를 재사용해도 교육과 업무의 수용 기준은 달라진다.

예시 01 | 토론 논거 분류 카드

국어 수업에서 주장을 뒷받침하는 근거를 종류별로 정리한다. 예시 입력은 교내 휴대전화 사용이라는 주제와 통계와 경험 및 전문가 의견으로 구분한 가짜 문장 여섯 개다. 학생의 실명과 실제 답안은 쓰지 않는다.

설계 항목	구체적인 예
첫 행동	통계 근거만 보기 선택
기대 결과	통계로 분류한 문장 두 개
흔한 실패	분류 이름만 바뀌고 원본은 그대로
확인 방법	여섯 문장의 분류와 결과를 직접 대조

AI가 문장의 타당성을 자동으로 판정하도록 늘리기 전에 사람이 정한 분류가 정확히 표시되는지 확인한다. 분류를 바꾸는 기능을 추가하면 수정 전후의 자료가 어디에 저장되는지도 적는다.

예시 02 | 영어 상황별 표현 카드

여행과 일상 및 학교 상황의 가짜 표현 카드를 만든다. 예시 입력은 도서관에서 도움을 요청하는 상황이다. 상황을 선택하면 해당 표현과 뜻이 나타나고 예문 열기로 추가 문장을 읽는다.

설계 항목	구체적인 예
첫 행동	학교 상황 선택 후 예문 열기
기대 결과	해당 카드의 예문만 표시
흔한 실패	모든 카드의 예문이 함께 열림

설계 항목	구체적인 예
확인 방법	서로 다른 카드 두 개를 차례로 조작

음성 기능을 추가한다면 사용 권한과 재생 오류 및 버튼 이름을 더 검토한다. 텍스트 카드가 완성됐다는 사실과 실제 발음 안내가 검증됐다는 사실은 따로 기록한다.

예시 03 | 수학 오답 유형 메모

분수 비교에서 자신의 실수 이유를 선택하고 짧게 적는 개인 도구다. 예시 입력은 분모가 큰 쪽을 무조건 크다고 판단함이다. 유형은 개념 혼동과 계산 실수 및 조건 누락으로 시작한다.

설계 항목	구체적인 예
첫 행동	개념 혼동 선택 후 메모 저장
기대 결과	같은 유형의 메모 목록에서 찾을 수 있음
흔한 실패	저장 성공 안내만 뜨고 새로 열면 없음
확인 방법	같은 주소에서 다시 열어 내용 비교

이 도구는 학습자의 성적을 자동 판정하는 시스템으로 설계하지 않는다. 메모의 목적과 저장 범위를 안내하고 개인 기록을 공용 화면에 올리지 않도록 범위를 정한다.

예시 04 | 사회 사료 비교 카드

같은 사건을 다르게 설명한 가상 자료 두 개를 나란히 읽는다. 예시 입력은 작성 시점과 작성자 역할 및 핵심 주장이다. 날짜순 보기와 관점별 보기를 전환한다.

설계 항목	구체적인 예
첫 행동	작성 시점순으로 정렬
기대 결과	연도와 월을 실제 날짜 기준으로 비교
흔한 실패	날짜 글자를 단순 문자열로 정렬

설계 항목	구체적인 예
확인 방법	2월과 11월이 있는 자료로 순서 확인

출처 링크와 인용 범위를 함께 관리한다. 시가 요약한 내용은 원문과 대조할 수 있게 하고 원문에 없는 주장을 사료의 사실처럼 표시하지 않는다.

예시 05 | 과학 관찰 변수 기록

간단한 수업 관찰에서 바꾼 조건과 비교할 결과를 정리한다. 예시 입력은 빛을 받은 시간과 관찰 날짜 및 가짜 성장 길이이다. 단위와 빈 값의 처리부터 정한다.

설계 항목	구체적인 예
첫 행동	관찰값 3.5와 단위 cm 입력
기대 결과	값과 단위를 함께 목록에 표시
흔한 실패	빈 값을 숫자 0으로 바꾸어 저장
확인 방법	빈 값과 0 및 3.5를 각각 입력

도구는 수업 기록의 구조를 돕는다. 실험의 안전이나 과학적 타당성을 화면 완성만으로 보장하지 않는다. 자료를 그래프로 옮기면 축과 단위 및 빠진 관찰도 함께 표시한다.

예시 06 | 초등 그림책 질문 선택기

그림책 수업에서 인물과 사건 및 자신의 경험을 연결하는 질문을 고른다. 예시 입력은 저학년과 인물의 마음이라는 조건이다. 사람이 검토한 질문 여덟 개로 시작한다.

설계 항목	구체적인 예
첫 행동	저학년과 인물의 마음 선택
기대 결과	두 조건에 맞는 질문 카드
흔한 실패	조건을 바꾸어도 이전 질문이 남음
확인 방법	서로 다른 조건을 연속으로 선택

질문을 무작위로 뽑는다면 바로 전 질문의 반복을 허용할지 정한다. 학생이 읽을 글자의 크기와 화면의 정보량도 실제 기기에서 확인한다.

예시 07 | 연수 준비물 체크리스트

교사가 연수 전에 계정과 파일 및 장비를 준비하는 개인 도구다. 예시 입력은 브라우저 업데이트와 가짜 실습 파일 다운로드 및 충전기 준비다. 완료 표시와 확인 날짜를 따로 둔다.

설계 항목	구체적인 예
첫 행동	실습 파일 준비 완료 선택
기대 결과	완료 수와 항목 상태가 함께 갱신
흔한 실패	색만 바뀌고 완료 집계는 그대로
확인 방법	선택과 해제를 반복해 수치 비교

완료 표시가 실제 설치나 파일 내용을 검사한 것은 아니라는 점을 안내한다. 준비물 목록의 다음 판본을 배포할 때 이전 완료 기록을 어떻게 이어 받을지도 정한다.

예시 08 | 학술 논문 읽기 카드

논문 제목과 연구 질문 및 방법과 자신의 메모를 정리한다. 예시 입력은 가상의 수업 피드백 연구다. 연구 방법과 읽기 상태로 자료를 찾는다.

설계 항목	구체적인 예
첫 행동	질적 연구와 읽는 중을 함께 선택
기대 결과	두 조건에 맞는 가짜 논문 카드
흔한 실패	제목올 바꾸자 기존 메모 연결이 끊김
확인 방법	ID를 유지하며 제목만 수정

논문 원문과 자신의 해석을 구분해 저장한다. 요약과 인용의 출처를 남기고 외부 서비스를 붙일 때 자료의 접근 조건과 보관 범위를 검토한다.

예시 09 | 모임 장비 예약

같은 장비를 같은 시간에 두 사람이 예약하지 않도록 설계한다. 예시 입력은 장비 A와 14시부터 15시까지다. 예약 구간의 끝과 다음 예약의 시작을 허용할지 먼저 정한다.

설계 항목	구체적인 예
첫 행동	14시 30분부터 15시 30분 요청
기대 결과	기존 예약과 겹치므로 거절
흔한 실패	각 브라우저에서는 비어 있어 둘 다 승인
확인 방법	서버에서 동시 요청과 구간 경계 검사

공용 저장과 서버의 일관성 처리가 필요한 설계 예시다. 브라우저 달력에 표시하는 것만으로 중복 예약 방지가 완성됐다고 설명하지 않는다.

예시 10 | 소규모 팀의 요청 분류판

가짜 업무 요청을 종류와 처리 상태로 분류한다. 예시 입력은 회의실 화면 연결 문의와 접수 상태다. 공개 안내와 담당자만 보는 내부 메모를 분리한다.

설계 항목	구체적인 예
첫 행동	작성자가 자신의 요청 열기
기대 결과	허용된 본문만 읽고 내부 메모는 제외
흔한 실패	화면에서는 숨겨도 응답에 내부 메모 포함
확인 방법	실제 서버 응답과 다른 계정의 요청 비교

누가 읽는지를 화면 구성만으로 판단하지 않는다. 반환할 필드와 자원별 권한을 함께 설계하고 거절 때 원본 자료가 유지되는지 확인한다.

예시 11 | 가게 영업 안내와 메뉴

영업 시간과 메뉴 및 문의 방법을 보여 주는 정적 사이트다. 예시 입력은 가상의 작은 카페와 휴무일 및 메뉴 여섯 개다. 예약과 결제는 첫 범위에서 제외한다.

설계 항목	구체적인 예
첫 행동	모바일에서 음료 종류 선택
기대 결과	해당 메뉴와 가격 및 안내 표시
흔한 실패	보기 좋은 이미지에만 메뉴 글자가 있음
확인 방법	텍스트 읽기와 확대 및 키보드 이동

실제 운영 정보를 공개할 때 담당자가 수정할 위치와 기준 날짜를 안내한다. 메뉴를 바꾸어도 배포가 이전 파일을 사용하면 옛 정보가 계속 보일 수 있으므로 공개 주소에서 확인한다.

예시 12 | 개인 독서 기록과 내보내기

책 제목과 읽은 날짜 및 짧은 메모를 보관한다. 예시 입력은 가상의 책과 한 문장의 감상이다. 기록을 파일로 내려받고 다시 가져오는 흐름까지 시험한다.

설계 항목	구체적인 예
첫 행동	기록 파일 내보내기 후 새 공간에 가져오기
기대 결과	제목과 날짜 및 메모가 같음
흔한 실패	잘못된 파일로 현재 기록을 모두 지움
확인 방법	손상된 파일을 거절하며 기존 자료 보존

브라우저 저장만으로 영구 보관을 약속하지 않는다. 내려받은 파일을 직접 열고 복원 시험을 한 결과를 남긴다. 개인 메모를 외부로 공개하는 기능은 별도 판단으로 추가한다.

프로젝트 기획서와 완료 조건

한 장으로 남길 기본 정보

항목	입력 예시	내 프로젝트에서 정할 것
사용자	자료를 찾는 교사	주 사용자 역할 한 종류
문제	주제 자료가 흩어져 있음	실제 번거로운 장면
첫 행동	과목을 선택한다	사용자가 시작할 행동
첫 결과	관련 제목만 남음	관찰 가능한 변화
자료	가짜 자료 여섯 개	필드와 예시
제외	로그인과 업로드	이번에 만들지 않을 범위

완료 조건 세 문장

- 1. 전체 조건에서는 가짜 자료 여섯 개가 보인다.
- 2. 과학과 실험을 함께 선택하면 해당 자료 한 개가 보인다.
- 3. 없는 제목을 입력하면 빈 결과 안내가 보인다.

의사결정 기록

결정	선택 이유	다시 볼 조건
정적 파일로 시작	공용 저장에 아직 필요 없음	사용자별 자료 필요
ID와 제목 분리	제목 수정 뒤 연결 유지	자료 가져오기 규칙 변경
첫 기능을 좁힘	검증 가능한 시점 확보	첫 사용자 시험 완료

문서의 빈칸을 AI가 모두 추측하게 두지 않는다. 미정인 부분은 미정으로 표시하고 구현 전에 결정할 항목과 실험으로 정할 항목을 나눈다.

검증 기록과 오류 보고 양식

오류를 재현할 수 있게 적는다

항목	예시
판본과 환경	v0.1과 브라우저 및 화면 너비
시작 자료	가짜 자료 여섯 개
순서	과학 선택 후 초기화
기대	전체 여섯 개
실제	한 개가 남음
오류 근거	필요한 범위의 메시지와 결과
변경	관련 함수와 상태 갱신
재검사	같은 순서와 기존 즐겨찾기 확인

검증 상태를 분명히 표시

상태	의미
통과	실제 수행한 조건에서 기대와 일치
실패	같은 조건에서 기대와 다름
미실행	환경이나 구현 등의 이유로 수행하지 않음
보류	완료 조건 자체를 다시 결정해야 함

로그에 실제 비밀과 개인정보가 포함되는지 확인한다. 실행하지 않은 검사를 통과로 적지 않는다. 증거를 찾을 수 있는 위치와 남은 확인을 함께 적으면 다른 사람이 다음 작업을 이어받기 쉽다.

AI 협업 요청과 인계 기록

작업 요청의 기본 틀

목적: 어떤 사용자의 어떤 행동을 돕는가
현재: 작동하는 기능과 재현되는 문제
범위: 이번에 바꿀 한 가지 결과
유지: 보존할 자료와 기능 및 제약
완료: 입력과 기대 결과 및 검사 방법
보고: 바꾼 파일과 실제 검사 및 남은 한계

요청을 구체화하는 예시

모호한 요청	바꾼 요청
다 고쳐 줘	이 재현 순서에서 처음 생기는 실패를 조사
자동으로 해 줘	이 입력을 이 결과로 바꾸는 한 단계 구현
저장되게 해 줘	어디에 누가 읽을 수 있게 보관할지 먼저 설명
완벽하게 해 줘	검사한 범위와 미확인 범위를 구분해 보고

새 작업자의 인계문

프로젝트 목적과 파일 구조 및 실행 방법을 적는다. 마지막으로 정상 확인한 판본과 검사 결과를 남긴다. 다음 작업과 미정인 규칙을 표시한다. 계정의 실제 비밀은 문서에 넣지 않고 담당 역할과 관리 경로를 안내한다.

용어 사전 1

용어	쉬운 뜻	함께 물을 질문
프런트엔드	사용자가 조작하는 화면 쪽	입력과 상태가 연결됐는가
백엔드	서버에서 처리하는 기능	누가 요청을 허용하는가
상태	현재 선택과 편집 중인 값	언제 어디에서 바뀌는가
데이터 모델	자료의 필드와 관계	ID와 표시값을 구분했는가
API	정한 형식으로 주고받는 통로	입력과 응답과 실패는 무엇인가
JSON	구조화된 자료 표현 형식	필드와 자료형을 확인했는가
인증	누구인지 확인	신원은 어떤 근거로 검증하는가
권한	무엇을 할 수 있는지 결정	그 자원에 그 작업을 허용하는가
세션	인증 이후의 상태 유지와 연결	종료와 만료를 어떻게 다루는가
출처	주소의 프로토콜과 호스트 및 포트 기준	같은 저장 공간과 요청 경계인가
localStorage	브라우저의 출처별 저장 공간	기기 간 공유를 기대하지 않는가
CORS	브라우저가 다른 출처 응답을 읽는 규칙	서버의 허용과 권한을 구분했는가

용어를 외우기보다 지금 만드는 기능에서 어느 위치를 가리키는지 찾아본다. 같은 저장이라는 말이라도 브라우저와 서버와 내려받은 파일은 서로 다른 운영 조건을 가진다.

용어 사전 2

용어	쉬운 뜻	함께 물을 질문
Git	변경 이력을 기록하는 도구	어떤 변경을 기록했는가
커밋	선택한 변경의 한 기록	목적과 검증이 설명되는가
브랜치	이력의 끝을 가리키는 이름	어떤 작업 기준인가
Diff	두 상태의 차이	요청 밖의 변경은 없는가
PR	변경을 검토하고 합치자는 제안	요구와 구현과 검사가 연결되는가
빌드	소스를 배포할 결과물로 준비	실제 결과 폴더는 어디인가
배포	실행할 판본을 환경에 반영	사용자 주소에서 확인했는가
환경 변수	실행 환경에 전달하는 설정	비밀인지 공개 가능한 값인지 구분했는가
도메인	사용자가 접속할 이름	어느 배포를 가리키는가
회귀	수정 뒤 기존 기능의 실패	관련 행동을 다시 시험했는가
롤백	이전 판본 등으로 복구	데이터 호환도 확인했는가
인수 조건	결과를 받아들일 확인 기준	같은 절차를 재현할 수 있는가

어려운 용어가 나타나면 예시 입력 하나를 따라가 본다. 설명과 실제 파일이 맞는지 확인하고 이해하지 못한 연결을 작은 질문으로 나눈다.

자주 묻는 질문 1

코딩을 전혀 몰라도 시작할 수 있나

작은 정적 도구부터 시작할 수 있다. 파일 위치와 입력 및 결과를 읽는 능력은 함께 익힌다. 공용 자료와 복잡한 권한이 필요한 기능은 그에 맞는 구현과 검토를 더 요구한다.

어떤 AI 도구가 가장 좋은가

프로젝트의 파일 접근과 변경 비교 및 실행과 검증 및 소스 보관 조건을 먼저 본다. 가격과 지원 기능은 현재 공식 안내에서 확인한다. 첫 결과를 자신의 환경에서 작은 범위로 시험한다.

좋은 프롬프트 하나면 계속 쓸 수 있나

프로젝트의 현재 상태와 이번 변경이 달라진다. 요청의 틀은 재사용할 수 있지만 자료와 제약 및 검사 표는 실제 상황에 맞게 바뀌어야 한다.

AI가 테스트했다고 하면 믿어도 되나

실제 명령과 환경 및 결과를 확인한다. 제안한 검사와 실제 실행한 검사를 구분한다. 자동 검사가 다루지 않은 사용 행동도 살핀다.

오류가 계속 나면 처음부터 다시 만들까

재현과 마지막 정상 판본을 먼저 확인한다. 상태를 이해하지 못한 채 전체를 다시 만들면 같은 원인을 옮길 수 있다. 작은 범위로 실패 지점을 좁힌다.

자주 묻는 질문 2

로그인 화면을 넣으면 보안이 생기나

화면과 실제 인증 및 서버의 권한 검사는 다르다. 가짜 역할 값과 다른 사람의 자원에 대한 요청도 거절되는지 확인한다.

환경 변수로 옮기면 키가 반드시 숨겨지나

브라우저에 포함되는 설정인지 서버에서만 사용하는 설정인지 확인해야 한다. 실제 배포 결과와 로그에 어떤 값이 들어가는지도 살핀다.

브라우저 저장은 영구적인가

사용자 설정과 삭제 및 실행 주소와 환경의 영향을 받는다. 중요한 자료는 내보내기와 복원 과정을 따로 시험한다.

무료로 계속 운영할 수 있나

서비스별 사용량과 제공 범위 및 제한은 바뀔 수 있다. 개발 도구와 배포 및 저장과 외부 API를 나누어 현재 조건과 청구 주체를 확인한다.

예제를 복사하면 내 서비스도 완성인가

예제의 가정과 구현 범위를 먼저 읽는다. 사용자와 자료 및 운영 조건이 달라지면 설계와 검증도 달라져야 한다. 공용 서비스의 미구현 범위는 남은 일로 표시한다.

네 번의 실습과 인계 점검

회차	할 일	남길 결과
첫째	문제 정의와 가짜 자료 준비	기획서와 검사 표
둘째	최소 기능과 작은 변경	실행 파일과 변경 이유
셋째	실패 재현과 사용성 점검	수정 전후 검사 기록
넷째	배포 검토와 복원 및 인계	적용 판본과 운영 카드

인계 전 확인할 연결

다음 판본을 정할 때

사용자가 실제로 막힌 지점을 우선 기록한다. 기능을 더 넣는 일과 이미 있는 기능을 믿을 수 있게 만드는 일을 비교한다. 수정의 기대 결과를 검사 표에 적은 뒤 작업을 시작한다. 반복할수록 대화는 짧아질 수 있어도 판단 근거는 더 분명해져야 한다.

자료 탐색기 전체 소스

14장의 실행 예시와 같은 코드다. HTML에서 완성 파일을 내려받을 수 있다. 로컬 저장의 동작은 브라우저와 실행 origin에 따라 다를 수 있다.

```
<!doctype html>
<html lang='ko'>
<meta charset='utf-8'>
<meta name='viewport' content='width=device-width,initial-scale=1
'>
<title>자료 탐색기 실습</title>
<style>

*{
  box-sizing:border-box}
body{
  font:17px/1.8 system-ui,sans-serif;
  background:#f7f4ee;
  color:#172d3a;
  margin:0;
  padding:24px}
main{
  max-width:900px;
  margin:auto}
h1{
  font-size:30px}
label{
  display:block;
  font-weight:700}
input,select,button{
  font:inherit;
  padding:10px;
  border:1px solid #b8b9b4;
  border-radius:8px;
  max-width:100%}
input[type=search]{
  width:100%}
button{
  cursor:pointer;
  background:#fff;
  color:#172d3a}
button:focus-visible,input:focus-visible,select:focus-visible{
  outline:3px solid #876429;
  outline-offset:3px}
.filters{
  display:grid;
```

```

    grid-template-columns:2fr 1fr;
    gap:14px}
.tools{
    display:flex;
    flex-wrap:wrap;
    gap:12px;
    align-items:center;
    margin:18px 0}
.tools label{
    font-size:15px}
#list{
    list-style:none;
    padding:0;
    display:grid;
    grid-template-columns:repeat(2,minmax(0,1fr));
    gap:15px}
li{
    border:1px solid #d7d5cc;
    border-radius:12px;
    background:#fff;
    padding:18px;
    overflow-wrap:anywhere}
li h2{
    font-size:21px;
    margin:0 0 5px}
li p{
    margin:0 0 12px;
    color:#526572}
button[aria-pressed=true]{
    background:#172d3a;
    color:#fff}
#storage{
    font-size:14px;
    color:#59656c}
@media(max-width:600px){
    body{
        padding:18px}
    .filters,#list{
        grid-template-columns:1fr}
    h1{
        font-size:26px}
}

</style>
<main>
<h1>자료 탐색기</h1>

```

```

<p>가짜 자료 여섯 개로 검색과 필터 및 즐겨찾기를 연습합니다.</p>
<div class='filters'>
<div><label for='query'>자료 제목 검색</label><input id='query' type
  ='search'></div>
<div><label for='subject'>과목</label><select id='subject'>
<option value=''>전체 과목</option><option>국어</option><option>영어<
  /option>
<option>수학</option><option>사회</option><option>과학</option><opti
  on>초등</option>
</select></div>
</div>
<div class='tools'>
<label><input id='only' type='checkbox'> 즐겨찾기만 보기</label>
<button id='reset' type='button'>검색 조건 초기화</button>
</div>
<p id='count' role='status' aria-live='polite'></p>
<p id='storage' role='status' aria-live='polite'></p>
<ul id='list'></ul>
</main>
<script>
'use strict';
const data = [
  {id:'k01',title:'토론 논거 카드',subject:'국어'},
  {id:'e01',title:'일상 표현 카드',subject:'영어'},
  {id:'m01',title:'분수 비교 활동',subject:'수학'},
  {id:'s01',title:'지역 변화 지도',subject:'사회'},
  {id:'c01',title:'실험 변수 기록',subject:'과학'},
  {id:'p01',title:'그림책 질문 모음',subject:'초등'}
];
const $ = id => document.getElementById(id);
const key = 'vibe-catalog-favorites-v1';
const known = new Set(data.map(x => x.id));
let favorites = new Set();
try {
  const saved = JSON.parse(localStorage.getItem(key) || '[]');
  if (Array.isArray(saved)) {
    favorites = new Set(saved.filter(id => known.has(id)));
  }
  $('storage').textContent = '즐거찾기는 이 브라우저에 보관합니다.';
} catch (_) {
  $('storage').textContent = '현재 화면에서만 즐겨찾기를 유지합니다.';
}
function save() {
  try {
    localStorage.setItem(key, JSON.stringify([...favorites]));
    $('storage').textContent = '이 브라우저에 즐겨찾기를 저장했습니다.';
  } catch (_) {

```

```

    $('storage').textContent = '저장이 제한돼 현재 화면에서만 유지합니다.
    ';
  }
}
function render() {
  const query = $('query').value.trim().toLocaleLowerCase('ko-KR'
  );
  const subject = $('subject').value;
  const shown = data.filter(item =>
    item.title.toLocaleLowerCase('ko-KR').includes(query) &&
    (!subject || item.subject === subject) &&
    (!$('only').checked || favorites.has(item.id))
  );
  $('count').textContent = shown.length ?
  `자료 ${shown.length}개` : '조건에 맞는 자료가 없습니다.';
  $('list').replaceChildren();
  for (const item of shown) {
    const li = document.createElement('li');
    const title = document.createElement('h2');
    title.textContent = item.title;
    const subjectText = document.createElement('p');
    subjectText.textContent = item.subject;
    const button = document.createElement('button');
    button.type = 'button';
    button.textContent = favorites.has(item.id) ? '즐거찾기 해제' :
    '즐거찾기';
    button.setAttribute('aria-label', item.title + ' 즐거찾기');
    button.setAttribute('aria-pressed', String(favorites.has(item
    .id)));
    button.addEventListener('click', () => {
      favorites.has(item.id) ? favorites.delete(item.id) : favori
      tes.add(item.id);
      save(); render();
      const next = Array.from($('list').querySelectorAll('button'
      )).find(
        el => el.getAttribute('aria-label') === item.title + ' 즐
        거찾기');
      if (next) next.focus();
      else $('only').focus();
    });
    li.append(title, subjectText, button);
    $('list').append(li);
  }
}
$('query').addEventListener('input', render);
$('subject').addEventListener('change', render);
$('only').addEventListener('change', render);

```

```
$('#reset').addEventListener('click', () => {  
  $('#query').value = ''; $('#subject').value = '';  
  $('#only').checked = false; render(); $('#query').focus();  
});  
render();  
</script>  
</html>
```

참고문헌과 기술 자료

학술 연구와 기술 문서 및 이 책의 해석을 구분해 읽는다. 새로운 학술 자료와 주요 기술 문서의 확인 기준은 2026년 10월 24일이다. R 자료는 원판에서 확인한 2026년 10월 24일의 기술 문서다. 논문의 결과는 해당 표본과 과제 및 시점의 범위에서 해석한다.

학술 연구와 연구기관 자료

[S01] Peng 외 / 2023

The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv 프리프린트.

활용 범위: 통제 실험의 과제 완료 시간과 일반화 범위.

<https://arxiv.org/abs/2302.06590>

[S02] Becker 외 / 2025

Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. arXiv 프리프린트.

활용 범위: 숙련 개발자와 성숙한 저장소에서의 무작위 실험.

<https://arxiv.org/abs/2507.09089>

[S03] Becker 외 / METR / 2026

We are Changing our Developer Productivity Experiment Design. 연구기관 후속 보고 / 2026-02-24.

활용 범위: 후속 실험의 선택 편향과 추정 한계.

<https://metr.org/blog/2026-02-24-uplift-update/>

[S04] Vaswani 외 / 2017

Attention Is All You Need. NeurIPS / arXiv 공개본.

활용 범위: Transformer의 attention 기반 구조.

<https://arxiv.org/abs/1706.03762>

[S05] Liu 외 / 2024

Lost in the Middle: How Language Models Use Long Contexts. TACL 12 / 157-173 / DOI 10.1162/tacl_a_00638.

활용 범위: 장문 문맥에서 관련 정보의 위치 효과.

<https://aclanthology.org/2024.tacl-1.9/>

[S06] Lewis 외 / 2020

Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS / arXiv 공개본.

활용 범위: 검색된 지식과 생성 모델의 결합.

<https://arxiv.org/abs/2005.11401>

[S07] Saltzer와 Schroeder / 1975

The Protection of Information in Computer Systems. Proceedings of the IEEE 63(9) / 1278-1308 / DOI 10.1109/PROC.1975.9939.

활용 범위: 최소 권한과 기본 거부 및 접근 검사 원칙.

<https://web.mit.edu/saltzer/www/publications/protection/>

[S08] Perry 외 / 2023

Do Users Write More Insecure Code with AI Assistants?. ACM CCS / DOI 10.1145/3576915.3623157 / 2022년 프리프린트.

활용 범위: 보안 과제에서의 코드 품질과 사용자 확산.

<https://arxiv.org/abs/2211.03622>

[S09] Jimenez 외 / 2024

SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. ICLR / 2023년 최초 공개 논문.

활용 범위: 저장소 수준 수정 과제와 평가의 범위.

<https://arxiv.org/abs/2310.06770>

[S10] Souppaya 외 / NIST / 2022

Secure Software Development Framework Version 1.1. 기술 프레임워크 / NIST SP 800-218 / DOI 10.6028/NIST.SP.800-218.

활용 범위: 개발 전 과정의 보안 활동 / 인증서가 아닌 권고 문서.

<https://csrc.nist.gov/pubs/sp/800/218/final>

[S11] Forsgren 외 / 2021

The SPACE of Developer Productivity: There is more to it than you think. ACM Queue 19(1) / 20-48.

활용 범위: 생산성의 여러 차원과 단일 지표의 한계.

<https://www.microsoft.com/en-us/research/publication/the-space-of-developer-productivity-theres-more-to-it-than-you-think/>

[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/>

[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>

[S17] Barke 외 / 2023

Grounded Copilot: How Programmers Interact with Code-Generating Models. PACMPL / DOI 10.1145/3586030 / 2022년 프리프린트.

활용 범위: 관찰 연구에서 제시한 가속과 탐색의 상호작용 모형.

<https://arxiv.org/abs/2206.15000>

[S18] Amershi 외 / 2019

Guidelines for Human-AI Interaction. ACM CHI / DOI 10.1145/3290605.3300233.

활용 범위: 초기 사용과 오류 및 장기 사용의 상호작용 설계.

<https://www.microsoft.com/en-us/research/project/guidelines-for-human-ai-interaction/>

[S19] Parnas / 1972

On the Criteria to Be Used in Decomposing Systems into Modules. Communications of the ACM 15(12) / 1053-1058 / DOI 10.1145/361598.361623.

활용 범위: 모듈 분해와 정보 은닉의 설계 논증.

<https://dl.acm.org/doi/10.1145/361598.361623>

[S20] Pearce 외 / 2022

Asleep at the Keyboard? Assessing the Security of GitHub Copilot Code Contributions. IEEE Symposium on Security and Privacy / DOI 10.1109/SP46214.2022.9833571 / 2021년 프리프린트.

활용 범위: 선정된 보안 시나리오에서의 생성 코드 분석.

<https://arxiv.org/abs/2108.09293>

[S21] Chen 외 / 2021

Evaluating Large Language Models Trained on Code. arXiv 프리프린트 / 2107.03374.

활용 범위: HumanEval과 pass@k 추정량 및 함수 수준 평가의 한계.

<https://arxiv.org/abs/2107.03374>

[S22] Liu 외 / 2023

Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. arXiv 공개본 / 2305.01210.

활용 범위: EvalPlus의 강화된 테스트와 평가 결과의 민감도.

<https://arxiv.org/abs/2305.01210>

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>

[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

[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/>

[T12] 움직이는 콘텐츠의 정지 기능

<https://www.w3.org/WAI/WCAG22/Understanding/pause-stop-hide.html>

[T13] SQLite의 트랜잭션

https://www.sqlite.org/lang_transaction.html

[T14] SQLite 연결 사이의 격리

<https://www.sqlite.org/isolation.html>

[T15] JavaScript 실행 모델

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Execution_model

[T16] TypeScript 기본 타입과 타입 정보 제거

<https://www.typescriptlang.org/docs/handbook/2/basic-types.html>

[T17] HTTP 메서드와 응답 의미 / RFC 9110

<https://www.rfc-editor.org/rfc/rfc9110.html>

[T18] git ls-files와 미병합 인덱스 단계

<https://git-scm.com/docs/git-ls-files>

[T20] git update-ref와 기대한 이전 값의 확인

<https://git-scm.com/docs/git-update-ref>

[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>

[T30] HTTP 캐시 지시자와 재검증 / RFC 9111

<https://www.rfc-editor.org/rfc/rfc9111.html>

[T31] JSON Schema 객체의 속성과 필수 값

<https://json-schema.org/understanding-json-schema/reference/object>

[T32] PostgreSQL 18 트랜잭션 격리

<https://www.postgresql.org/docs/18/transaction-iso.html>

[T33] Stryker 변이 상태와 점수의 정의

<https://stryker-mutator.io/docs/mutation-testing-elements/mutant-states-and-metrics/>

[T34] OpenTelemetry 분산 추적과 span

<https://opentelemetry.io/docs/concepts/signals/traces/>

[T35] OWASP 프롬프트 주입 방어 지침

https://cheatsheetseries.owasp.org/cheatsheets/LLM_Prompt_Injection_Prevention_Cheat_Sheet.html

[T36] OWASP CSRF 방어 지침

https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html

[T37] Google SRE의 SLO 경보와 소진율

<https://sre.google/workbook/alerting-on-slos/>

[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>

주제별 공식 문서

[R01] Google Cloud / 바이브 코딩의 개념

<https://cloud.google.com/discover/what-is-vibe-coding>

[R02] VS Code / AI와 함께 작업하기

<https://code.visualstudio.com/docs/agents/overview>

[R03] Lovable / 공식 소개

<https://docs.lovable.dev/introduction/welcome>

[R04] Replit / Agent 공식 안내

<https://docs.replit.com/features/agent/overview>

[R05] MDN / HTML로 내용 구조 만들기

https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Structuring_content

[R06] Playwright / 사용자 행동을 검증하는 기준

<https://playwright.dev/docs/best-practices>

[R07] MDN / textContent와 텍스트 표시

<https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent>

[R08] MDN / localStorage의 범위와 제약

<https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>

[R09] MDN / Fetch의 응답과 오류 처리

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

[R10] MDN / CORS와 출처 간 응답 공유

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS>

[R11] OWASP / 인증 설계 안내

https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html

[R12] OWASP / 권한 검사 안내

https://cheatsheetseries.owasp.org/cheatsheets/Authorization_Cheat_Sheet.html

[R13] W3C WAI / 입력 요소의 라벨

<https://www.w3.org/WAI/tutorials/forms/labels/>

[R14] Git / 변경 차이 확인

<https://git-scm.com/docs/git-diff>

[R15] GitHub / Pages의 배포 범위

<https://docs.github.com/en/pages/getting-started-with-github-pages/what-is-github-pages>

[R16] Vercel / 실행 환경과 첫 배포 규칙

<https://vercel.com/docs/deployments/environments>

[G06] PR 병합 방식

<https://docs.github.com/en/pull-requests/reference/pull-request-merges>

[G10] Git 명령어 공식 참조

<https://git-scm.com/docs>

[G11] git push와 force-with-lease

<https://git-scm.com/docs/git-push>

[G14] git restore

<https://git-scm.com/docs/git-restore>

[G15] git reset

<https://git-scm.com/docs/git-reset>

[G16] git revert

<https://git-scm.com/docs/git-revert>

[G17] git reflog

<https://git-scm.com/docs/git-reflog>

[G21] Pro Git의 객체 설명

<https://git-scm.com/book/en/v2/Git-Internals-Git-Objects>

[G22] Projects 개념

<https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects>

[G23] PR과 Issue 연결

<https://docs.github.com/en/issues/tracking-your-work-with-issues/using-issues/linking-a-pull-request-to-an-issue>

[G24] PR 리뷰 개념

<https://docs.github.com/en/pull-requests/reference/pull-request-reviews>

[G29] CODEOWNERS

<https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners>

[G35] Codespaces 사용량과 결제

<https://docs.github.com/en/billing/managing-billing-for-github-codespaces/about-billing-for-github-codespaces>

[G37] Actions Workflow 문법

<https://docs.github.com/en/actions/reference/workflows-and-actions/workflow-syntax>

[G38] actions/checkout 공식 저장소

<https://github.com/actions/checkout>

[G39] actions/setup-python 공식 저장소

<https://github.com/actions/setup-python>

[G40] 의존성 캐시

<https://docs.github.com/en/actions/reference/workflows-and-actions/dependency-caching>

[G41] Workflow Artifact 보관

<https://docs.github.com/en/actions/tutorials/store-and-share-data>

[G42] GITHUB_TOKEN 권한

<https://docs.github.com/en/actions/reference/authentication-in-a-workflow>

[G43] Actions 안전한 사용

<https://docs.github.com/en/actions/reference/security/secure-use>

[G44] Actions의 OIDC

<https://docs.github.com/en/actions/concepts/security/openid-connect>

[G45] 배포와 Environment 보호

<https://docs.github.com/en/actions/reference/workflows-and-actions/deployments-and-environments>

[G46] Dependabot 설정 참조

<https://docs.github.com/en/code-security/reference/supply-chain-security/dependabot-options-reference>

[G47] Code scanning 개념

<https://docs.github.com/en/code-security/concepts/code-scanning/code-scanning>

[G48] Secret scanning 개념

<https://docs.github.com/en/code-security/concepts/secret-security/secret-scanning>

[G60] 커밋 서명 검증

<https://docs.github.com/en/authentication/managing-commit-signature-verification/about-commit-signature-verification>

[G61] Artifact attestation

<https://docs.github.com/en/actions/how-tos/secure-your-work/use-artifact-attestations/use-artifact-attestations>

[G62] 저장소 백업

<https://docs.github.com/en/repositories/archiving-a-github-repository/backing-up-a-repository>

[G63] Actions 사용량과 결제

<https://docs.github.com/en/billing/managing-billing-for-your-products/managing-billing-for-github-actions/about-billing-for-github-actions>

[G65] git diff

<https://git-scm.com/docs/git-diff>

[G66] git log

<https://git-scm.com/docs/git-log>

[G68] git fetch

<https://git-scm.com/docs/git-fetch>

[G69] git merge

<https://git-scm.com/docs/git-merge>

[G70] git commit

<https://git-scm.com/docs/git-commit>

[G72] git status

<https://git-scm.com/docs/git-status>

[G78] 재사용 Workflow

<https://docs.github.com/en/actions/how-tos/reuse-automations/reuse-workflows>

[G83] git bundle

<https://git-scm.com/docs/git-bundle>

마지막 점검과 인계

본문에서 설명한 요구와 실행 및 검증의 경계를 실제 결과에 적용한다. 미확인 항목을 확인으로 바꾸려면 관찰 가능한 증거가 필요하다.

마지막 점검 1 내가 만든 것을 설명한다

화면을 닫고도 목적을 설명할 수 있는가

도구의 이름과 기능 목록보다 사용자의 행동을 먼저 말한다. 누가 어떤 상황에서 무엇을 입력하고 무엇을 얻는지 설명한다. 자료를 고치는 사람과 읽기만 하는 사람도 구분한다. 이 설명이 없다면 새 기능을 더 넣기 전에 문제 정의로 돌아간다.

남길 근거	예시
목적	교사가 제목과 과목으로 자료를 찾는다
입력	검색어와 과목
결과	조건에 맞는 제목과 개수
한계	로그인과 공용 저장은 미구현

설명이 막히면 돌아갈 위치

문제는 2장과 4장에서 다시 쓴다. 파일과 실행 위치는 3장과 기술 해설에서 확인한다. AI의 설명이 실제 파일과 맞는지 읽고 예시 입력 한 개를 끝까지 따라간다.

검토 질문 사용자가 다르면 같은 완료 조건을 그대로 써도 될까? **해설** 역할과 자료 및 사용 환경이 달라지면 완료 조건도 다시 판단한다.

마지막 점검 2 정상과 실패를 재현한다

검사한 범위만 완료로 남긴다

정상 입력 하나와 경계 입력 및 실패 입력을 정한다. 같은 조건에서 다시 실행하고 기대와 실재를 비교한다. 핵심 함수 검사와 브라우저 조작 및 실제 서버 검사는 범위가 다르므로 별도 기록으로 남긴다.

시험	기대	실제 상태
검색과 과목	두 조건의 교집합	통과와 실패 또는 미실행
초기화	전체 자료와 선택 보존	실제 결과 기록
가져오기 실패	기존 자료 유지	실제 결과 기록

실패하면 어떻게 할까

재현 절차와 마지막 정상 판본을 확인한다. 오류가 난 층을 구분하고 한 가지 수정 뒤 같은 검사를 수행한다. 예상 결과를 코드의 현재 출력에 맞춰 조용히 바꾸지 않는다.

검토 질문 자동 검사만 통과했다면 배포 주소에서도 같다고 말할 수 있을까? **해설** 실제 적용 판본과 실행 환경을 별도로 확인해야 한다.

마지막 점검 3 실제 사용 환경에서 읽는다

만드는 사람의 화면 밖을 확인한다

작은 화면과 글자 확대 및 키보드 조작으로 대표 행동을 반복한다. 입력의 이름과 현재 상태 및 실패 안내를 읽을 수 있어야 한다. 처음 쓰는 사람에게 목표만 알려 주고 어디에서 멈추는지 관찰한다.

관찰	남길 근거
키보드	이동 순서와 실행 결과
작은 화면	사용한 너비와 넘침 여부
글자 확대	버튼 이름과 본문의 읽기
첫 사용자	멈춘 위치와 재질문

수정 뒤 확인

색을 바꾸어 해결한 문제가 실제 이름과 상태의 문제였는지 다시 본다. 자동 검사에서 확인한 항목과 사람이 조작해 확인한 항목을 나누어 기록한다.

검토 질문 사용자가 사용법을 틀렸다면 안내만 늘리면 될까? **해설** 화면의 선택과 이름 및 흐름 자체가 오해를 유도했는지도 관찰한다.

마지막 점검 4 자료와 권한을 확인한다

보호한다고 말한 범위를 실제 요청으로 시험한다

화면에서 숨긴 기능과 서버가 거절하는 기능은 다르다. 공용 자료와 계정이 있는 경우 다른 계정과 직접 요청의 결과를 확인한다. 이 책의 판단 모형만 실행했다면 실제 서비스의 인증과 권한 검사는 아직 남은 일이다.

자원	확인할 범위
개인 메모	소유자와 읽기 및 수정
관리 자료	역할과 담당 범위
외부 서비스	키 보관과 사용 권한
백업	포함 자료와 접근 주체

구현하지 않은 것은 남은 일로 표시

실제 계정과 공용 저장기 없는 정적 예제에는 적용하지 않는 항목도 있다. 적용 여부와 이유를 기록한다. 모든 칸에 완료를 표시하는 것보다 현재 범위를 정확히 설명하는 것이 목적이다.

검토 질문 권한 함수를 시험했으니 실제 API도 보호된 것일까? **해설** 검증된 신원과 요청 경로 및 저장 처리에 그 규칙이 적용되는지 실제 서버에서 확인해야 한다.

마지막 점검 5 배포와 비용과 복구를 확인한다

사용자에게 전달할 주소에서 확인한다

로컬 성공과 배포 성공 및 운영 반영을 구분한다. 적용 판본과 실제 주소 및 연결 자원을 적는다. 기본 주소에서 작동한 뒤 도메인 연결을 확인한다. 미리보기라는 이름만으로 운영 자료와 분리됐다고 생각하지 않는다.

운영 근거	남길 내용
적용	판본과 주소 및 확인 결과
사용량	실행과 저장 및 외부 요청
장애	관련 로그와 필요한 식별 정보
복구	되돌릴 상태와 자료 처리

복구가 실패하면

단순히 사본을 더 만드는 일로 끝내지 않는다. 필요한 설정과 접근 및 자료가 무엇인지 찾는다. 코드 복구와 데이터 복구의 관계를 확인하고 새로운 입력을 잃지 않는 절차를 검토한다.

검토 질문 이전 코드로 돌아가면 데이터도 자동 복구될까? **해설** 자료 구조와 저장소는 별도로 확인한다. 코드만 되돌렸을 때의 호환성도 시험한다.

마지막 점검 6 결과를 인계한다

마지막 산출물은 설명할 수 있는 결과다

다른 사람이 실행 안내로 도구를 열고 작은 수정을 할 수 있는지 확인한다. 실제로 검증한 범위와 알려진 한계 및 다음 작업을 전달한다. 확인란은 수행 기록이며 품질 전체를 자동 보증하는 점수가 아니다.

최종 기록	직접 적을 내용
프로젝트와 판본	이름과 적용한 상태
확인한 핵심 행동	입력과 결과 및 실행 환경
남은 일	미구현과 미확인 및 담당
다음 작은 변경	이유와 기대 결과

다시 시작할 때의 질문

다음 기능은 어떤 사용자의 문제를 줄이는가. 무엇을 보존해야 하는가. 어떤 입력에서 어떤 결과가 나와야 하는가. 실패했을 때 어디로 돌아갈 것인가. 이 질문을 다시 적을 수 있다면 시와 함께 다음 변경을 시작할 준비가 된 것이다.