M07-02 · GIBALJA VISUAL MANUAL

작업을 작게 나누고 완료 기준 쓰기

> **한 문장 목표:** `큰 요청 → 원하는 outcome → 한 decision → 검토 가능한 slice → acceptance → exact check → evidence` 로 연결해, AI가 추측하지 않고 사람은 결과를 직접 판정할 수 있는 작업 명세를 만듭니다.

난이도	개념	실습	셀프 테스트	최종 산출물
Level 1	60분	60분	15분	작업 분할 지도, 작업 명세서, 완료 기준·증거 매트릭스, readiness checklist

좋은 작업은 “작아서 빨리 끝나는 일”이 아닙니다. **한 가지 결정을 다루고, 다른 작업과 독립적으로 검증할 수 있으며, 끝난 뒤 repository가 계속 작동하고, 문제가 생기면 그 범위만 되돌릴 수 있는 일**입니다. 줄 수나 파일 수는 힌트일 뿐 판정 기준이 아닙니다.

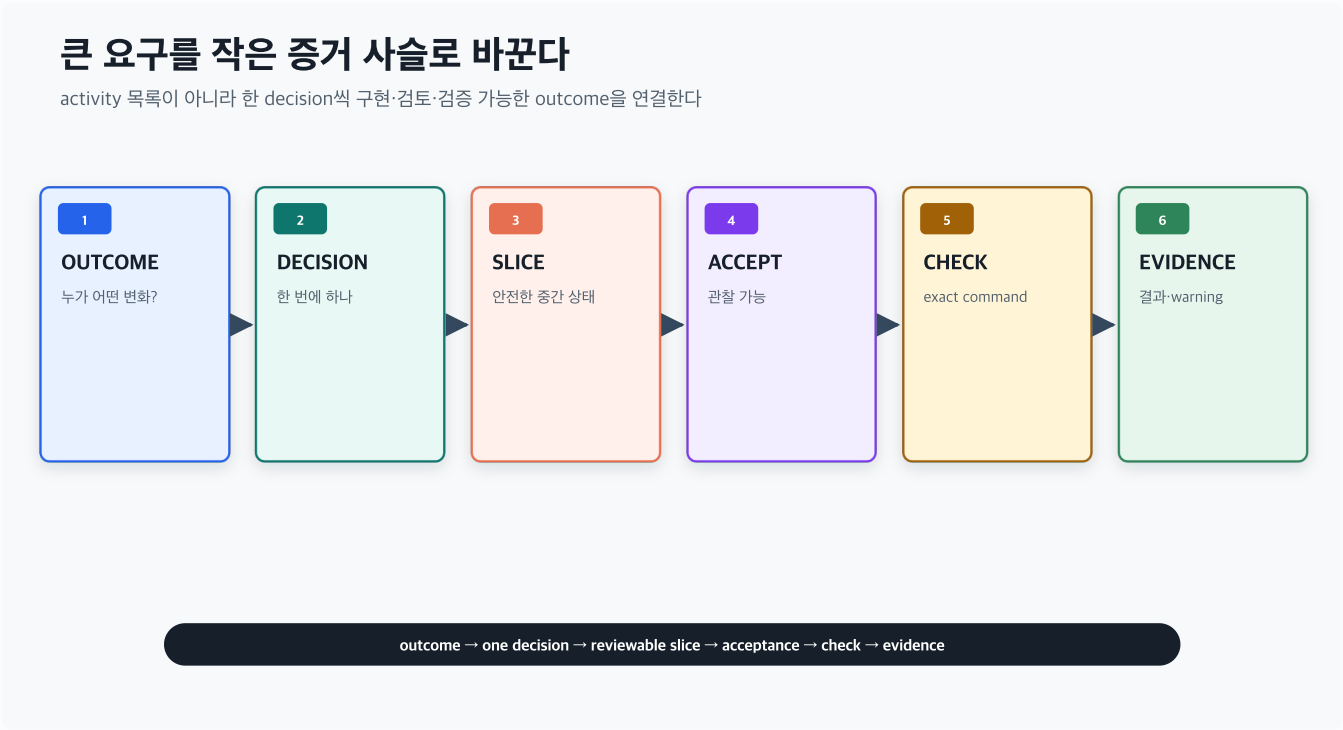
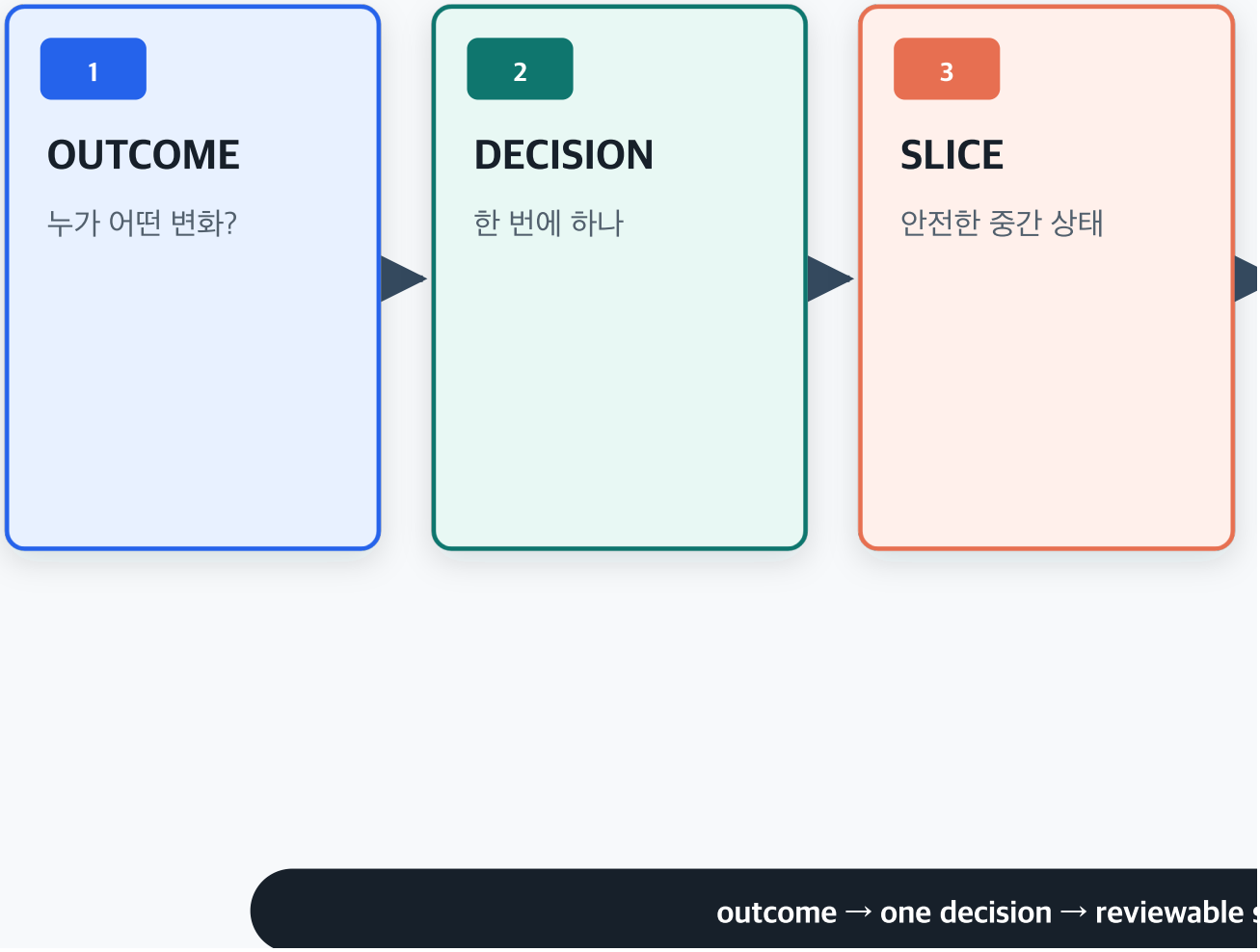
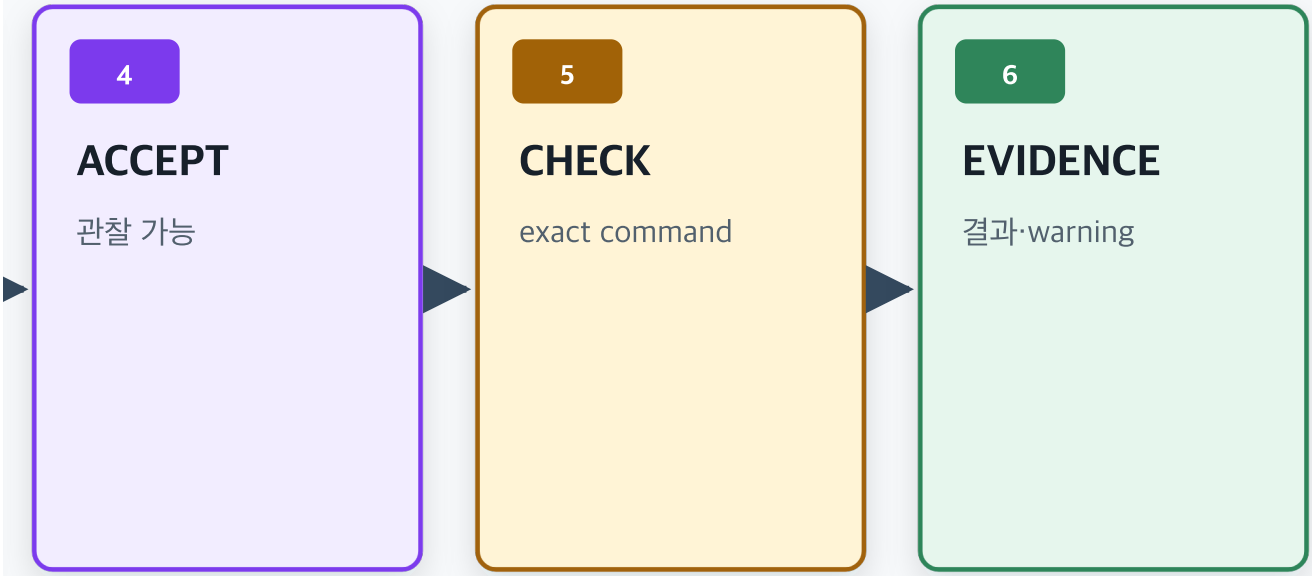


그림 1. 작업 명세는 요청을 구현 문장으로 바꾸는 문서가 아니라, 결과와 판정 증거를 끊김 없이 연결하는 계약입니다.


1
OUTCOME
누가 어떤 변화?
2
DECISION
한 번에 하나
3
SLICE
안전한 중간 상태
outcome → one decision → reviewable



slice → acceptance → check → evidence

0. 이 PDF를 공부하는 방법

1회차 · 그림만 읽기 · 20분

그림 1부터 그림 14까지 제목, 화살표, 결론 띠만 읽습니다. 다음 여덟 문장을 소리 내어 말할 수 있으면 됩니다.

```
작은 작업은 한 가지 결정을 다룬다.
크기는 줄 수가 아니라 검토 질문의 수로 판단한다.
각 작업 뒤 repository는 WORKING 상태여야 한다.
의존성은 암묵적인 순서가 아니라 DAG로 드러낸다.
acceptance는 positive·negative·boundary를 함께 쓴다.
완료 기준은 정확한 check와 evidence에 연결한다.
Ready는 시작 가능, Done은 증거로 완료됨을 뜻한다.
권한·data·외부 효과가 불명확하면 ASK 또는 BLOCK한다.
```

2회차 · 실습 저장소 만들기 · 10분

작업 명세 실습 생성기를 실행합니다.

```
./02_Labs/G07_AI_Spec/L07-02_create-task-spec-practice.sh
```

생성기는 외부 package와 network 없이 작은 Node.js repository, 큰 요구 문서, 다섯 개의 작업 명세, readiness audit와 evidence folder를 만듭니다. 같은 target이 이미 있으면 exit code 2로 멈추며 덮어쓰지 않습니다.

3회차 · 12개 작업 장면 판독 · 35분

작업 분할 판정 데스크를 엽니다. 각 장면에서 해설을 보기 전에 다음 일곱 값을 말합니다.

```
원하는 outcome =
이번 task의 decision =
in scope / out of scope =
positive / negative / boundary =
exact check와 evidence =
permission·stop condition =
판정 = READY / NOT_READY / BLOCKED
```

4회차 · 내 프로젝트에 적용 · 70분

AI 작업 명세 양식을 채우고, 단계별 실습서에 따라 audit합니다. 낯선 단어는 작업 분할·완료 기준 용어집에서 찾습니다.

1. 큰 요청을 그대로 AI에게 주면 왜 위험한가

다음 요청은 한 문장처럼 보이지만 실제로는 서로 다른 결정을 섞고 있습니다.

```
검토가 필요한 학습자를 찾아 이유를 보여 주고,
대시보드에서 필터링하고 CSV로 내려받고,
자동 이메일과 분석 이벤트까지 붙인 뒤 배포해 줘.
기왕이면 오래된 모듈 이름도 바꿔 줘.
```

이 안에는 최소한 다음 질문이 숨어 있습니다.

숨은 결정	필요한 owner·evidence	잘못 섞었을 때의 위험
검토 상태의 이유 코드는 누가 소유하는가	policy owner·contract test	UI와 API가 이유를 따로 계산
queue는 어떤 data를 읽는가	API·data owner	실제 개인정보를 임의 사용
filter의 contract는 무엇인가	product·UI owner	빈 상태·경계값 누락
CSV에 어떤 field를 내보내는가	data owner·approval	과도한 정보 노출
이메일 전송 조건은 무엇인가	security·procurement·consent	동의 없는 외부 발송
analytics event는 무엇인가	analytics governance	의미 없는 event와 민감 data
rename은 호환성을 깨뜨리는가	maintainer·reference audit	행동 변경과 구조 변경 혼합
어디에 배포하는가	operations owner	환경·승인 없는 side effect

AI가 이 요청을 한 번에 받으면 빈칸을 추정하거나, 보이는 부분만 구현하거나, 많은 변경을 만든 뒤 “완료”라고 말할 수 있습니다. 문제는 AI의 문장력이 아니라 **작업 계약에 판정 경계가 없다는 것**입니다.

1.1. 활동 목록과 결과 계약을 구분합니다

```
활동:
  API 만들기, 화면 만들기, 테스트 추가하기

결과:
  운영자가 synthetic case의 검토 이유를
  두 번째 policy 구현 없이 식별한다.
```

활동은 수단입니다. outcome은 actor가 얻는 관찰 가능한 변화입니다. 같은 outcome을 더 작은 변경으로 달성할 수 있다면 활동 목록은 바뀔 수 있습니다.

1.2. 작업 명세의 품질 식

TASK QUALITY

한 decision × 독립 검증 × working state × reversible scope × visible evidence

한 요소가 0이면 파일이 적어도 좋은 작업이 아닙니다.

30초 확인 1

“대시보드를 개선한다”는 outcome입니까? 누가, 언제, 무엇을 관찰하면 개선됐다고 말할 수 있는지 다시 써 봅니다.

2. 작은 작업의 기준은 줄 수가 아닙니다

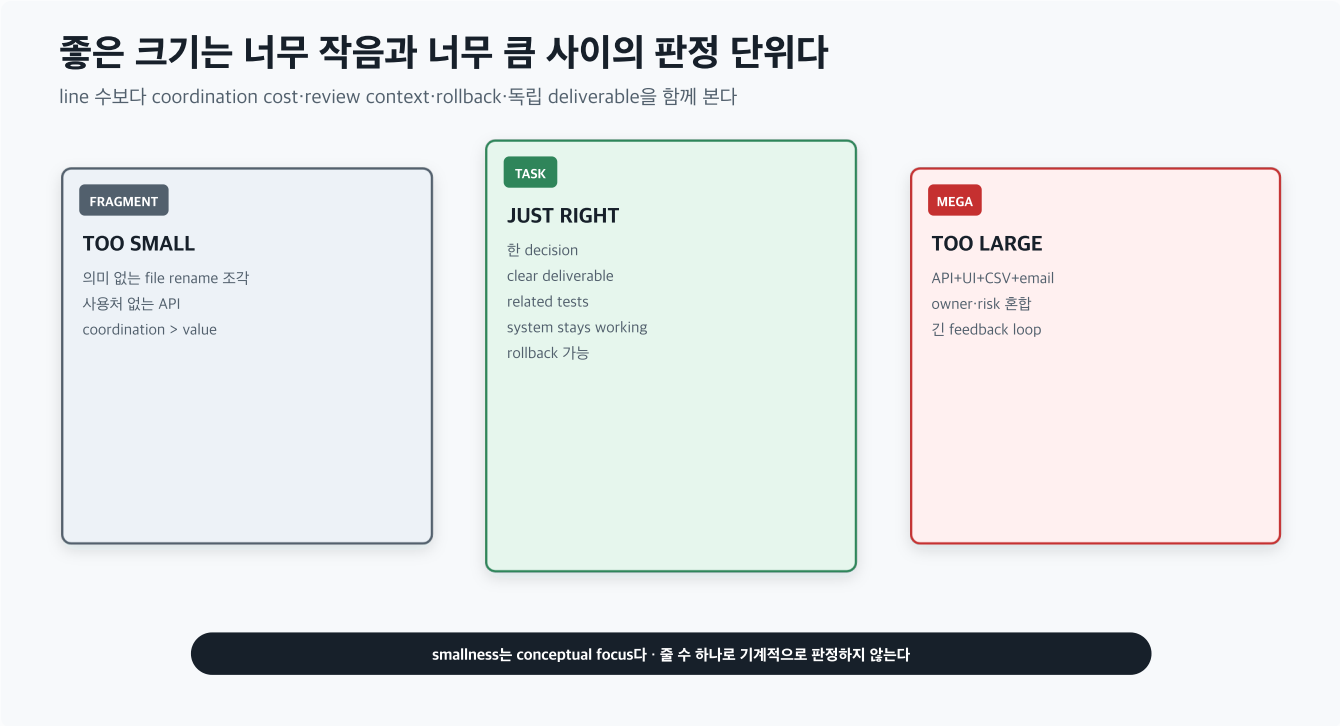


그림 2. 적정 크기는 구현량보다 한 사람이 한 번의 검토에서 이해하고 판정할 수 있는지로 결정합니다.

FRAGMENT

TOO SMALL

의미 없는 file rename 조각

사용처 없는 API

coordination > value

TASK

JUST RIGHT

한 decision

clear deliverable

related tests

system stays working

rollback 가능

smallness는 conceptual focus다 · 줄

MEGA

TOO LARGE

API+UI+CSV+email

owner·risk 혼합

긴 feedback loop

수 하나로 기계적으로 판정하지 않는다

2.1. 너무 작은 작업

```
T1 변수 이름 한 개 만들기
T2 import 한 줄 추가하기
T3 테스트 파일 이름 만들기
```

이 작업들은 각각 독립적인 사용자·시스템 outcome이 없고, 단독으로 완료해도 repository에 의미 있는 working state를 만들지 못합니다. 관리 overhead만 늘어납니다.

2.2. 너무 큰 작업

```
API·UI·CSV·email·analytics·refactor·deploy를 모두 구현한다.
```

검토 질문, owner, permission, failure mode와 rollback이 너무 많습니다. 하나가 실패하면 무엇이 완료됐는지 판정하기 어렵습니다.

2.3. 알맞은 작업

```
T01
outcome:
  API consumer가 synthetic case가 왜 manual review인지 식별한다.
decision:
  policy가 stable reviewReason code를 소유하고 HTTP는 pass-through한다.
```

policy와 HTTP, test 세 surface를 바꾸더라도 결정은 하나입니다. 반대로 파일 한 개만 바꿔도 인증·개인정보·배포 결정을 동시에 한다면 큰 작업입니다.

2.4. 빠른 크기 판정 질문

질문	YES면 좋은 신호	NO면 할 일
한 문장으로 decision을 말할 수 있는가	검토 초점이 하나	decision별 분리
독립 acceptance가 있는가	결과를 단독 판정	outcome 재작성
exact check로 검증 가능한가	주장 대신 evidence	check path 확보

질문	YES면 좋은 신호	NO면 할 일
끝난 뒤 WORKING인가	안전한 중간점	slice 재배치
그 범위만 rollback 가능한가	실패 격리	coupling 축소
owner가 한 명 또는 한 팀인가	결정 권한 명확	owner boundary 분리

3. 한 작업에는 한 가지 decision을 둡니다

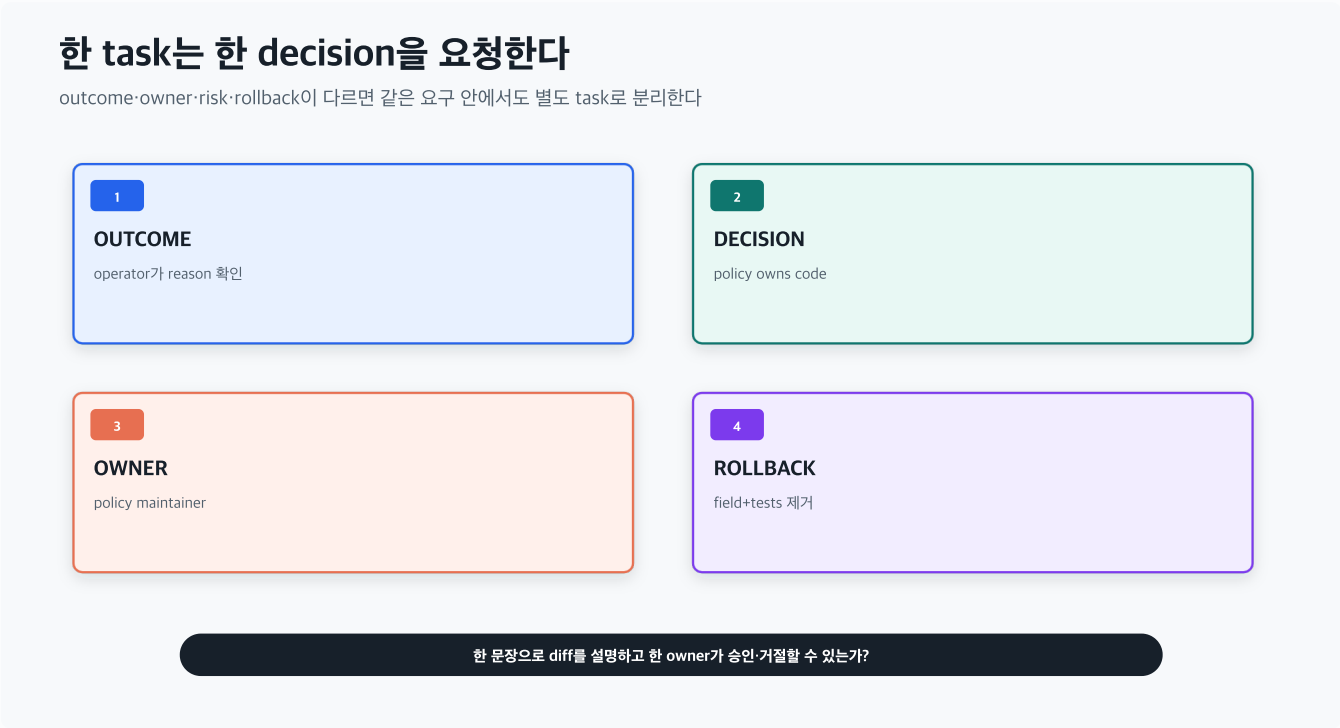


그림 3. 작업의 중심은 파일 목록이 아니라 한 가지 decision입니다. 나머지 블록은 그 결정을 검토하기 위한 경계입니다.

1

OUTCOME

operator가 reason 확인

3

OWNER

policy maintainer

한 문장으로 diff를 설명하고 한 c

2

DECISION

policy owns code

4

ROLLBACK

field+tests 제거

wner가 승인·거절할 수 있는가?

3.1. decision은 구현 동사가 아닙니다

나쁜 예:

```
API 파일을 수정한다.
테스트를 추가한다.
UI를 업데이트한다.
```

좋은 예:

```
Policy owns one stable reviewReason code,
and HTTP passes it through without recomputing it.
```

좋은 decision은 **누가 무엇을 소유하고 어떤 경계를 지키는지**를 드러냅니다. 그래서 reviewer가 "이 책임 배치에 동의하는가?"라는 한 질문에 집중할 수 있습니다.

3.2. decision count가 2 이상이면 분리를 검토합니다

```
Decision A: policy가 review reason을 소유한다.
Decision B: CSV에는 learner email을 포함한다.
Decision C: email provider로 Vendor X를 쓴다.
```

세 decision은 owner와 위험, 검증 방식이 다릅니다. 한 작업에 넣으면 review reason의 안전한 local change가 data export와 외부 발송 승인에 묶입니다.

3.3. 함께 둬도 되는 변경

같은 decision을 완성하기 위해 policy, adapter, test가 함께 바뀌는 것은 자연스럽습니다.

```
policy output 변경
→ HTTP pass-through
→ positive·negative·boundary tests
```

파일 수가 세 개여도 하나의 public behavior contract를 세로로 완성합니다.

3.4. decision sentence 양식

```
[owner component]가 [stable contract]를 소유하고,
[consumer component]는 [허용된 방식]으로만 사용한다.
```

또는:

```
이번 작업에서 결정할 것은 ______ 하나이며,
______와 ______ 결정은 별도 task로 남긴다.
```

30초 확인 2
“reviewReason을 추가하고 CSV로 내보낸다”에는 decision이 몇 개입니까? contract 소유와 data export 승인을 별도로 세어 봅니다.

4. 수평 분할과 수직 분할을 구분합니다

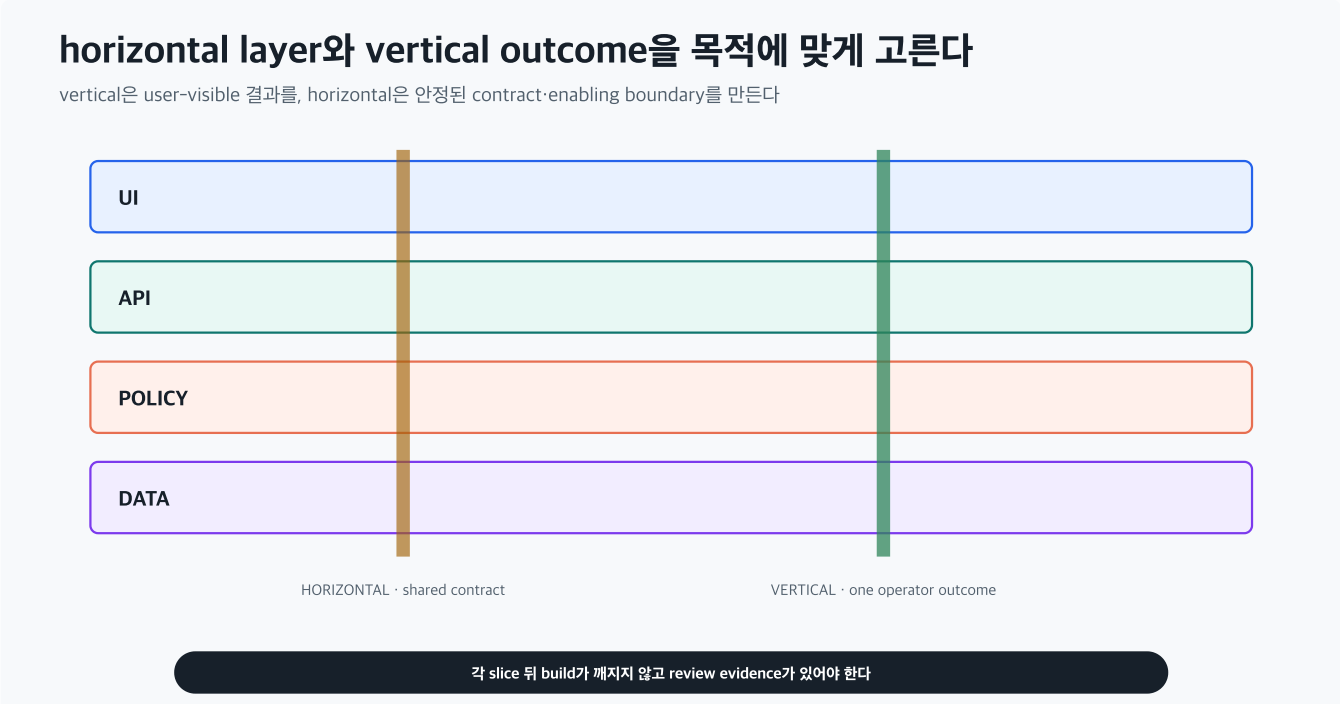
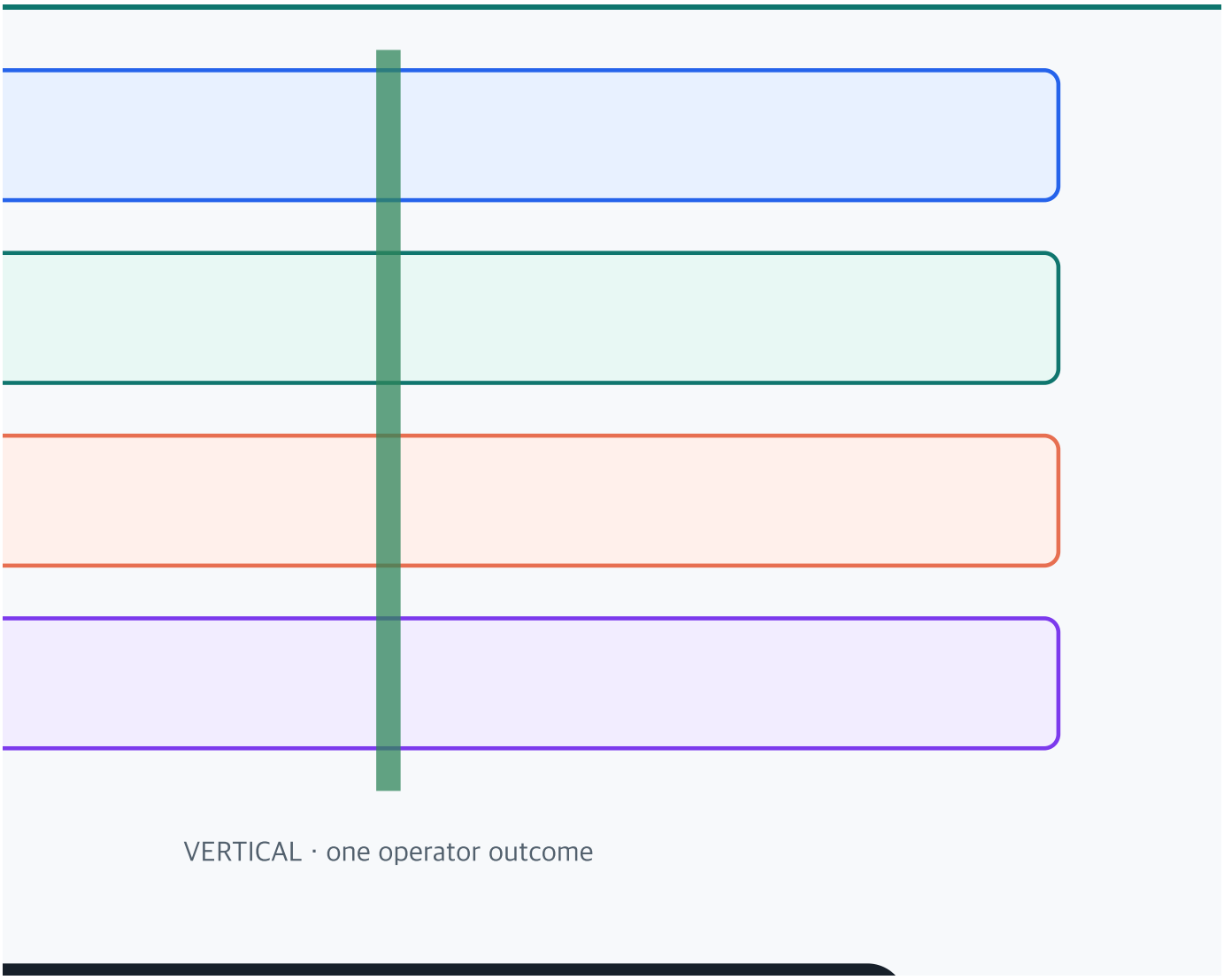


그림 4. 수직 slice는 작더라도 끝에서 관찰 가능한 결과를 만듭니다. 수평 slice는 contract·기반 작업일 때 목적을 명시합니다.

HORIZONTAL · shared contract


VERTICAL · one operator outcome

4.1. 수평 분할

```
DB schema 전부
→ API 전부
→ UI 전부
→ test 전부
```

각 단계가 오래 진행돼도 actor는 결과를 볼 수 없습니다. 마지막 단계까지 연결 문제를 발견하지 못할 수 있습니다.

4.2. 수직 분할

```
review reason 한 개를 policy→HTTP→test로 통과
→ synthetic queue 한 개를 API→empty state→test로 통과
→ 화면에 reason 표시→loading·empty·error state 검증
→ reason filter 추가→boundary test
```

각 작업이 작지만 end-to-end로 관찰 가능한 결과를 만듭니다.

4.3. 수평 작업이 필요한 경우

모든 수평 작업이 나쁜 것은 아닙니다. 다음에는 enabling task가 필요할 수 있습니다.

상황	좋은 수평 작업의 조건
migration 준비	production 실행 없이 schema contract와 rollback 검증
shared interface	소비자 하나로 contract를 입증
test harness	뒤 작업의 실패를 검출하는 최소 fixture 제공
security control	별도 owner·threat·enforcement evidence 존재

수평 작업은 “나중에 필요할 것 같아서”가 아니라 다음 task의 risk를 실제로 줄이고 독립 evidence를 만들 때 사용합니다.

5. 네 가지 slice 유형을 선택합니다

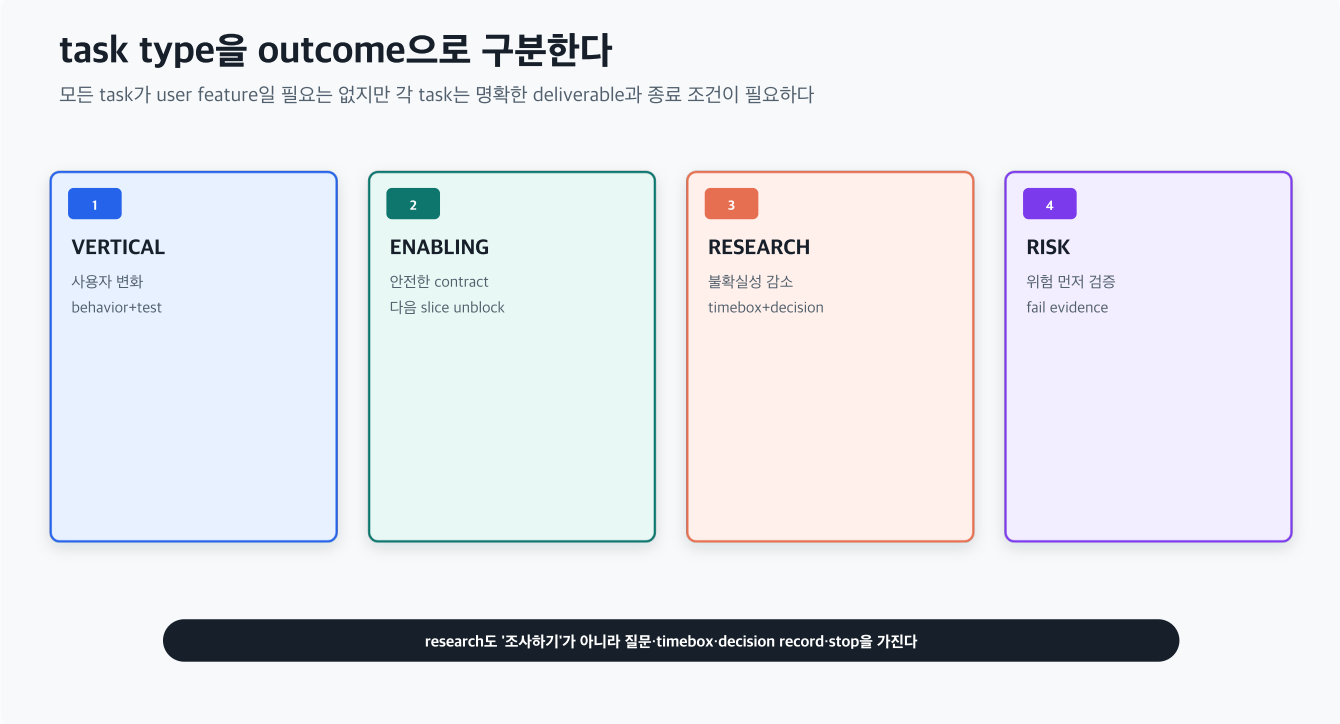


그림 5. 모든 불확실성을 구현 task로 해결하지 않습니다. 무엇을 배우고 어떤 evidence를 남길지에 따라 slice 유형을 고릅니다.

1

VERTICAL

사용자 변화

behavior+test

2

ENABLING

안전한 contract

다음 slice unblock

research도 '조사하기'가 아니라 질문·tir

3

RESEARCH

불확실성 감소

timebox+decision

4

RISK

위험 먼저 검증

fail evidence

nebox·decision record·stop을 가진다

5.1. Vertical slice

사용자나 consumer가 관찰 가능한 작은 행동을 완성합니다.

```
예: synthetic REVIEW_REQUIRED case에 stable reason이 보인다.
evidence: contract test + formatted output
```

5.2. Enabling slice

다음 vertical task가 안전하게 진행되도록 contract·harness·boundary를 준비합니다.

```
예: email 발송 없이 notification port와 fake adapter 계약만 검증한다.
evidence: interface contract + fake test
```

5.3. Timeboxed research slice

정답을 구현하지 않고 선택에 필요한 불확실성을 줄입니다.

```
질문: CSV에 필요한 field와 승인 owner는 누구인가?
시간: 45분
산출물: 후보 3개, evidence, 미해결 질문, 추천과 비추천
금지: 실제 개인정보 export
```

research task도 “조사하기”로 끝내지 않습니다. 질문, 시간 상한, source 범위, 의사결정 산출물과 stop condition을 씁니다.

5.4. Risk-first slice

가장 위험하거나 실패 가능성이 큰 가정을 작은 실험으로 먼저 검증합니다.

```
예: production email 전에 synthetic recipient만 허용하는 dry-run과
approval gate가 실제로 발송을 차단하는지 검증한다.
```

5.5. 선택표

지금 필요한 것	추천 slice	완료 evidence
작은 사용자 결과	vertical	observable behavior

지금 필요한 것	추천 slice	완료 evidence
다음 작업의 안전한 기반	enabling	contract·harness
owner·규칙·기술 가능성 확인	research	decision memo
큰 위험 가정 먼저 제거	risk-first	bounded experiment

6. 작업 사이의 의존성을 DAG로 그립니다

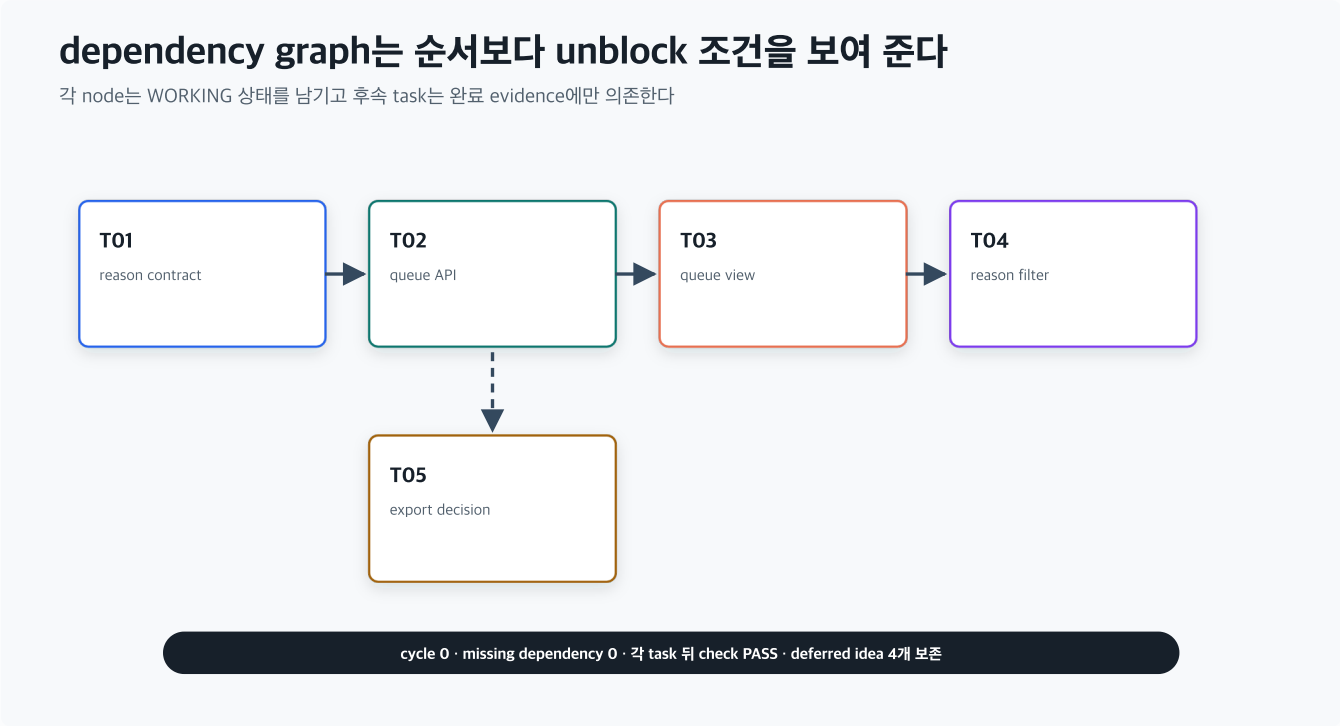
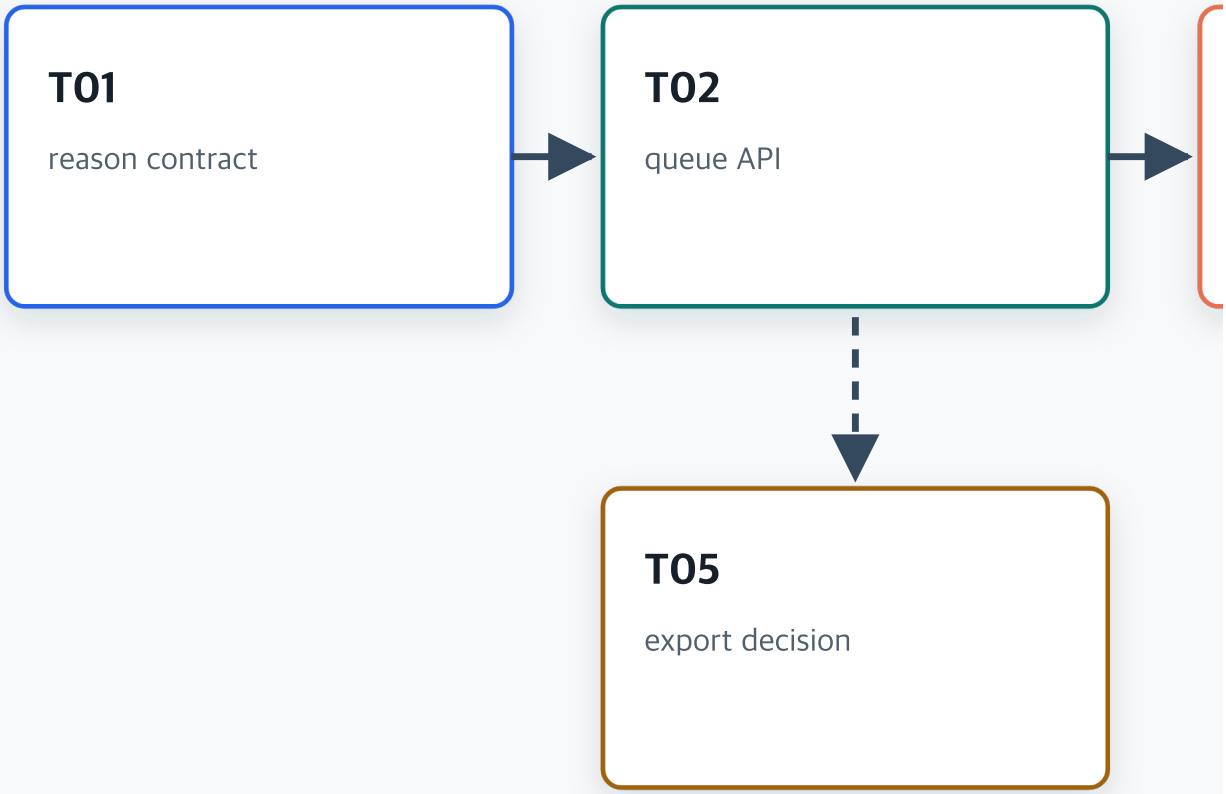


그림 6. 순서는 번호가 아니라 필요한 contract의 방향으로 정합니다. deferred는 삭제가 아니라 이유를 가진 보존입니다.



cycle 0 · missing dependency 0 · 각 tas

k 뒤 check PASS · deferred idea 4개 보존

6.1. 실습의 안전한 작업 지도

```
T01 stable review reason
 └ T02 synthetic review queue API
     ├ T03 operator queue view
     │   └ T04 reason filter
     └ T05 CSV export decision research

deferred:
  email / analytics / refactor / deploy
```

T05는 CSV 구현이 아니라 data owner와 최소 field를 결정하는 research task입니다. T02의 queue contract 없이는 export 대상도 정의할 수 없으므로 T02에 의존합니다.

6.2. dependency는 파일 선후가 아닙니다

```
좋은 dependency:
  T02는 T01의 reviewReason contract를 소비한다.

나쁜 dependency:
  T02는 번호가 뒤이므로 T01 다음이다.
```

6.3. 각 node 뒤에는 WORKING state가 있어야 합니다

작업이 끝날 때마다 다음이 참이어야 합니다.

```
build/check/test가 통과한다.
기존 public behavior가 의도 없이 깨지지 않는다.
미완성 기능이 production path에 노출되지 않는다.
다음 task가 없어도 현재 변경을 설명하고 되돌릴 수 있다.
```

6.4. DAG audit

gate	BLOCK 조건
missing dependency	존재하지 않는 task를 참조
cycle	T01→T02→T01 같은 순환
broken intermediate	중간 node가 WORKING이 아님

gate	BLOCK 조건
invisible deferred	요청 일부가 이유 없이 사라짐
unsafe parallel	같은 contract·file을 동시에 소유

주의
"나중에 하자"는 out of scope가 아닙니다. 무엇을 왜 미뤘는지, 다시 여는 trigger와 owner가 무엇인지 남겨야 요구가 조용히 사라지지 않습니다.

7. In scope와 Out of scope를 함께 씁니다

7.1. In scope만 쓰면 생기는 문제

```
in scope: review reason 추가
```

AI는 자연스럽게 queue, UI, export까지 "도움이 될 것"이라 추정할 수 있습니다. 범위는 포함 목록만으로 닫히지 않습니다.

7.2. T01의 경계

```
in scope:
  REVIEW_REQUIRED policy output의 reviewReason
  HTTP pass-through
  positive·negative·boundary tests

out of scope:
  dashboard queue
  CSV export
  email·analytics·persistence·deployment
```

7.3. Out of scope의 세 가지 상태

상태	뜻	기록 예
deferred	후속 decision 필요	CSV: T05 research 뒤 재판정

상태	뜻	기록 예
prohibited	현재 하면 안 됨	real learner data 사용 금지
unrelated	outcome과 무관	progress module rename

7.4. 범위가 좋은지 확인하는 질문

1. 이 항목은 outcome을 직접 달성하는가?
2. 이번 decision과 같은 owner가 검토하는가?
3. 같은 check와 rollback으로 판정할 수 있는가?
4. 제외하면 repository가 여전히 WORKING인가?
5. 제외된 요구에 이유·owner·trigger가 있는가?

8. Acceptance는 세 방향으로 씁니다

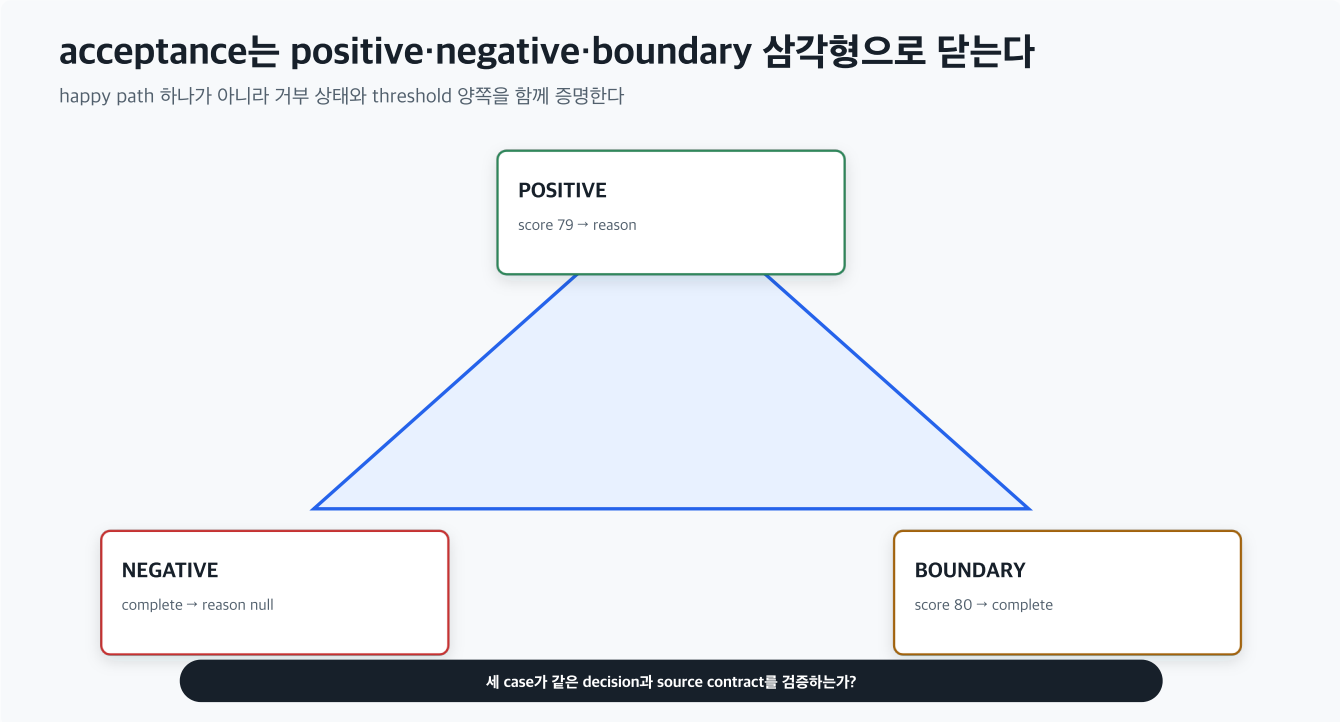
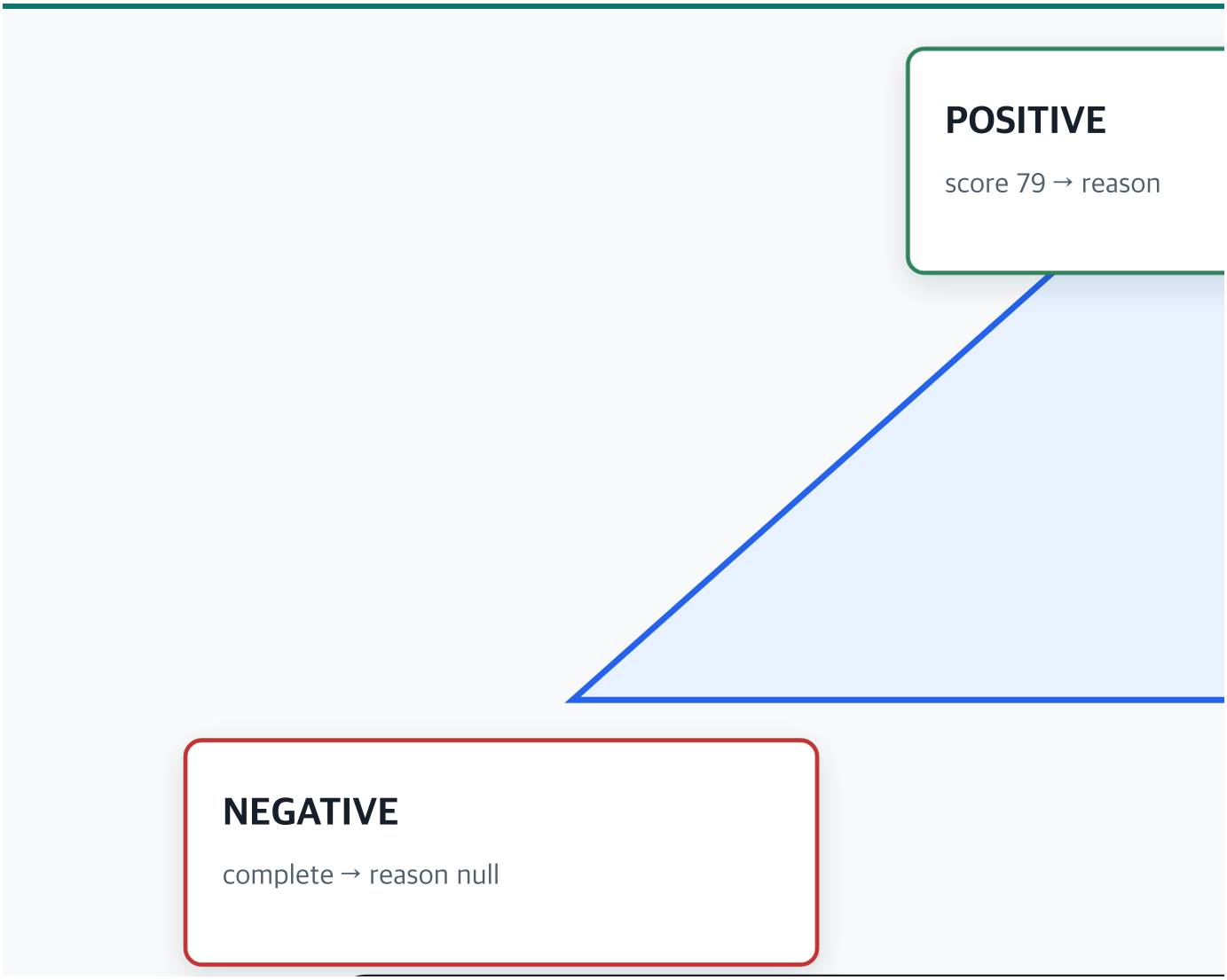
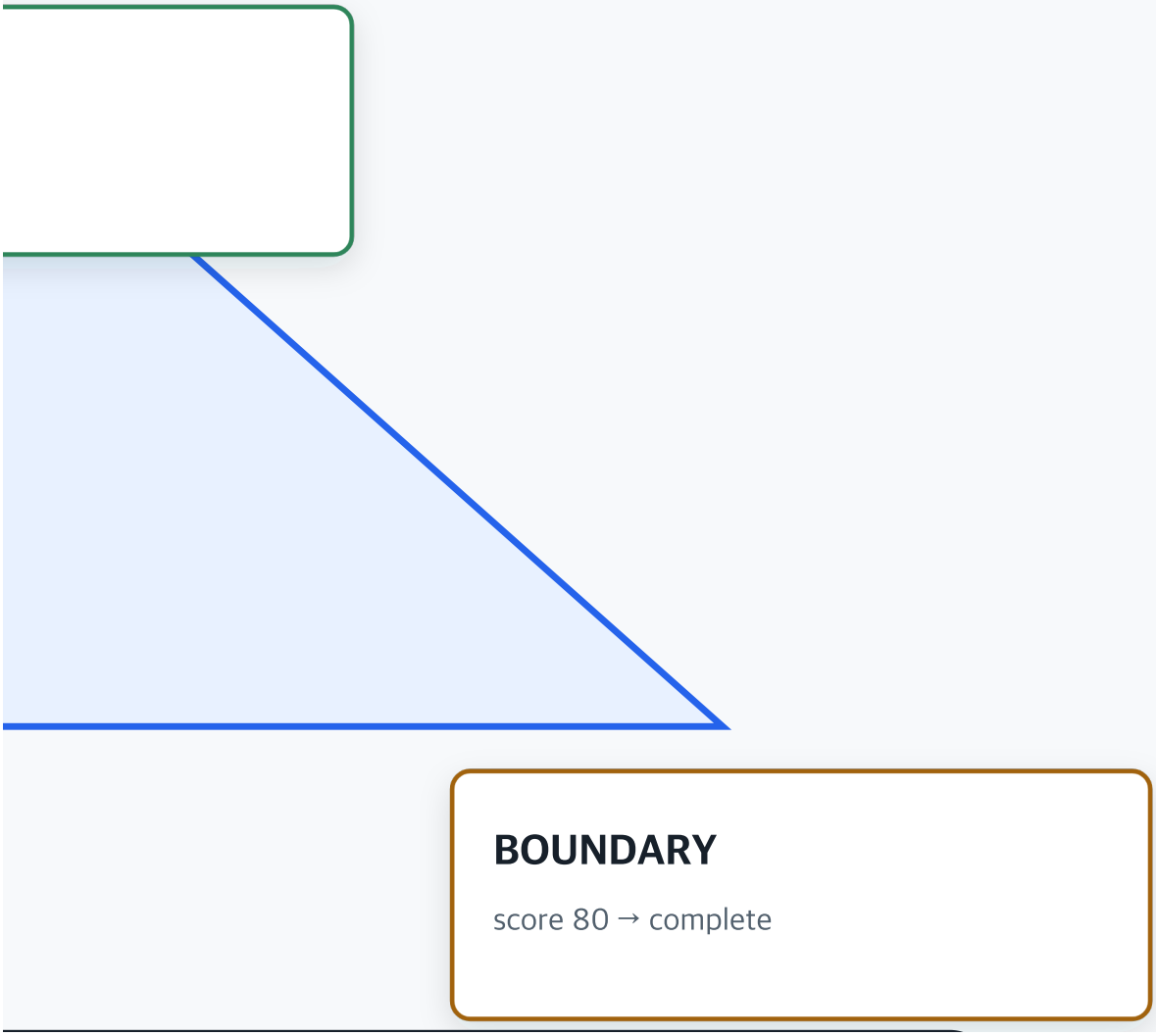


그림 7. 성공 예 하나만으로는 contract가 닫히지 않습니다. 하지 않아야 할 행동과 경계값을 함께 씁니다.


POSITIVE
score 79 → reason
NEGATIVE
complete → reason null


BOUNDARY
score 80 → complete

8.1. Positive

의도한 조건에서 원하는 결과가 나오는지 확인합니다.

```
Given progress 100 and score 79
When policy decides review state
Then state is REVIEW_REQUIRED
And reviewReason is SCORE_BELOW_THRESHOLD
```

8.2. Negative

하지 않아야 할 행동을 확인합니다.

```
Given state is COMPLETE or IN_PROGRESS
When result is formatted
Then reviewReason is null
And HTTP does not recompute it
```

negative는 단순 error case가 아닙니다. 권한 없는 사용자, 비대상 상태, 중복 실행, 예상하지 않은 side effect도 포함합니다.

8.3. Boundary

규칙이 바뀌는 정확한 가장자리를 확인합니다.

```
Given progress 100 and score exactly 80
When policy decides review state
Then state remains COMPLETE
And reviewReason is null
```

`79`, `80`, `81` 중 어느 값이 경계인지 명시하지 않으면 구현과 검토가 서로 다른 가정을 할 수 있습니다.

8.4. acceptance가 관찰 가능한지 확인합니다

나쁜 예:

```
코드가 깔끔하다.
성능이 좋아진다.
문제가 없어야 한다.
사용하기 편하다.
```

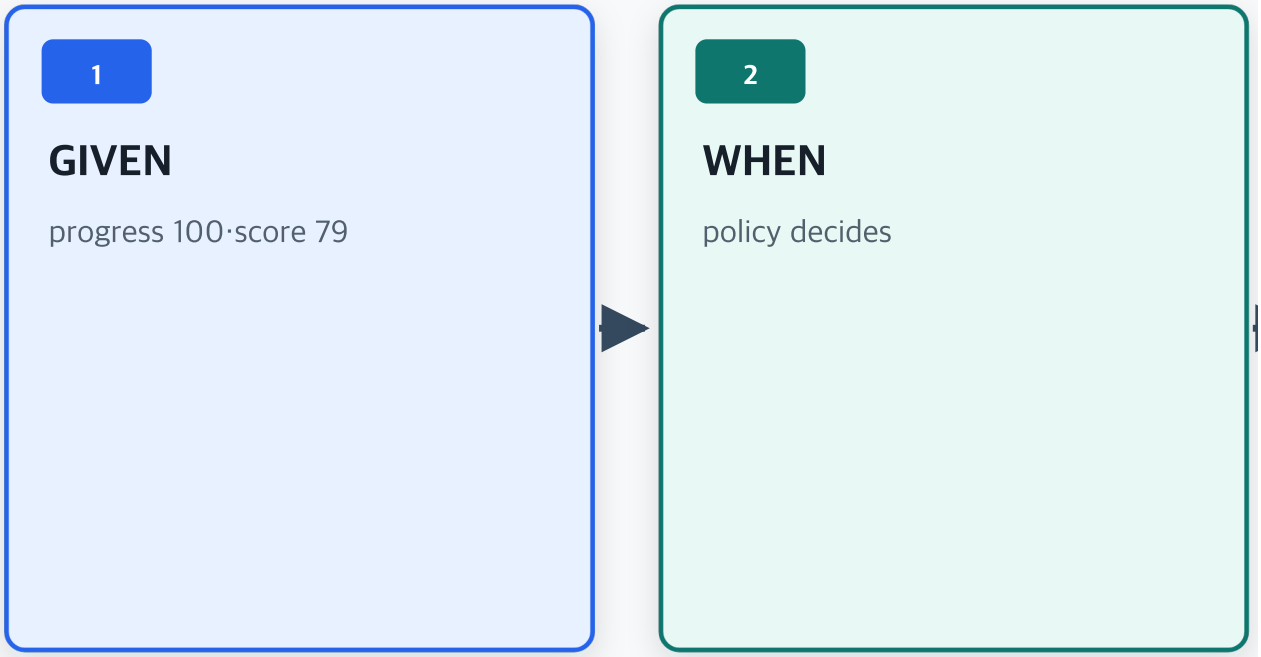
개선 예:

```
focused test가 reviewReason contract를 검증한다.
synthetic 100-case benchmark의 p95가 baseline 40ms 이하를 유지한다.
empty list에서 200과 []를 반환한다.
operator가 reason filter 선택 후 해당 reason만 본다.
```

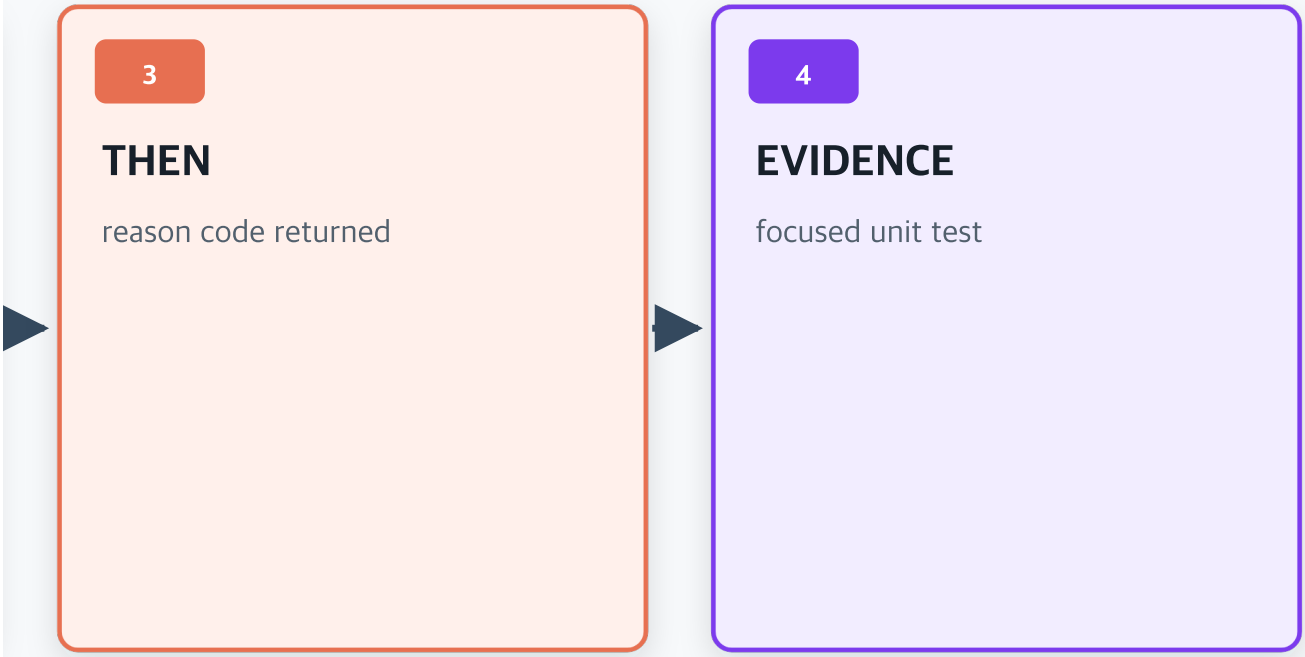
"좋다"를 없애는 것이 목적이 아닙니다. 누가 어떤 상태에서 무엇을 관찰해 PASS라고 말하는지 바꿔 씁니다.

9. Given·When·Then을 Evidence까지 연결합니다

그림 8. acceptance 문장만 있으면 아직 주장입니다. 어떤 command와 artifact가 결과를 입증하는지 연결해야 합니다.



‘잘 동작한다’ 대신 input·sta



te·output·evidence를 쓴다

9.1. 네 칸 계약

칸	질문	T01 예
Given	어떤 시작 상태인가	progress 100, score 79
When	어떤 행동·trigger인가	policy decides review state
Then	정확히 무엇이 관찰되는가	state와 reason code
Evidence	무엇으로 다시 확인하는가	focused unit test

9.2. acceptance-to-evidence matrix

AC	종류	관찰값	check	evidence
AC1	positive	`REVIEW_REQUIRED` + reason	`npm test`	focused test name·PASS
AC2	negative	다른 state의 reason은 `null`	`npm test`	formatter test·PASS
AC3	boundary	score 80은 `COMPLETE`	`npm test`	threshold regression·PASS

9.3. evidence의 강도

```
약함: AI가 "정상입니다"라고 설명
중간: command exit 0
강함: acceptance별 expected와 actual이 보이는 test
강함: UI state screenshot + 재현 step + revision
강함: policy gate log + immutable artifact
```

설명은 evidence를 해석할 수 있지만 evidence를 대신하지 못합니다.

9.4. 실행하지 않은 것은 표시합니다

```
PASS          실제 실행해 기대 결과 확인
FAIL          실제 실행해 불일치 확인
NOT EXECUTED  실행하지 않음
UNKNOWN       어떤 check가 필요한지 아직 모름
N/A           적용되지 않으며 이유가 있음
```

`NOT EXECUTED`를 PASS처럼 보이게 하지 않습니다.

10. Exact check를 작업 안에 씁니다

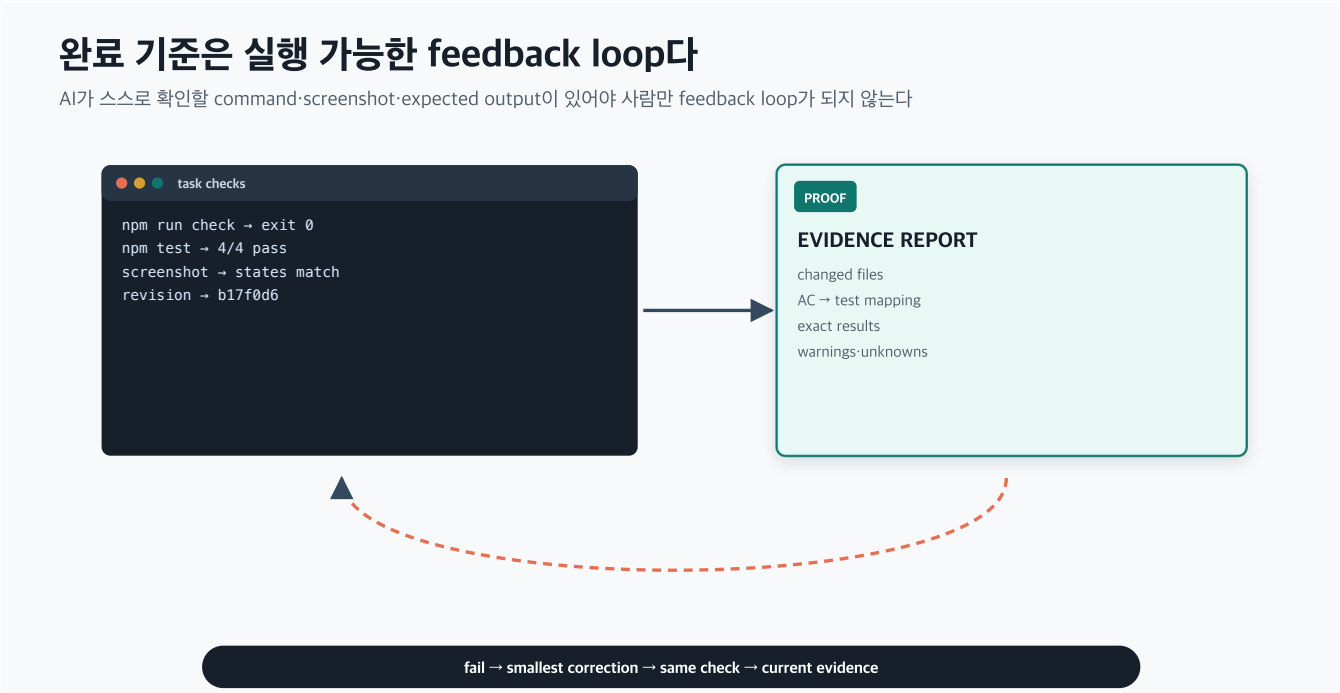
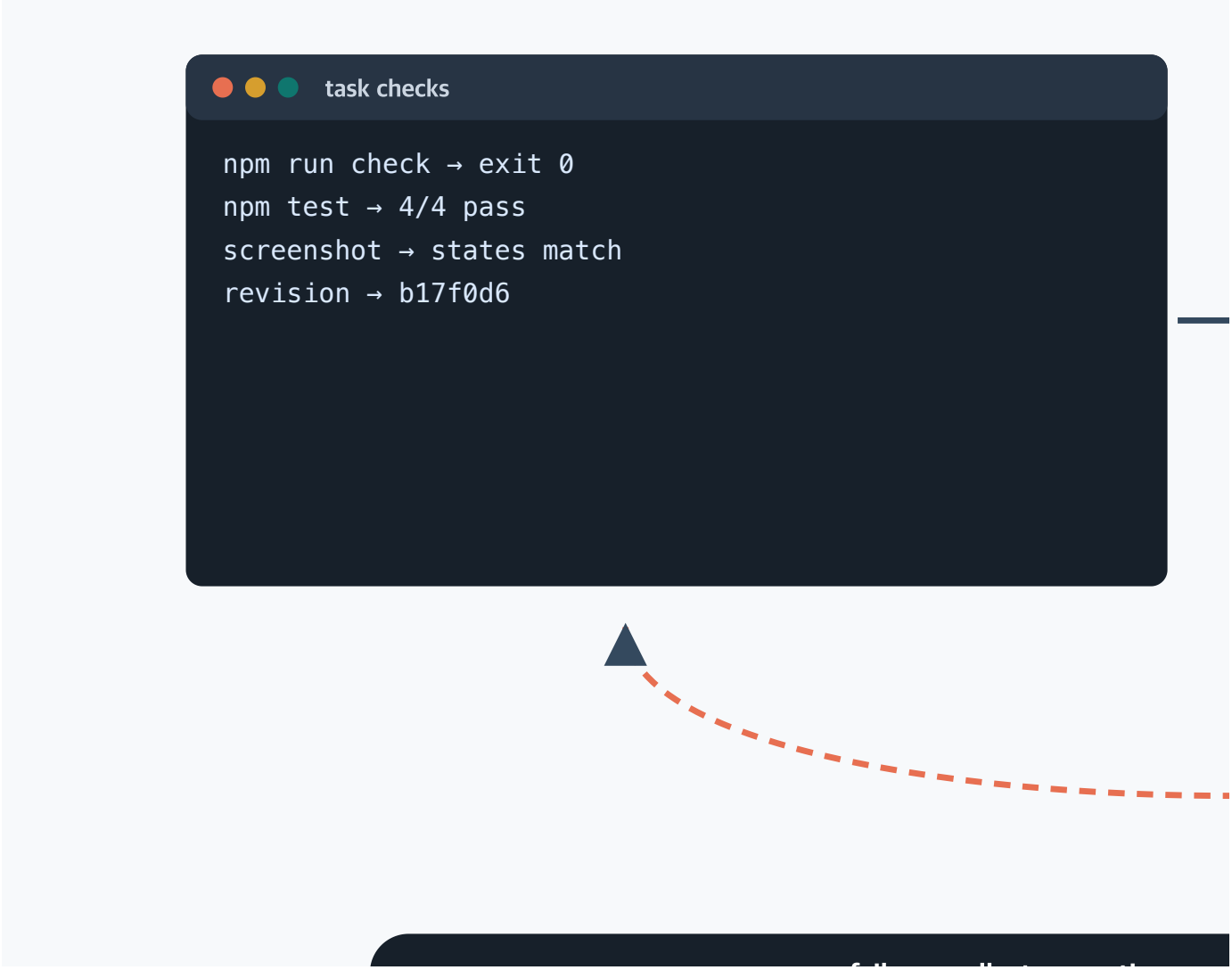
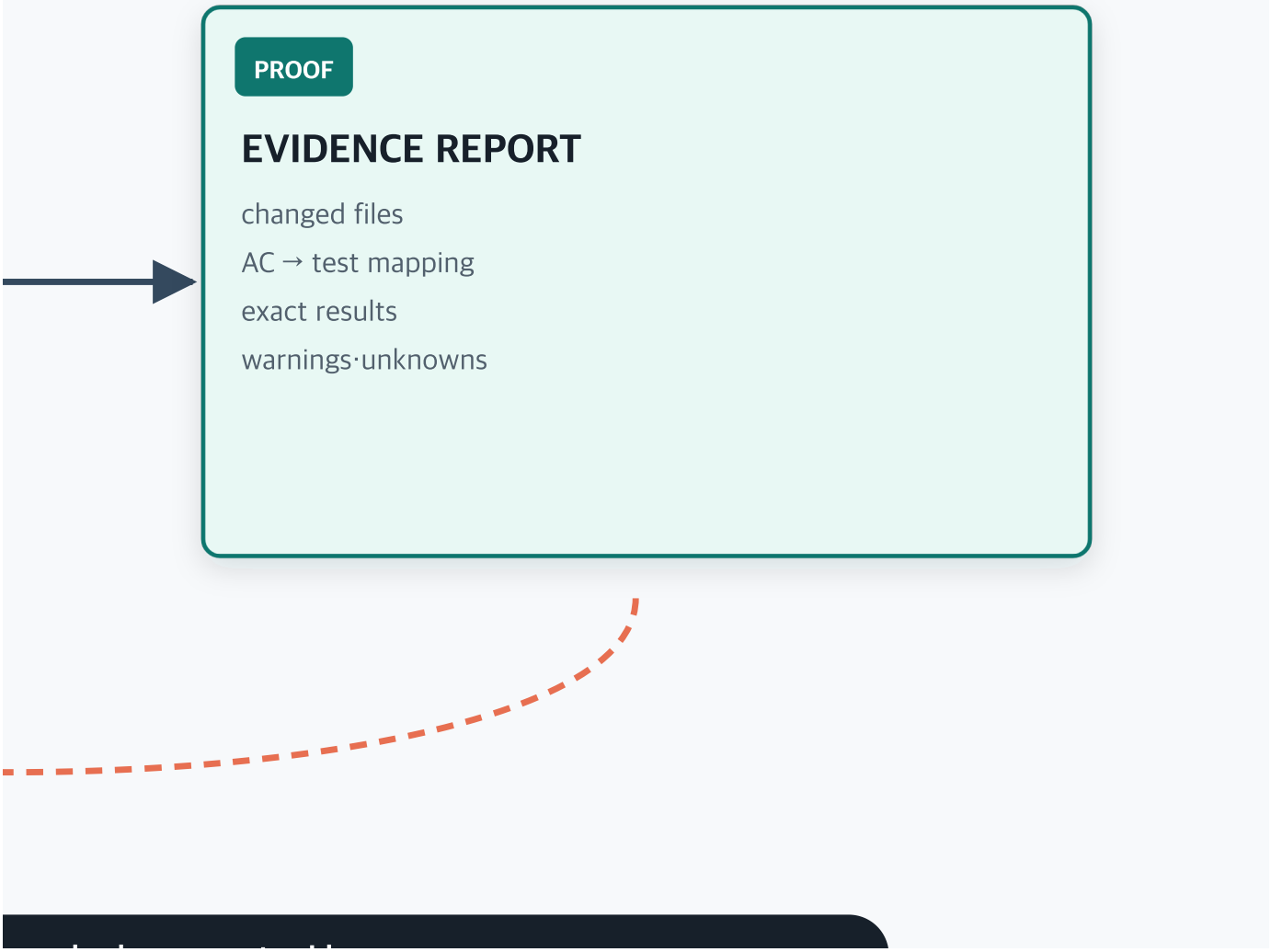


그림 9. self-verification은 구현 뒤 한 번이 아니라 작은 변경과 focused check를 반복하고 마지막에 넓은 검증으로 확장합니다.


task checks
npm run check → exit 0
npm test → 4/4 pass
screenshot → states match
revision → b17f0d6


PROOF
EVIDENCE REPORT
changed files
AC → test mapping
exact results
warnings·unknowns

10.1. “테스트한다”는 check가 아닙니다

좋은 check contract에는 다음이 있습니다.

필드	이유	예
command	무엇을 실행하는가	`npm test`
cwd	어디서 실행하는가	repository root
input·fixture	어떤 data인가	synthetic cases only
expected	무엇이 PASS인가	all tests pass·exit 0
side effect	무엇을 읽고 쓰는가	local test, no network
revision	어느 코드의 결과인가	full Git OID
artifact	무엇을 보존하는가	test output·screenshot

10.2. focused에서 broad로 갑니다

```
1. 변경한 contract의 focused test
2. 관련 module test
3. repository required check
4. 전체 regression test
5. 필요할 때만 build·preview·security check
```

모든 수정마다 가장 비싼 전체 test만 돌리면 feedback이 느려집니다. 반대로 focused test만 통과하고 끝내면 integration regression을 놓칩니다.

10.3. side effect를 분리합니다

```
OBSERVE: 파일 읽기, status 확인
CHECK: 정적 검사, format check
TEST: local fixture 실행
BUILD: artifact 생성
EXTERNAL: network·vendor 호출
MUTATE: data·migration·email·deploy
```

외부 효과가 있는 command는 정확하더라도 승인과 dry-run이 별도로 필요합니다.

11. Ready와 Done은 서로 다른 상태입니다

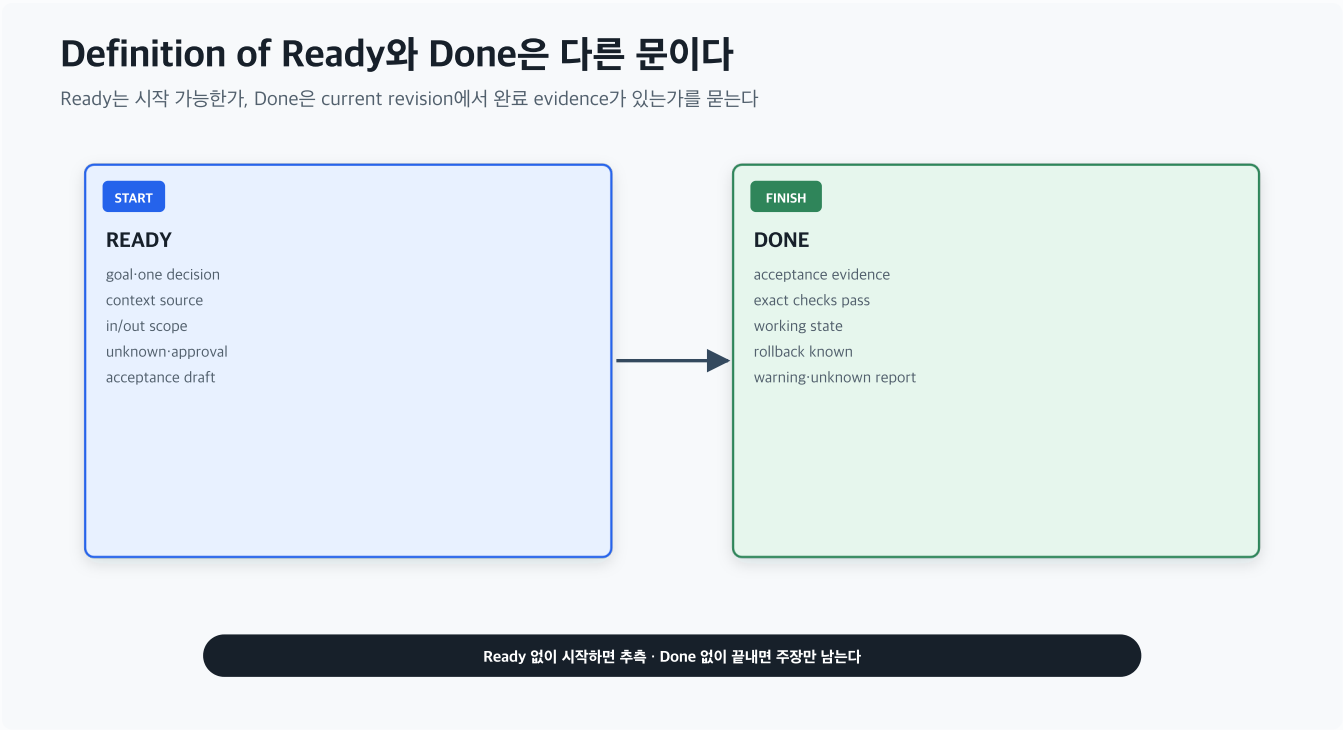


그림 10. Ready는 안전하게 시작할 수 있다는 판정이고, Done은 acceptance가 실제 evidence로 입증됐다는 판정입니다.

START

READY

goal·one decision

context source

in/out scope

unknown·approval

acceptance draft

Ready 없이 시작하면 추측 · D

FINISH

DONE

acceptance evidence

exact checks pass

working state

rollback known

warning·unknown report

one 없이 끝내면 주장만 남는다

11.1. Definition of Ready

시작 전에 확인합니다.

```
baseline revision이 있다.
outcome과 one decision이 있다.
in scope·out of scope가 있다.
dependency와 owner가 있다.
positive·negative·boundary acceptance가 있다.
exact check와 permission이 있다.
stop condition과 rollback이 있다.
```

11.2. Definition of Done

실행 후 확인합니다.

```
acceptance별 expected·actual evidence가 있다.
required check가 해당 revision에서 PASS했다.
changed surface와 이유가 기록됐다.
warning·unknown·not executed가 숨겨지지 않았다.
repository가 WORKING 상태다.
rollback path가 여전히 유효하다.
review owner가 결과를 판정할 수 있다.
```

11.3. 흔한 상태 혼동

말	실제 상태
“계획이 자세하다”	Ready 후보, Done 아님
“코드를 작성했다”	Implemented, verified 아님
“AI가 테스트했다고 했다”	evidence 확인 전 UNKNOWN
“CI가 초록이다”	CI 범위 PASS, 모든 acceptance 보장 아님
“스크린샷이 예쁘다”	한 UI state evidence, behavior 전체 아님

11.4. 상태 기계

```
DRAFT
  → READY
  → IN_PROGRESS
  → VERIFYING
  → DONE

어느 단계에서든:
  ASKING / BLOCKED / FAILED / ROLLED_BACK
```

Done을 boolean 하나로만 쓰지 말고 어떤 acceptance와 check가 어떤 상태인지 남깁니다.

12. Permission과 Stop condition을 먼저 씁니다

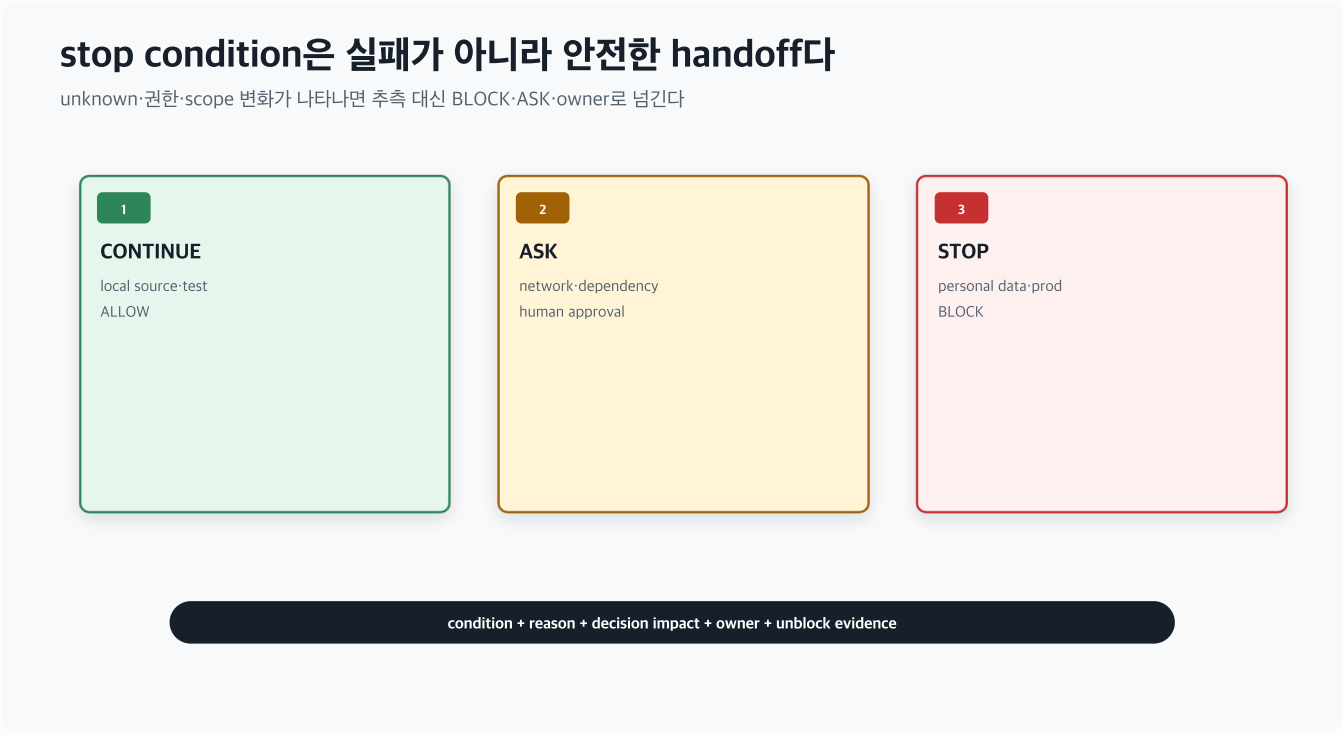


그림 11. 작업 명세는 할 일뿐 아니라 멈춰야 할 순간을 알려 줍니다. 안전한 AI 작업에서 stop은 실패가 아니라 올바른 결과입니다.

1

CONTINUE

local source·test

ALLOW

2

ASK

network·dependency

human approval

condition + reason + decision imp

3

STOP

personal data·prod

BLOCK

pact + owner + unblock evidence

12.1. 세 가지 permission

상태	의미	예
ALLOW	현재 범위에서 바로 실행 가능	local source·synthetic test
ASK	owner 승인 뒤 실행	dependency 설치·CSV field 결정
BLOCK	현재 task에서 금지	real data·production email·deploy

12.2. T01의 효과와 permission

```
permission: ALLOW
effects:
  local-source
  local-test

not allowed:
  network
  real learner data
  runtime dependency
  production endpoint
```

12.3. Stop condition은 구체적인 trigger입니다

```
STOP when:
  public consumer가 다른 compatibility contract를 요구한다.
  real learner record나 production endpoint가 필요하다.
  runtime dependency 또는 network access가 필요하다.
```

“문제가 생기면 멈춘다”는 사용할 수 없습니다. 어떤 관찰값이 scope·authority·risk를 바꾸는지 씁니다.

12.4. ASK packet

AI가 질문만 던지고 끝내지 않도록 다음을 함께 보냅니다.

```
Observed =
Why current task cannot decide =
Decision owner =
Options =
Risk of each option =
Recommended safe default =
Work that can continue meanwhile =
```

30초 확인 3

CSV export에 learner email을 넣을지 모르는데 구현을 계속해야 합니까? data owner, 최소 field, 승인 evidence가 없으면 research 또는 ASK로 바꿉니다.

13. 행동 변경과 Refactor를 분리합니다

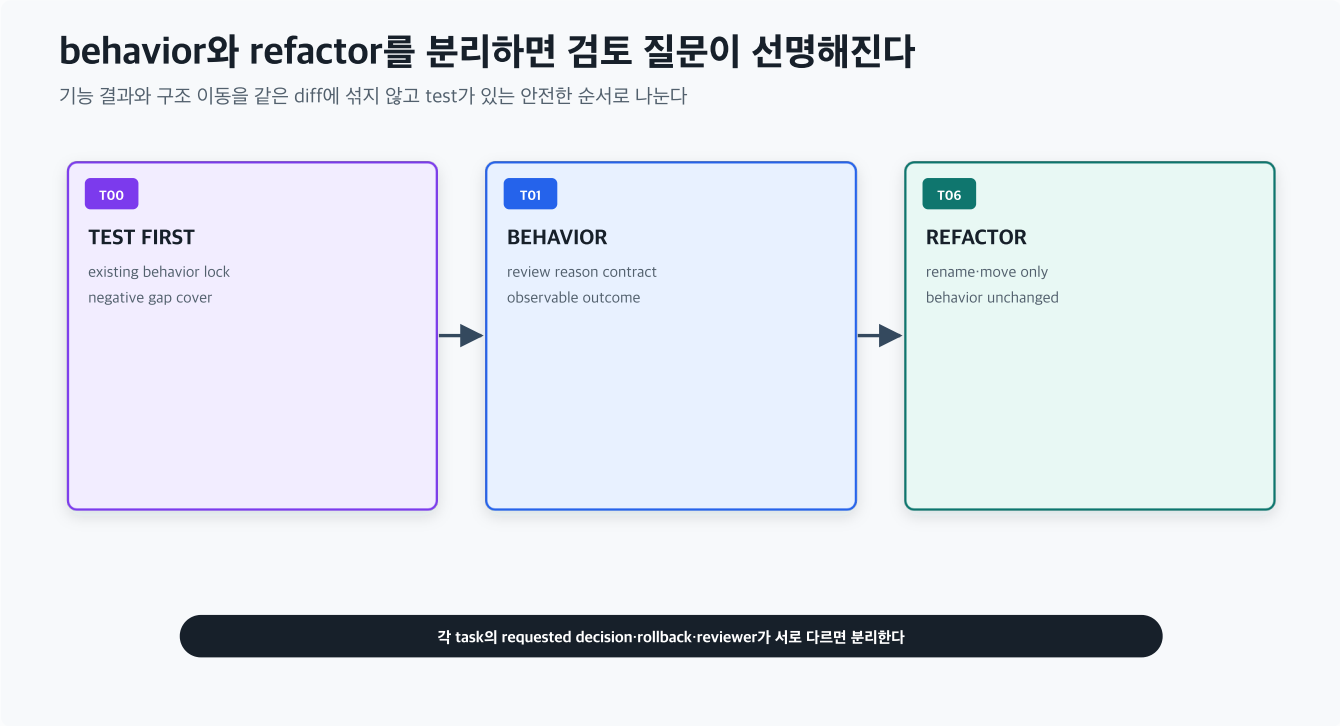
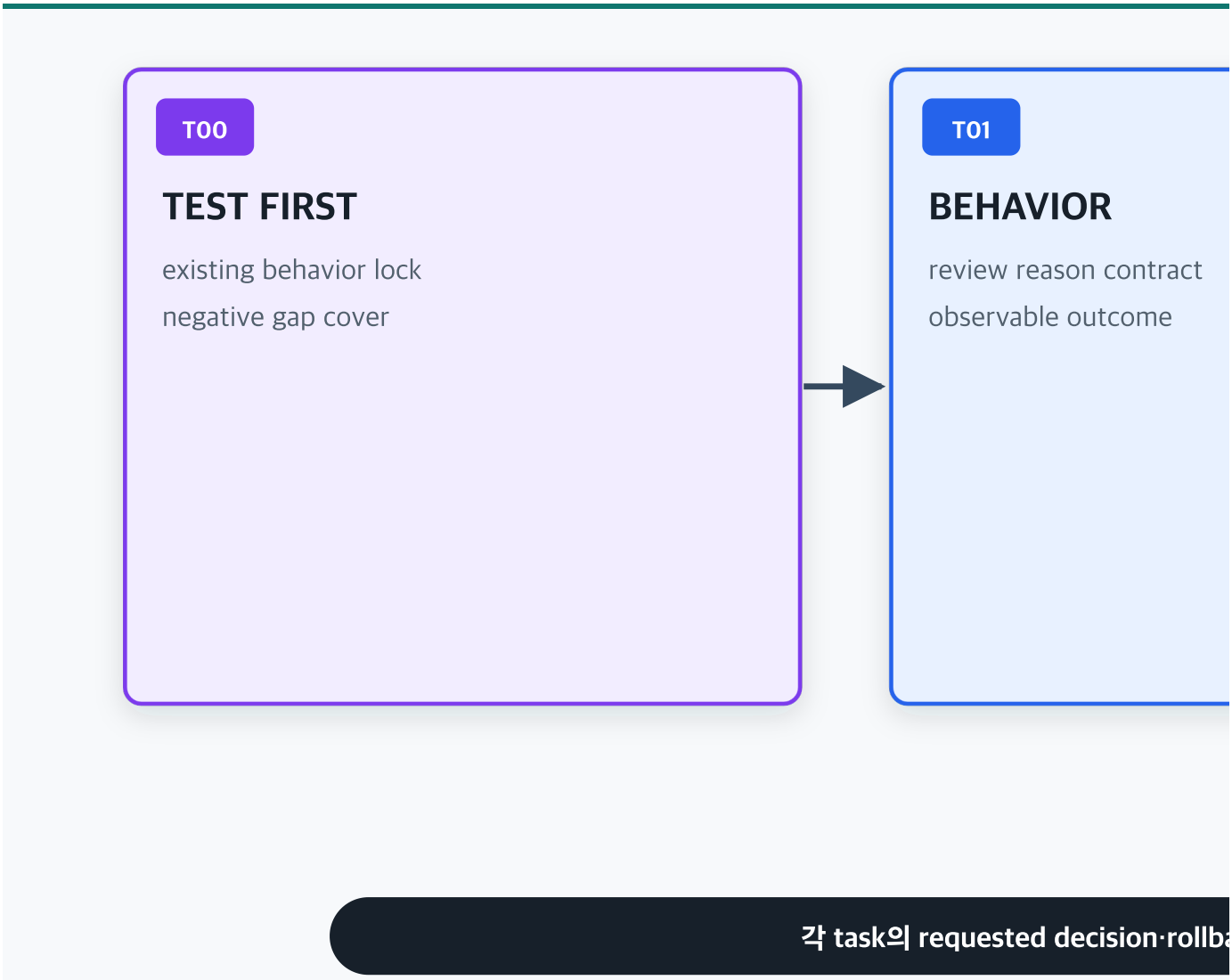
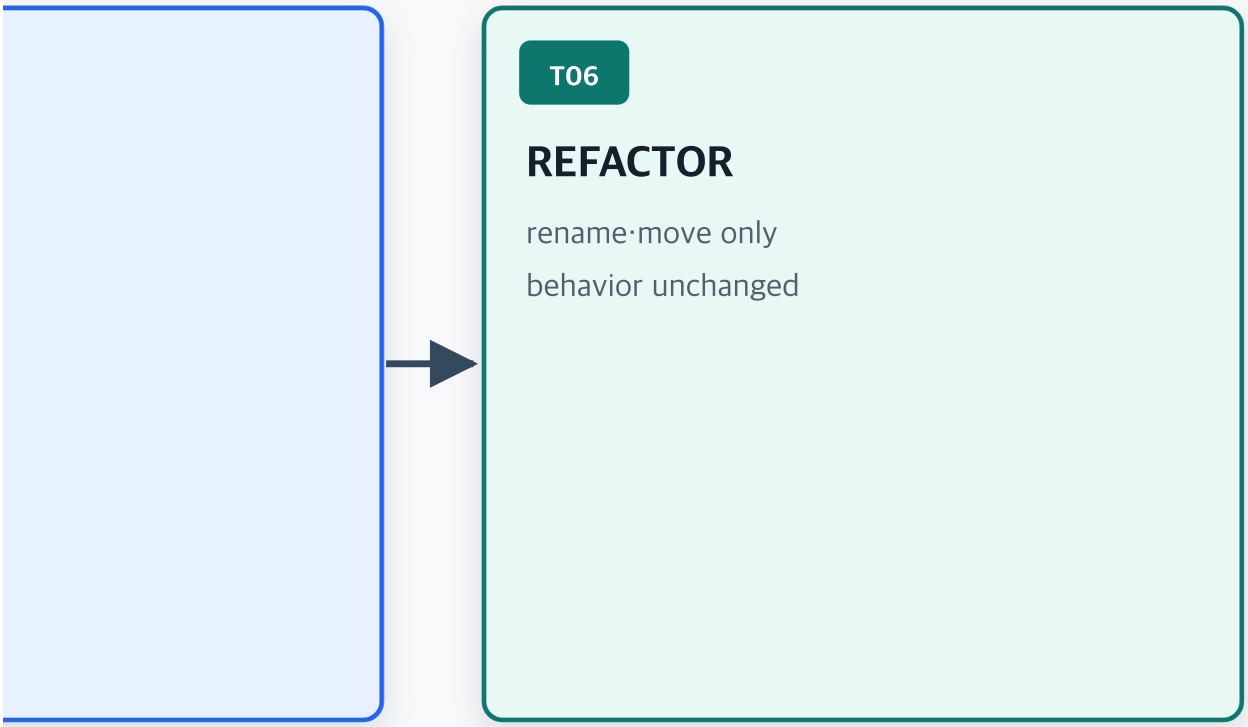


그림 12. 같은 파일을 만져도 review 질문과 rollback이 다르면 별도 작업입니다.


T00
TEST FIRST
existing behavior lock
negative gap cover
T01
BEHAVIOR
review reason contract
observable outcome
각 task의 requested decision·rollb



ack·reviewer가 서로 다르면 분리한다

13.1. 섞인 작업

```
reviewReason을 추가하면서 progress module 이름을 전부 바꾼다.
```

오류가 나면 새 contract 때문인지 rename 누락 때문인지 분리하기 어렵습니다. reviewer도 behavior와 구조 두 관점을 동시에 확인해야 합니다.

13.2. 분리된 작업

```
T01 behavior:
  policy가 stable reviewReason을 소유한다.
  evidence: behavior contract tests

T06 refactor:
  public behavior를 유지하면서 module name을 변경한다.
  evidence: reference audit + unchanged behavior tests
```

13.3. 분리 판단 질문

질문	다르면 분리
reviewer가 묻는 핵심 질문	behavior가 맞는가 / 구조가 명확한가
acceptance	새 결과 / 동일 결과 유지
rollback	contract 제거 / rename 되돌림
owner	product·policy / maintainer
failure diagnosis	rule 오류 / reference 누락

Google Engineering Practices도 기능 변경과 refactoring을 분리하면 review와 rollback이 쉬워진다고 권고합니다.

14. 작업 명세의 열두 블록

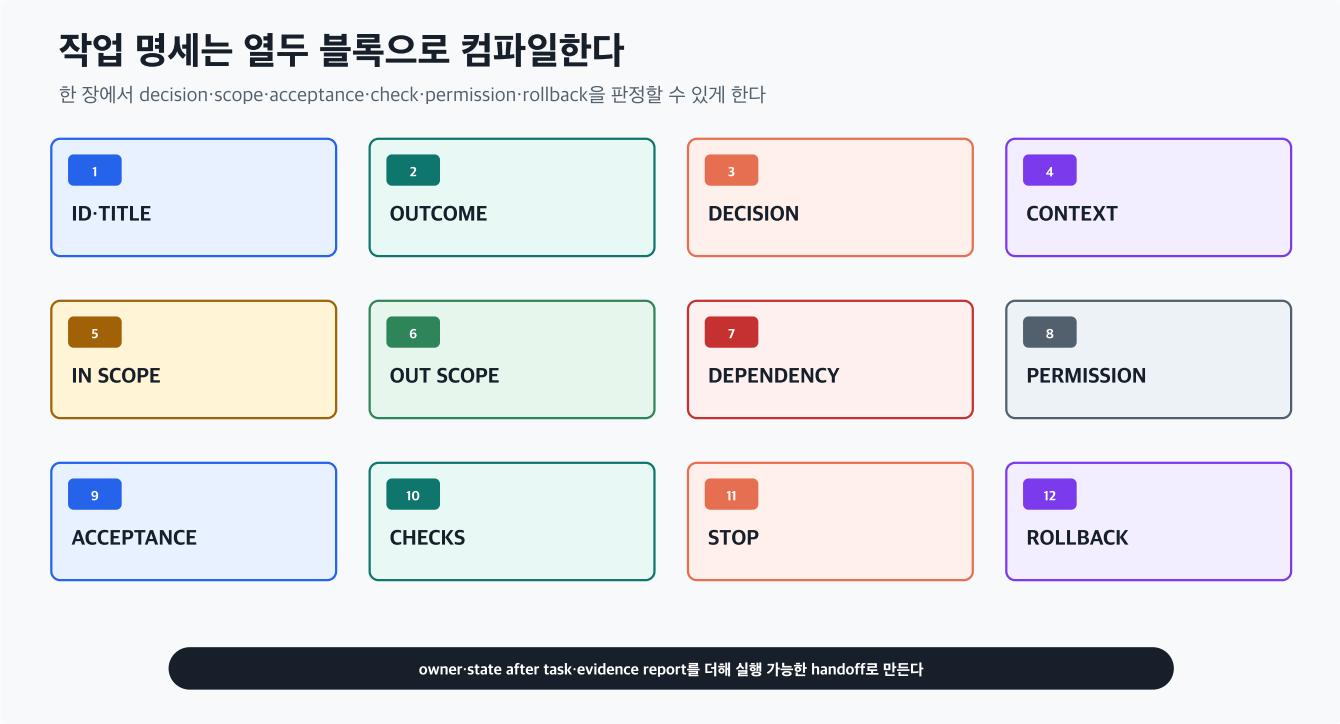


그림 13. 이 블록들은 문서를 길게 만들기 위한 항목이 아니라 추정하면 위험한 빈칸을 닫는 최소 계약입니다.

1

ID·TITLE

2

OUTCOME

5

IN SCOPE

6

OUT SCOPE

9

ACCEPTANCE

10

CHECKS

3

DECISION

4

CONTEXT

7

DEPENDENCY

8

PERMISSION

11

STOP

12

ROLLBACK

블록 1 · Identity

```
task ID, title, type, owner
baseline revision, current status
```

ID는 dependency와 evidence를 연결합니다. title은 활동보다 outcome 또는 decision을 드러냅니다.

블록 2 · Outcome

```
[actor/consumer]가 [trigger]에서 [observable result]를 얻는다.
```

블록 3 · One decision

```
이번 작업에서 owner·contract·boundary 중 무엇을 하나 결정하는가?
```

블록 4 · Context sources

```
AGENTS.md
requirements/big-request.md
관련 source·test
Accepted decision record
```

문서를 통째로 복사하지 않고 현재 task에 필요한 source와 anchor를 가리킵니다.

블록 5 · In scope

outcome을 완성하는 change surface와 test를 씁니다.

블록 6 · Out of scope

헷갈리기 쉬운 인접 요구, 금지 행동, deferred decision을 씁니다.

블록 7 · Dependency·working state

어느 contract를 소비하며 작업 뒤 repository가 어떤 상태인지 씁니다.

블록 8 · Permission·effects

ALLOW·ASK·BLOCK과 network, data, write, external side effect를 씁니다.

블록 9 · Acceptance

positive·negative·boundary 각각에 Given·When·Then·Evidence를 씁니다.

블록 10 · Checks

command, cwd, expected, side effect, artifact를 씁니다.

블록 11 · Stop·rollback

어떤 조건에서 누구에게 질문하며, 실패 시 어느 범위를 되돌리는지 씁니다.

블록 12 · Evidence report

```
baseline·head revision
changed files·decisions
acceptance-to-evidence mapping
exact check results
warnings·unknowns·not executed
remaining risk·next owner
```

15. 작업 전 일곱 Readiness Gate

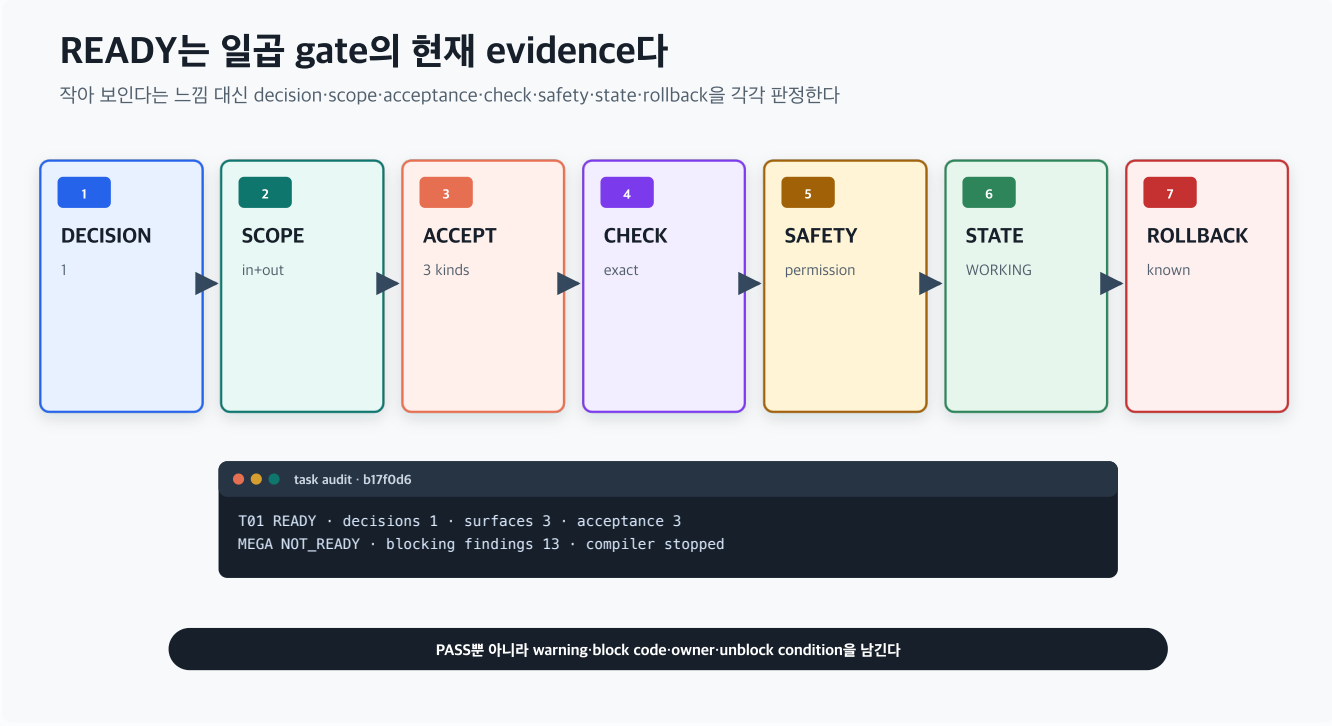
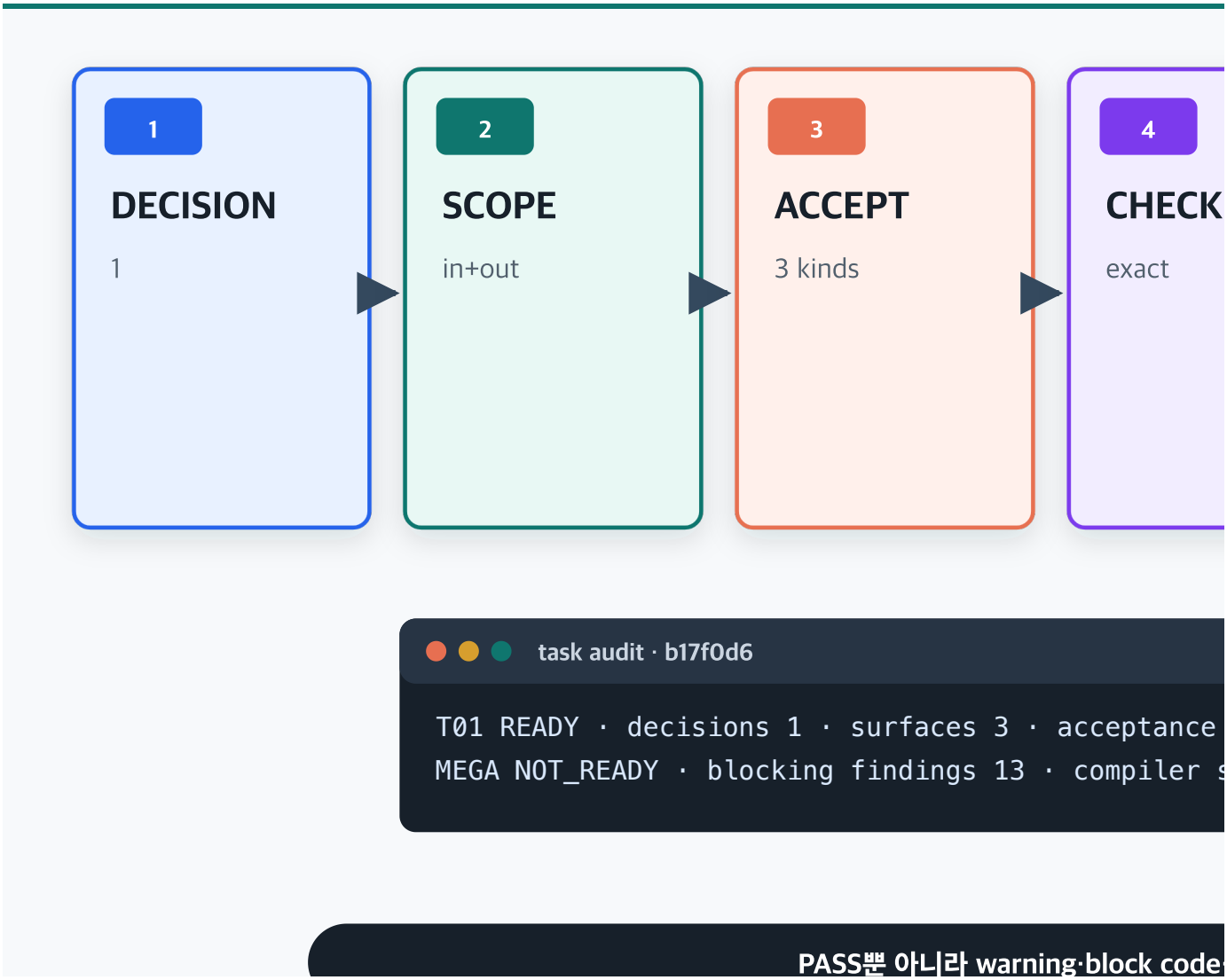
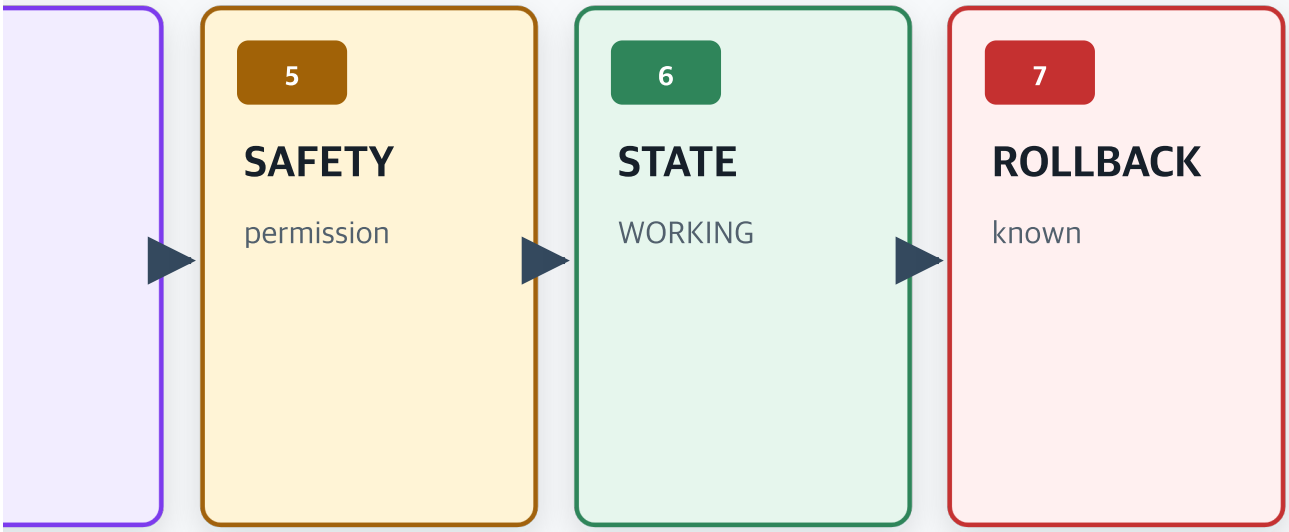


그림 14. 하나라도 BLOCK이면 task packet을 AI에게 넘기기 전에 빈칸을 해결하거나 research·ASK task로 전환합니다.


1
DECISION
1
2
SCOPE
in+out
3
ACCEPT
3 kinds
4
CHECK
exact
task audit · b17f0d6
T01 READY · decisions 1 · surfaces 3 · acceptance
MEGA NOT_READY · blocking findings 13 · compiler
PASS뿐 아니라 warning·block code




```
 3
stopped
```

owner·unblock condition을 남긴다

Gate 1 · Outcome

```
actor·trigger·observable result가 있는가?
활동 목록이 outcome을 대신하지 않는가?
```

Gate 2 · Decision

```
decision이 정확히 하나인가?
owner와 contract boundary가 드러나는가?
```

Gate 3 · Scope

```
in·out scope가 모두 있는가?
review surface가 한 번에 이해 가능한가?
```

Gate 4 · Acceptance

```
positive·negative·boundary가 있는가?
각 Then은 실제 관찰 가능한가?
```

Gate 5 · Verification

```
exact command·cwd·expected·artifact가 있는가?
required check가 누락되지 않았는가?
```

Gate 6 · Safety

```
permission과 side effect가 분명한가?
data·network·email·migration·deploy에 승인 경계가 있는가?
stop condition과 rollback이 있는가?
```

Gate 7 · Working state

```
작업 뒤 repository가 WORKING인가?
dependency 누락·cycle이 없는가?
deferred 요구가 이유와 함께 보존되는가?
```

15.1. 판정 상태

상태	의미	다음 행동
READY	required gate 모두 PASS	task packet 전달
READY_WITH_WARNINGS	실행 가능, 후속 위험 존재	warning·owner 보존
NOT_READY	명세 핵심 빈칸	재분할·acceptance 보완
BLOCKED	authority·safety 해결 필요	owner decision 대기

16. 완성된 T01 작업 명세

다음은 실습 저장소의 machine-readable task를 사람이 읽을 수 있게 compile한 예입니다.

```
# T01 · Expose a stable review reason through policy and HTTP

baseline revision: b17f0d6e09fd1f4066e7a63b1ff298f3e5d38f72
type: vertical
owner: policy maintainer
permission: ALLOW

Outcome
  An API consumer can identify why a synthetic case
  requires manual review.

Decision
  Policy owns one stable reviewReason code,
  and HTTP passes it through.

In scope
  reviewReason on REVIEW_REQUIRED policy output
  HTTP pass-through
  positive·negative·boundary tests

Out of scope
  dashboard queue, CSV export,
  email, analytics, persistence, deployment
```

16.1. acceptance

```
AC1 positive
Given progress 100 and score 79
When policy decides review state
Then state is REVIEW_REQUIRED and
     reviewReason is SCORE_BELOW_THRESHOLD
Evidence focused unit test

AC2 negative
Given state is COMPLETE or IN_PROGRESS
When result is formatted
Then reviewReason is null and HTTP does not recompute it
Evidence policy and formatter tests

AC3 boundary
Given progress 100 and score exactly 80
When policy decides review state
Then state remains COMPLETE and reviewReason is null
Evidence threshold regression test
```

16.2. checks·stop·rollback

```
CHECK npm run check
  cwd repository root
  expect exit 0
  effect local read, no network

TEST npm test
  cwd repository root
  expect all tests pass
  effect local test process, no network

STOP
  public consumer requires another compatibility contract
  real learner record or production endpoint becomes necessary
  runtime dependency or network access becomes necessary

ROLLBACK
  remove reviewReason fields and focused tests
  inside the same task range
```

16.3. audit 결과

gate	result
decision count	PASS · one decision
scope boundary	PASS · in and out present
review surface	PASS · 3 surfaces
acceptance kinds	PASS · positive·negative·boundary
required checks	PASS · check and test
permission	PASS · ALLOW
rollback	PASS
stop conditions	PASS · 3
state after task	PASS · WORKING

최종 판정은 `READY` 입니다. compiler는 audit이 PASS한 뒤에만 task packet을 만듭니다.

17. 실패하는 Mega Task를 해부합니다

```
title:
  Build the entire operator review system

decisions:
  policy reason
  dashboard queue
  CSV with learner data
  automatic email
  deployment

acceptance:
  Everything works and looks good.

permission:
  ALLOW
```

17.1. audit가 잡아낸 것

finding	뜻	고치는 방법
MISSING_OWNER	결정을 승인할 owner 없음	decision별 owner 분리
DECISION_COUNT 5	검토 질문 다섯 개	task 분할
SCOPE_BOUNDARY	in·out scope 없음	포함·제외 명시
REVIEW_SURFACE 7	한 번에 넓은 surface	vertical slice 축소
MISSING negative	하지 않을 행동 없음	non-target·unauthorized 작성
MISSING boundary	threshold 경계 없음	exact edge 작성
NON_OBSERVABLE	“works and looks good”	expected actual로 변환
MISSING_CHECK	실행 가능한 command 없음	exact check 연결
UNSAFE_PERMISSION	email·data·deploy를 ALLOW	ASK·BLOCK 분리
MISSING_ROLLBACK	실패 범위 불명	task별 rollback
MISSING_STOP	authority 변화 시 계속 진행	trigger·owner 작성
BROKEN_STATE	중간 상태 UNKNOWN	WORKING slice 설계

결론은 `NOT_READY`입니다. 더 자세한 prompt로 보완하는 것이 아니라 작업 자체를 다시 나눠야 합니다.

18. 작업 분할 판정 데스크 실습

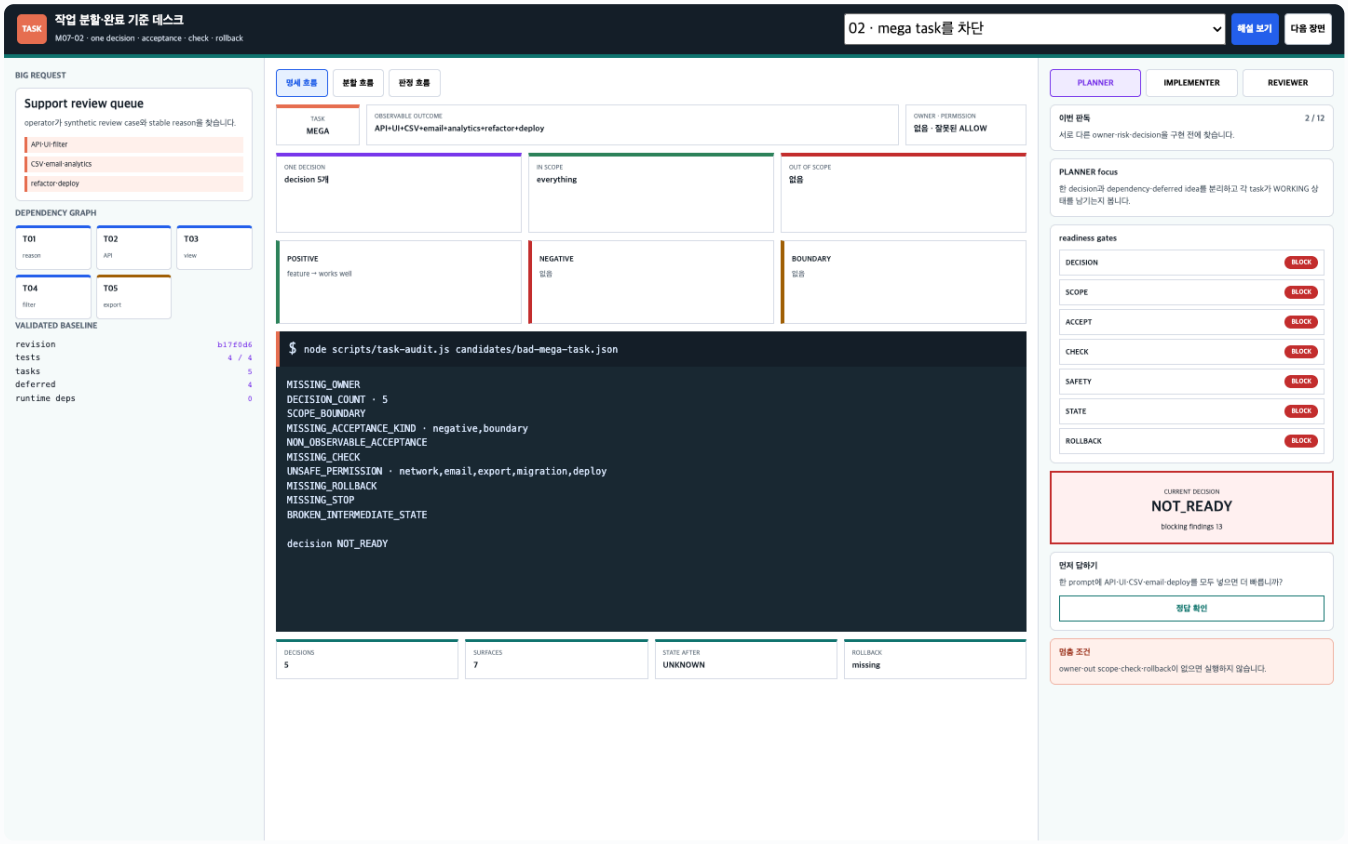


그림 15. 12개 장면에서 PLANNER·IMPLEMENTER·REVIEWER 역할과 명세·분할·판정 모드를 바꿔 readiness를 판단합니다.

18.1. 생성 결과

```
gibalja-task-spec-practice/
├── repo/
│   ├── AGENTS.md
│   ├── requirements/big-request.md
│   ├── context/task-policy.json
│   ├── candidates/bad-mega-task.json
│   ├── plans/task-graph.json
│   ├── tasks/T01...T05.json
│   ├── src/
│   ├── test/
│   └── scripts/
└── evidence/
```

18.2. deterministic baseline

```
Git: 2.54.0
Node: 24.14.0
revision: b17f0d6e09fd1f4066e7a63b1ff298f3e5d38f72
runtime dependency: 0
baseline behavior tests: 4/4 PASS
T01 task audit: READY
plan audit: READY
compiled task packet: 1,603 characters
```

실습 generator는 Git metadata를 고정해 같은 tool version과 script에서 같은 repository OID를 만들도록 설계했습니다. 실제 업무에서는 OID를 예상하지 말고 현재 full revision을 직접 관찰합니다.

18.3. 12개 장면

장면	핵심 판정
1	활동을 actor·observable outcome으로 바꾸는가
2	decision이 정확히 하나인가
3	vertical과 horizontal 중 무엇인가
4	task가 너무 작거나 너무 크지 않은가
5	dependency DAG와 working state가 안전한가
6	in·out scope와 deferred가 보존되는가
7	positive acceptance가 관찰 가능한가
8	negative와 boundary가 있는가
9	exact check와 evidence가 연결되는가
10	Ready와 Done을 구분하는가
11	permission·stop·rollback이 안전한가
12	mega task를 NOT_READY로 막는가

상세 command와 기록 순서는 단계별 실습서를 따릅니다.

19. 60분 실전 미션

미션 1 · baseline·request inventory · 5분

```
repository root =
full revision =
working tree =
requirement source =
current authority =
```

큰 요청에서 명시된 아이디어를 빠짐없이 목록으로 만듭니다. 지금 할 것과 나중에 할 것을 아직 나누지 않습니다.

미션 2 · outcome · 5분

actor, trigger, observable result, evidence를 한 문장으로 씁니다.

미션 3 · decision map · 10분

요청 속 decision을 owner·contract·risk 기준으로 분리합니다. 비슷해 보여도 owner와 rollback이 다르면 다른 decision입니다.

미션 4 · slice 선택 · 10분

각 decision을 vertical·enabling·research·risk-first 중 하나로 분류합니다. 첫 task는 가장 작은 사용자 outcome 또는 가장 큰 risk를 줄이는 slice로 정합니다.

미션 5 · DAG·scope · 5분

dependency를 화살표로 그리고 각 node 뒤 `WORKING`을 확인합니다. in scope·out of scope·deferred reason을 씁니다.

미션 6 · acceptance trio · 10분

positive, negative, boundary를 Given·When·Then으로 씁니다. “잘”, “정상”, “문제없이” 같은 비관찰 표현을 바꿉니다.

미션 7 · checks·safety · 10분

각 acceptance에 exact check와 evidence를 연결하고 ALLOW·ASK·BLOCK, effects, stop condition, rollback을 씁니다.

미션 8 · readiness audit · 5분

일곱 gate를 PASS·WARN·BLOCK으로 판정합니다. BLOCK 하나라도 있으면 compiler나 AI 구현을 시작하지 않습니다.

20. 자주 실패하는 작업 분할 패턴

실패	왜 위험한가	고치는 evidence
파일 하나당 task 하나	독립 outcome 없음	one decision + working state
줄 수로 small 판정	위험·owner를 못 봄	review question count
활동만 나열	성공 판정 불가	actor·observable outcome
happy path만 작성	non-target·edge 누락	positive·negative·boundary
“테스트 완료”	command·revision 불명	exact check result
UI와 API를 무조건 분리	end-to-end 결과 지연	작은 vertical slice
모든 기반을 먼저 구축	과잉 설계	next outcome에 필요한 enabling만
refactor를 끼워 넣음	rollback·진단 혼합	behavior와 별도 task
out of scope 없음	AI가 인접 요구 추정	explicit exclusion
deferred를 삭제	요구 유실	reason·owner·trigger
dependency를 번호로 표현	contract 방향 불명	DAG edge reason
중간 build가 깨짐	다음 task와 결합	WORKING state gate
email·deploy를 ALLOW	외부 효과 무승인	ASK·BLOCK·dry-run
stop condition 없음	scope creep에도 계속	observable trigger
rollback이 “git revert”뿐	data·external effect 미복구	effect별 recovery
Ready를 Done으로 보고	계획과 증거 혼동	acceptance evidence
AI 설명을 evidence로 사용	재현 불가	log·test·artifact
giant task에 prompt만 추가	근본 복잡성 유지	decision별 재분할

21. 셀프 테스트 · 먼저 답한 뒤 해설을 엽니다

Q1

작업 크기를 판정할 때 줄 수보다 중요한 세 가지는 무엇입니까?

정답 보기
한 가지 decision인지, 독립적으로 검증 가능한지, 끝난 뒤 repository가 WORKING인지가 핵심입니다. 여기에 reversible scope와 owner·evidence를 함께 봅니다.

Q2

policy, HTTP, test 세 파일을 바꾸면 항상 큰 작업입니까?

정답 보기
아닙니다. 세 surface가 하나의 stable reviewReason contract를 end-to-end로 완성하고 같은 acceptance·rollback으로 판정된다면 하나의 작은 vertical task가 될 수 있습니다.

Q3

“API를 만들고 UI를 개선한다”를 먼저 무엇으로 바꿔야 합니까?

정답 보기
actor, trigger, observable result와 evidence가 있는 outcome으로 바꿉니다. 활동은 그 outcome을 달성하는 수단으로 뒤에 배치합니다.

Q4

수평 slice가 적절한 경우는 언제입니까?

정답 보기
다음 vertical task의 실제 risk를 줄이는 contract, test harness, migration 준비 같은 enabling outcome이 있고 독립 evidence와 rollback을 가질 때입니다.

Q5

CSV field와 data owner가 정해지지 않았는데 CSV 구현 task를 만들면 어떤 상태입니까?

정답 보기
구현 task는 BLOCKED 또는 NOT_READY입니다. 먼저 질문·시간 상한·source·decision deliverable이 있는 research task를 만들거나 data owner에게 ASK합니다.

Q6

positive acceptance 하나가 있어도 negative와 boundary가 필요한 이유는 무엇입니까?

정답 보기
positive는 의도한 성공만 보여 줍니다. non-target 상태에서 하지 않아야 할 행동과 threshold가 바뀌는 정확한 가장자리를 검증해야 contract가 닫힙니다.

Q7

“모든 테스트 통과”만 기록하면 충분합니까?

정답 보기
아닙니다. command, cwd, input, expected, side effect, 실행 revision, 결과 artifact와 acceptance-to-test mapping이 있어야 재현하고 판정할 수 있습니다.

Q8

Ready와 Done의 차이는 무엇입니까?

정답 보기
Ready는 outcome·scope·acceptance·checks·safety가 있어 안전하게 시작할 수 있는 상태입니다. Done은 실제 변경 revision에서 acceptance와 required checks가 evidence로 입증된 상태입니다.

Q9

작업 중 network access가 새로 필요해졌습니다. 명세에 없었다면 어떻게 합니까?

정답 보기
stop condition에 따라 멈추고 ASK packet을 만듭니다. 목적, endpoint, data, credential, side effect와 owner 승인을 확인한 뒤 scope와 permission을 다시 판정합니다.

Q10

기능 변경과 module rename을 분리해야 하는 핵심 기준은 무엇입니까?

정답 보기
review 질문, acceptance, owner, failure diagnosis와 rollback이 다르기 때문입니다. 같은 파일을 만져도 이 기준이 다르면 별도 task로 둡니다.

Q11

dependency graph에서 각 task 뒤 `WORKING`을 확인하는 이유는 무엇입니까?

정답 보기
다음 task가 없어도 현재 변경을 실행·검토·rollback할 수 있게 하기 위해서입니다. 깨진 중간 상태는 task가 실제로 독립적이지 않다는 신호입니다.

Q12

Mega task audit에서 하나의 BLOCK이 나오면 compiler를 실행해도 됩니까?

정답 보기
아닙니다. decision·scope·acceptance·check·permission 같은 required gate의 BLOCK을 먼저 해결하거나 task를 다시 나눕니다. compiler는 READY 판정 뒤에만 실행합니다.

22. 최종 제출 체크리스트

baseline·outcome

- [ ] repository root·full revision·working tree가 있다.
- [ ] requirement source와 current owner가 있다.
- [ ] actor·trigger·observable outcome·evidence가 있다.
- [ ] 활동 목록이 outcome을 대신하지 않는다.

decision·slice

- [ ] task마다 decision이 정확히 하나다.
- [ ] vertical·enabling·research·risk-first 유형을 적었다.
- [ ] review 질문과 owner가 하나의 초점이다.
- [ ] behavior와 refactor를 분리했다.

scope·dependency

- [ ] in scope와 out of scope가 모두 있다.
- [ ] deferred 항목에 reason·owner·trigger가 있다.
- [ ] dependency edge가 소비하는 contract를 설명한다.
- [ ] missing dependency와 cycle이 없다.
- [ ] 각 task 뒤 repository가 WORKING이다.

acceptance·verification

- [ ] positive·negative·boundary acceptance가 있다.
- [ ] Given·When·Then이 관찰 가능하다.
- [ ] acceptance마다 evidence가 연결된다.
- [ ] exact command·cwd·expected·side effect가 있다.
- [ ] focused check와 broad regression을 구분했다.

safety·recovery

- [ ] permission이 ALLOW·ASK·BLOCK으로 분류됐다.
- [ ] data·network·email·migration·deploy 효과를 적었다.
- [ ] observable stop condition과 decision owner가 있다.
- [ ] task 범위의 rollback과 external effect recovery가 있다.

evidence·decision

- [ ] Ready와 Done을 별도로 판정했다.
- [ ] baseline·head revision이 있다.
- [ ] changed surface와 acceptance-to-evidence mapping이 있다.
- [ ] PASS·FAIL·NOT EXECUTED·UNKNOWN을 구분했다.
- [ ] warning·remaining risk·next owner가 있다.

23. 공식 자료와 확인 범위

OpenAI Codex

- Codex best practices

- Codex prompting

공식 가이드의 goal·context·constraints·done when 구조, 명확한 검증 기준, scoped task와 가장 작은 안전한 변경 원칙을 이 매뉴얼의 task contract에 적용했습니다.

Google Engineering Practices

- Small CLs

작고 self-contained한 change, 관련 test 포함, system의 working state 유지, 쉬운 review·rollback, 기능 변경과 refactor 분리 원칙을 적용했습니다. Google 문서도 단순한 line count로 small을 정의하지 않습니다.

GitHub

- About task lists
- Adding sub-issues

Markdown task list는 진행을 보이게 할 수 있지만 dependency·acceptance·permission을 자동으로 보장하지 않습니다. 복잡한 hierarchy는 current GitHub sub-issues 기능과 project 관계를 확인해 사용합니다.

Claude Code·Gemini CLI

- Claude Code best practices
- Claude Code agent teams
- Gemini CLI task planning
- Gemini CLI plan mode

공식 문서의 verification criteria, evidence 중심 검증, explore→plan→implement, self-contained deliverable, 너무 작거나 큰 task 회피, plan visibility와 파일별 verification 원칙을 적용했습니다.

규범 언어

- RFC 2119 key words
- RFC 8174 clarification

조직에서 `MUST`, `SHOULD`, `MAY`를 규범적으로 사용할 때는 의미를 정의하고 대문자 키워드의 적용 범위를 명확히 합니다. 이 매뉴얼의 ALLOW·ASK·BLOCK은 실행 permission 판정을 위한 별도 학습 표기입니다.

적용 원칙

AI tool의 planning, task, subagent, review 기능과 UI는 version·surface·setting에 따라 달라질 수 있습니다. 이 매뉴얼은 2026-07-16에 공식 자료를 확인해 작성했으며, 실제 작업에서는 현재 tool documentation과 repository policy를 다시

확인합니다. 개인정보, 외부 발송, migration, production 접근, 배포에는 조직의 보안·data·변경 승인 절차가 우선합니다.

다음 매뉴얼: M07-03 「AI 생성 코드를 검토하고 되돌리기」에서 이 작업 명세를 기준으로 변경 diff, test evidence, 위험, rollback 가능성을 검토합니다.