M06-02 · GIBALJA VISUAL MANUAL

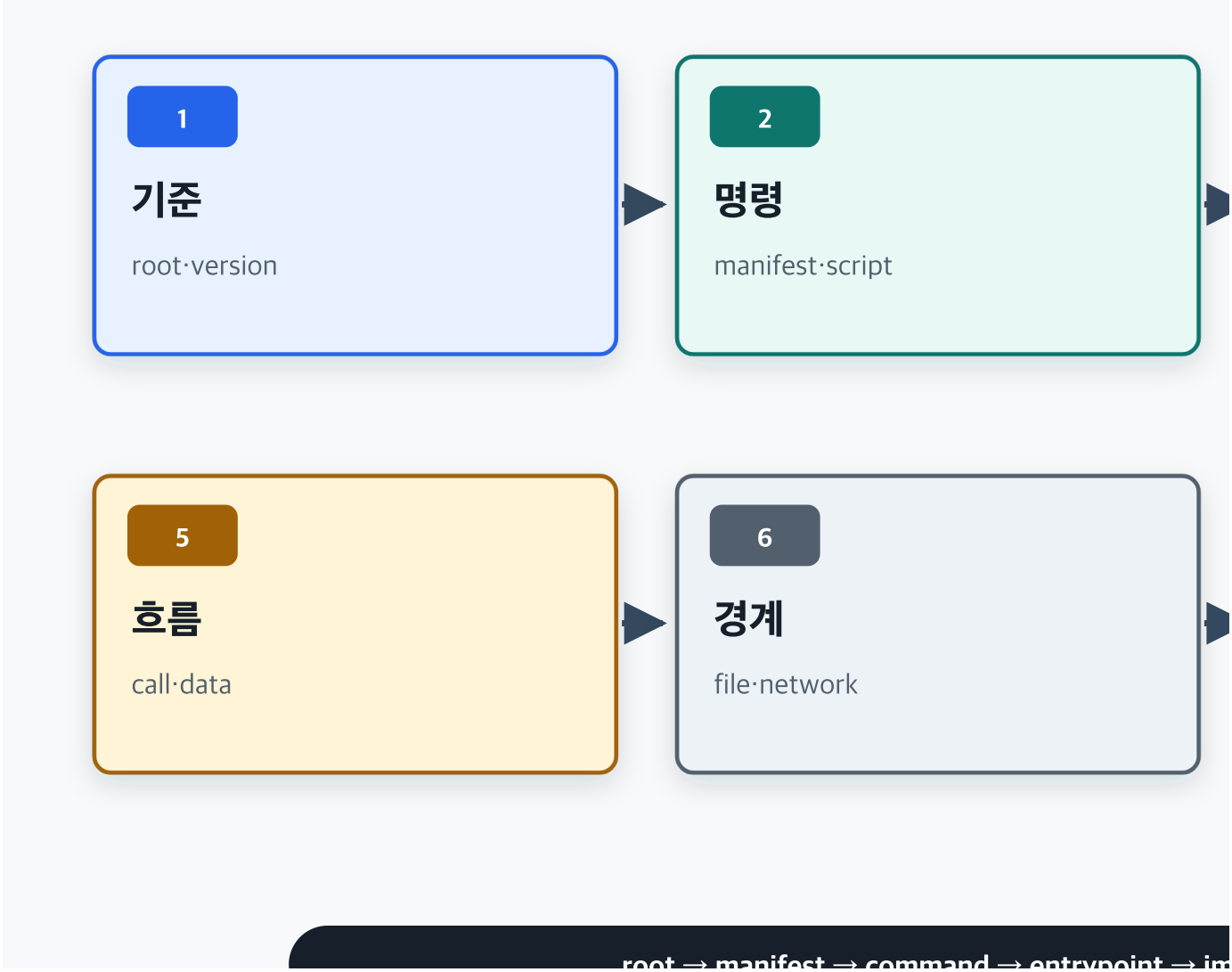
코드 구조와 설정 파일 읽기

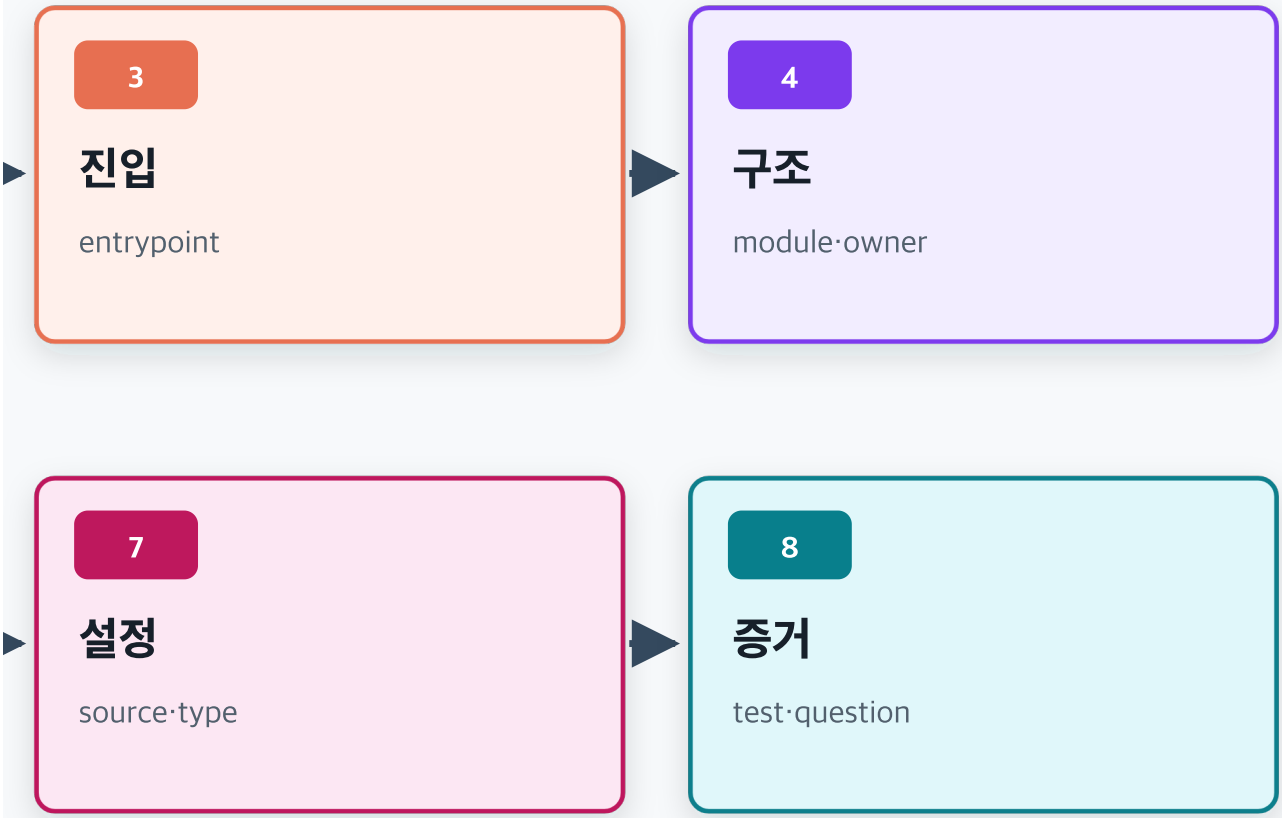
> **한 문장 목표:** `repository 기준 → runtime·manifest → 실행 command → entrypoint → import graph → call·data flow → I/O boundary → config source·type → test·deploy gap → evidence` 순서로 읽고, 한 기능의 실제 동작을 파일 이름 추측이 아니라 재현 가능한 경로로 설명합니다.

난이도	개념	실습	셀프 테스트	최종 산출물
Level 1	60분	60분	15분	코드 구조 지도, entrypoint·dependency 추적표, 설정 inventory, 코드 탐색 기록

`src/index.js`처럼 보이는 파일이 언제나 시작점은 아닙니다. `package.json` script가 다른 file을 실행할 수 있고, container의 `CMD`와 배포 YAML이 다시 command를 바꿀 수 있습니다. `PASSING_SCORE=79`도 number 79가 아니라 먼저 text `'79'`로 들어옵니다. 코드를 읽는다는 것은 file을 많이 여는 일이 아니라 **실제 command와 value가 어떤 경로로 behavior가 되는지 증거로 잇는 일**입니다.

그림 1. 코드는 기준·명령·진입·구조·흐름·경계·설정·증거의 여덟 gate로 읽습니다. 파일 목록은 이 경로를 찾기 위한 후보 지도입니다.


1
기준
root·version
2
명령
manifest·script
5
흐름
call·data
6
경계
file·network



nport → call·data → boundary·config → evidence

0. 이 PDF를 공부하는 방법

1회차 · 그림만 읽기 · 20분

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

```
실행 command를 먼저 기록한다.
manifest의 script value에서 executable·entry file·argument를 나눈다.
한 repository에는 CLI·server·test·deploy entrypoint가 따로 있을 수 있다.
import graph는 의존 가능성이고 call flow는 실제 실행 순서다.
call flow와 data flow를 분리한다.
file·environment·HTTP·stdout은 I/O boundary다.
설정 우선순위는 application code와 공식 문서로 확인한다.
환경 변수는 먼저 string 또는 undefined다.
effective value와 selected source를 함께 남긴다.
test가 증명하지 않은 deploy gap을 적는다.
```

2회차 · 실습 project 만들기 · 10분

코드 읽기 실습 생성기를 실행합니다. 기존 folder가 있으면 exit code 2로 중단하며 덮어쓰지 않습니다.

```
./02_Labs/G06_Git/L06-02_create-code-reading-project.sh
```

생성기는 외부 package를 설치하지 않습니다. Node.js built-in API만 사용하는 26개 file의 작은 service를 만듭니다.

3회차 · 12개 실제 경로 읽기 · 40분

코드 경로 탐색기를 열어 12개 증거를 순서대로 봅니다. 해설을 열기 전에 다음 다섯 값을 말합니다.

```
exact command =
entrypoint =
input shape·type =
first side effect =
output·open question =
```

4회차 · 내 repository 탐색 · 65분

코드 구조·설정 탐색 기록을 채웁니다. 낯선 말은 코드 구조·설정 용어집에서 확인합니다. 처음에는 관찰과 syntax check까지만 진행하고, install·build·test·server 실행은 script와 side effect를 읽은 뒤 승인된 환경에서 수행합니다.

1. 먼저 repository·runtime 기준을 고정합니다

1.1. Git 기준을 이어받습니다

M06-01에서 만든 repository identity를 다시 기록합니다.

```
git rev-parse --show-toplevel
git rev-parse --verify HEAD
git status --short --branch
```

기준	이유
top-level	다른 상위 repository를 잘못 읽지 않음
full HEAD OID	움직이는 branch 이름 대신 exact source state 고정
status	분석 중 local file이 baseline과 다른지 확인
capture 시각	environment·deploy·dependency 정보의 관찰 시점 표시

1.2. runtime version을 기록합니다

```
node --version
npm --version
```

실습 검증 기준은 Node.js `v24.14.0` 입니다. 2026-07-15 공식 Node.js 24 LTS 문서는 24.18.x 계열을 확인했습니다. version이 다르면 module resolution, CLI flag, test output, built-in API가 달라질 수 있습니다.

```
repository root =
HEAD full OID =
working tree status =
Node version =
npm version =
OS·architecture =
capture time·timezone =
```

1.3. 실행 전 신뢰 경계

source를 읽는 것과 source를 실행하는 것은 다릅니다.

- `package.json` script는 shell command를 실행할 수 있습니다.
- test도 application code이며 file·network·credential에 접근할 수 있습니다.
- install lifecycle은 dependency code를 실행할 수 있습니다.
- build는 generated file을 만들거나 외부 service를 호출할 수 있습니다.
- server는 port를 열고 database·queue·third-party API에 연결할 수 있습니다.
- untrusted repository의 local config·hook·tool configuration도 실행에 영향을 줄 수 있습니다.

30초 확인 1
`npm test`라는 이름만 보고 안전한 read-only command라고 말할 수 있습니까? 실행 전 어떤 file과 side effect를 확인해야 합니까?

2. 코드는 command에서 output까지 한 줄로 읽습니다

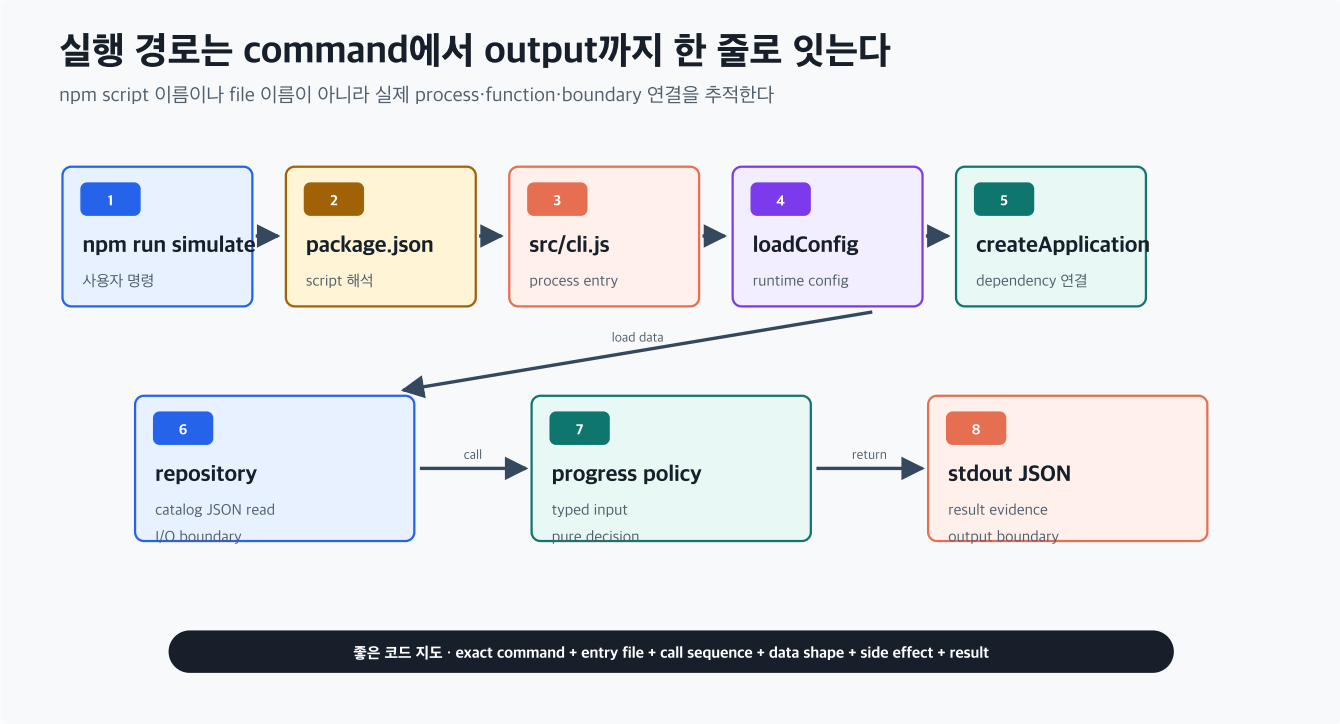
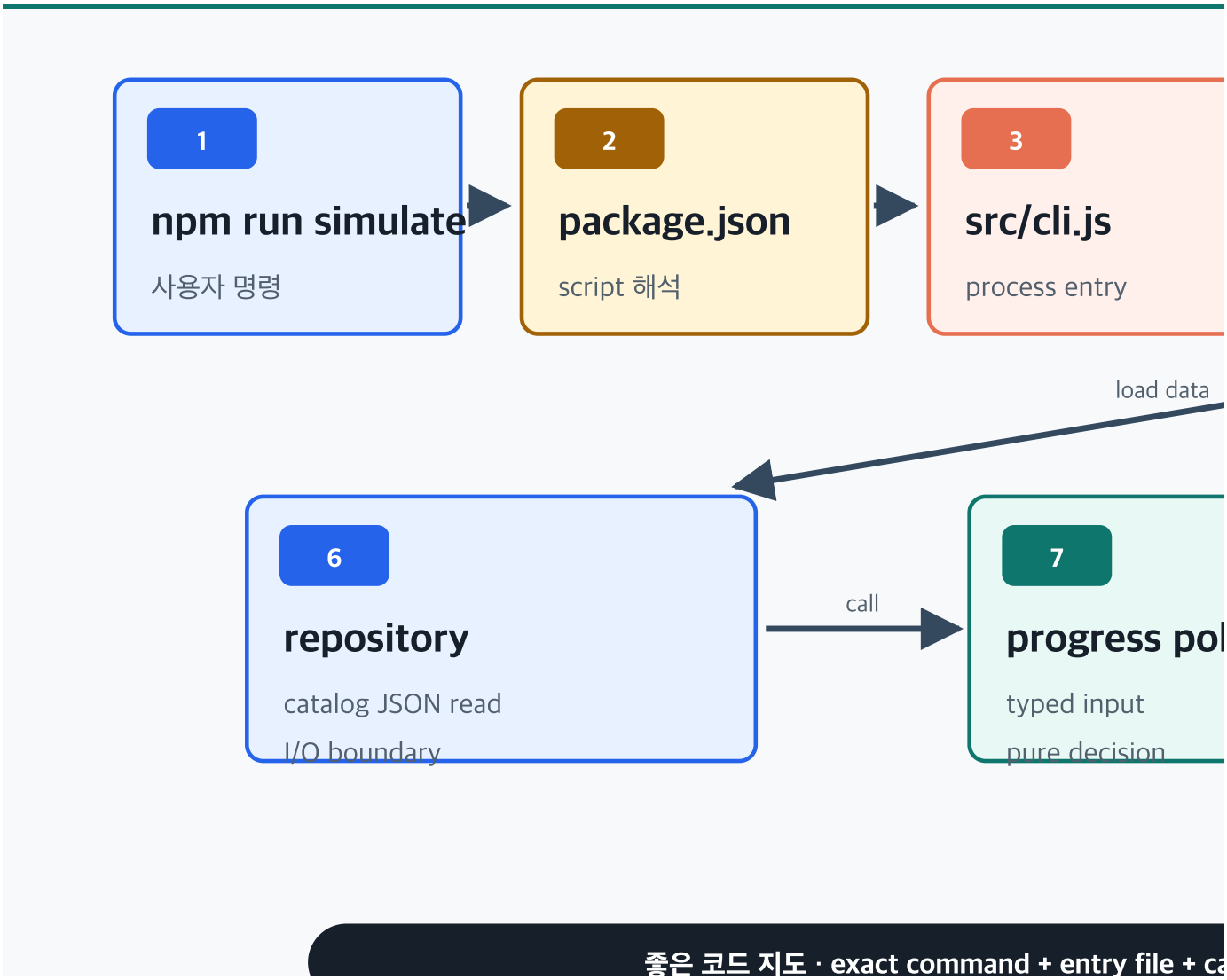
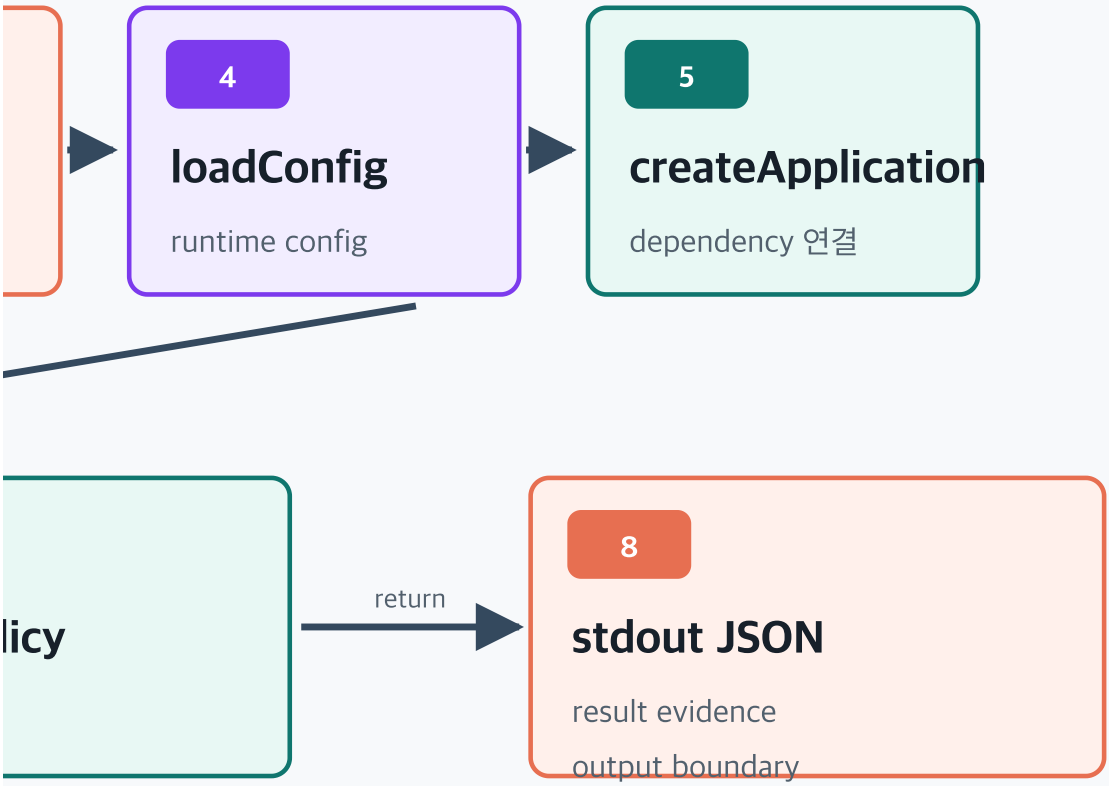


그림 2. 실행 경로의 시작은 눈에 띄는 함수가 아니라 실제 command입니다. process entry·config·composition·boundary·policy·output을 끊기지 않게 연결합니다.


1
npm run simulate
사용자 명령
2
package.json
script 해석
3
src/cli.js
process entry
load data
6
repository
catalog JSON read
I/O boundary
call
7
progress pol
typed input
pure decision
좋은 코드 지도 · exact command + entry file + ca



all sequence + data shape + side effect + result

실습의 한 경로는 다음과 같습니다.

```
node src/cli.js simulate --learner=planning --score=74
  → Node process
  → src/cli.js main()
  → loadConfig()
  → createApplication()
  → catalogService.simulate()
  → repository.loadCatalog()
  → evaluateProgress()
  → JSON.stringify()
  → stdout
```

각 화살표마다 다음 표를 채웁니다.

단계	질문	실습 답
command	정확히 무엇을 실행했는가	`node src/cli.js simulate --learner=planning --score=74`
process	어느 runtime·version인가	Node.js 24.14.0
entry	첫 user code file은	`src/cli.js`
input	처음 들어온 type은	argv string array
config	threshold의 effective value·source는	75·file, 80·file
data	learner record는 어디서 왔는가	`data/catalog.json`
decision	어느 function이 판정하는가	`evaluateProgress()`
output	무엇이 관찰되는가	stdout JSON, exit code

실행 경로와 설명 경로

실행 경로는 runtime이 실제로 밟은 순서입니다. 설명 경로는 학습자가 이해하기 쉽게 정리한 순서일 수 있습니다. 둘을 섞지 않습니다. `package.json → cli.js → policy.js`라는 설명은 중간 config·repository call을 생략할 수 있으므로, 증거서에는 생략 여부를 표시합니다.

3. project tree를 책임 지도로 바꿉니다

그림 3. directory name은 단서입니다. manifest·import·call·deploy evidence가 실제 책임을 확인합니다.

1

package.json

명령·module·engine

2

src/

runtime source

5

test/

behavior evidence

6

scripts/

project inspection

3

config/

file configuration

4

data/

catalog input

7

deploy/

runtime contract

8

Dockerfile

image entry

3.1. 실습 tree

```
repo/
├── package.json
├── package-lock.json
├── config/
│   ├── minimal.json
│   ├── service.json
│   └── pilot.json
├── data/catalog.json
├── deploy/service.yaml
├── Dockerfile
├── scripts/project-facts.js
├── src/
│   ├── cli.js
│   ├── server.js
│   ├── app.js
│   ├── config/
│   ├── catalog/
│   ├── http/
│   └── shared/
└── test/
```

3.2. 책임을 검증하는 세 증거

후보	이름이 암시하는 것	확인할 증거
`src/`	runtime source	command·import·build input
`config/`	file configuration	누가 parse하며 우선순위가 무엇인지
`data/`	input data	runtime read path·schema·owner
`test/`	behavior evidence	runner·assertion·test double·coverage gap
`scripts/`	개발·운영 도구	manifest script·side effect·permission
`deploy/`	배포 contract	platform parser·command·env·secret reference
`Dockerfile`	image recipe	base image·copy·user·CMD
`package-lock.json`	resolved dependency tree	lockfile version·actual install policy

폴더가 `service`라고 해서 독립 deploy unit은 아닙니다. `utils`라고 해서 risk가 낮지도 않습니다. 많은 module이 import하는 작은 shared function은 blast radius가 클 수 있습니다.

3.3. project facts를 structured parser로 읽기

실습은 `package.json`을 text grep이 아니라 `JSON.parse`로 읽습니다.

```
node scripts/project-facts.js
```

```
{
  "node": "v24.14.0",
  "package": "yeoncore-code-reading-lab@0.2.0",
  "private": true,
  "moduleSystem": "module",
  "engines": { "node": ">=20.0.0" },
  "declaredRuntimeDependencies": []
}
```

`declaredRuntimeDependencies: []`는 외부 runtime dependency 선언이 없다는 뜻이지, built-in module·OS·file·network dependency가 없다는 뜻이 아닙니다.

4. manifest에서 command·entrypoint matrix를 만듭니다

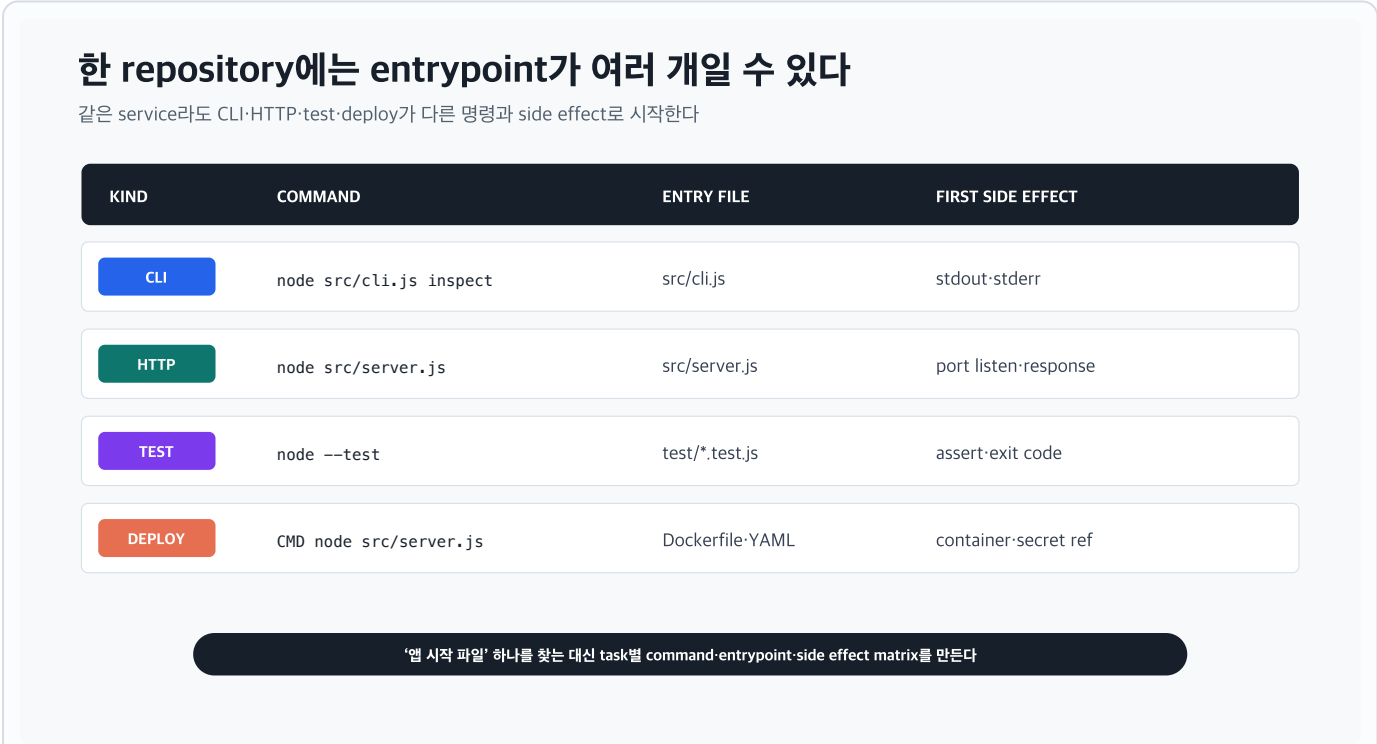


그림 4. 한 repository의 시작점은 하나가 아닐 수 있습니다. task마다 command·entry file·first side effect를 따로 기록합니다.

KIND	COMMAND	ENT
CLI	`node src/cli.js inspect`	src/
HTTP	`node src/server.js`	src/
TEST	`node --test`	test
DEPLOY	`CMD node src/server.js`	Doc

‘앱 시작 파일’ 하나를 찾는 대신 task별 comma

TRY FILE	FIRST SIDE EFFECT
'cli.js	stdout·stderr
'server.js	port listen·response
:/*.test.js	assert·exit code
:kerfile·YAML	container·secret ref

and·entrypoint·side effect matrix를 만든다

4.1. package.json scripts

npm 공식 package.json 문서는 `scripts`를 lifecycle event와 실행 command의 사전으로 설명합니다. 이름은 label이고 value가 실제 command입니다.

```
"scripts": {
  "facts": "node scripts/project-facts.js",
  "inspect": "node src/cli.js inspect",
  "simulate": "node src/cli.js simulate --learner=planning",
  "start": "node src/server.js",
  "check": "node --check src/cli.js && node --check src/server.js",
  "test": "node --test"
}
```

script	executable	entry·target	argument	first effect
facts	node	`scripts/project-facts.js`	없음	package JSON read·stdout
inspect	node	`src/cli.js`	`inspect`	config JSON read
simulate	node	`src/cli.js`	command·learner	config·catalog read
start	node	`src/server.js`	없음	local TCP listen
check	node + shell `&&`	cli·server source	`--check`	syntax parse
test	node	test discovery	`--test`	test code execution

`npm run`은 package root에서 script를 실행하지만, script value는 POSIX에서 `/bin/sh`, Windows에서 `cmd.exe` 같은 shell을 거칠 수 있습니다. quoting·environment·operator behavior는 platform에 따라 달라질 수 있습니다.

4.2. Dockerfile과 deploy YAML

```
CMD ["node", "src/server.js"]
```

```
runtime:
  command: ["node", "src/server.js"]
```

source repository의 `npm start`와 production image의 `CMD`, deployment platform의 override가 항상 같다고 가정하지 않습니다.

```
task = local CLI / local server / test / build / production deploy
declared command =
effective command =
entry file =
runtime version·image =
working directory =
first side effect =
evidence source =
```

5. ESM import·export graph를 읽습니다

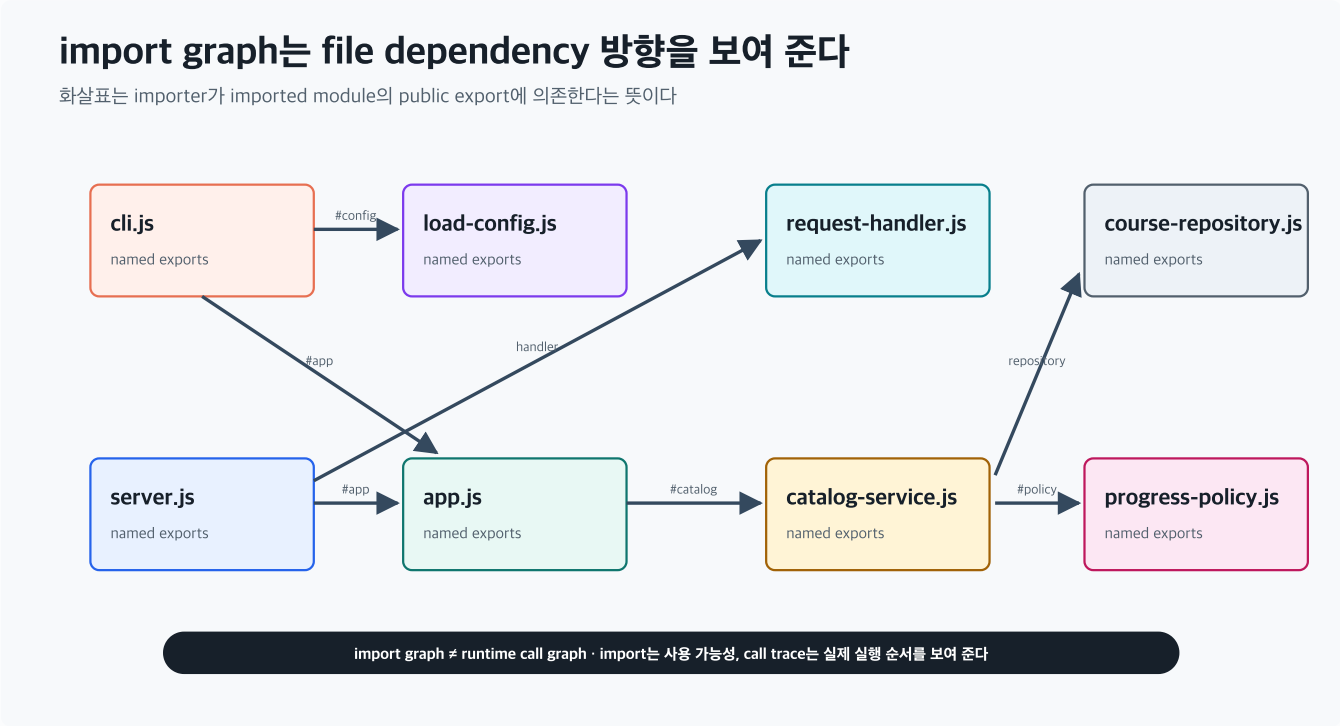
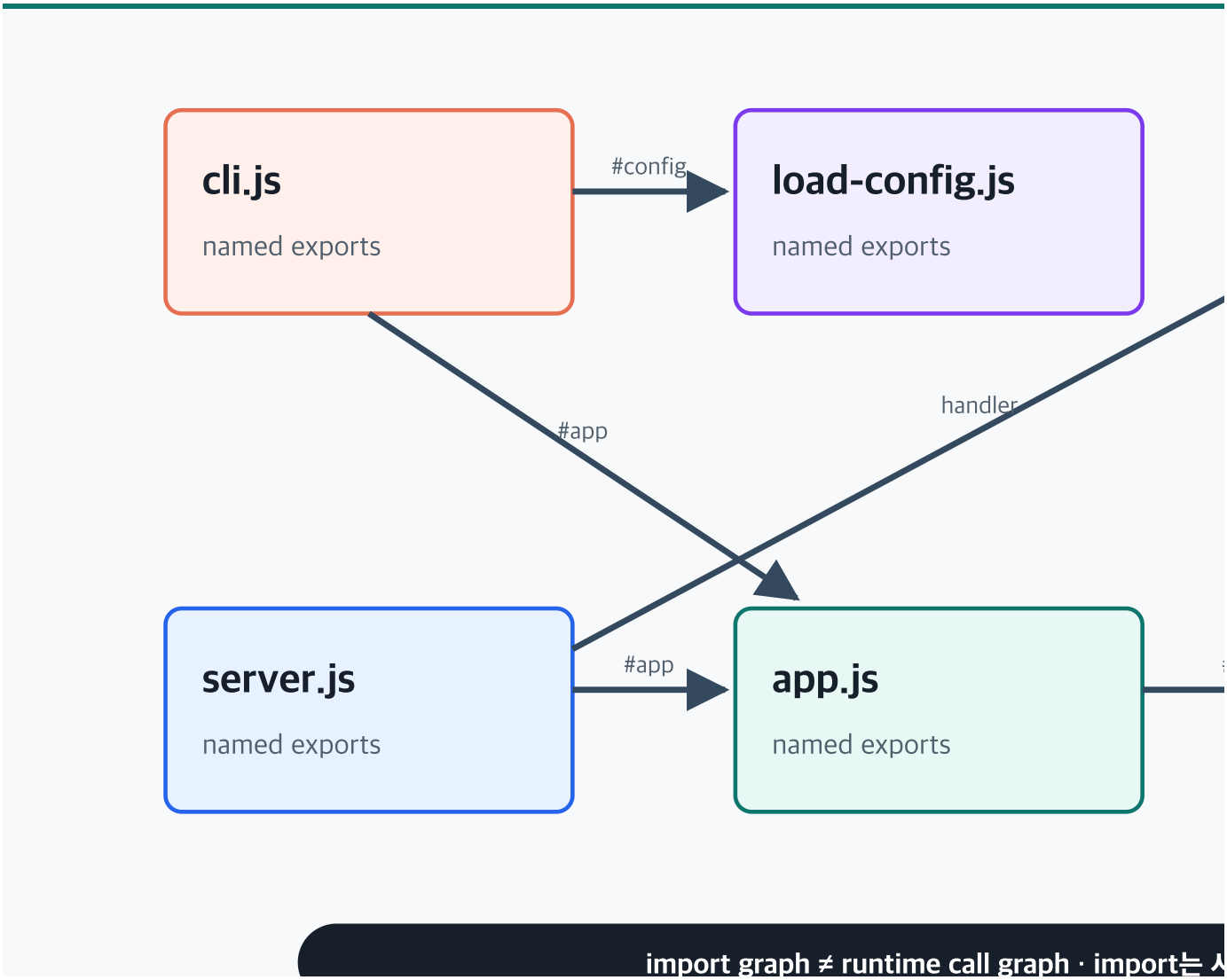
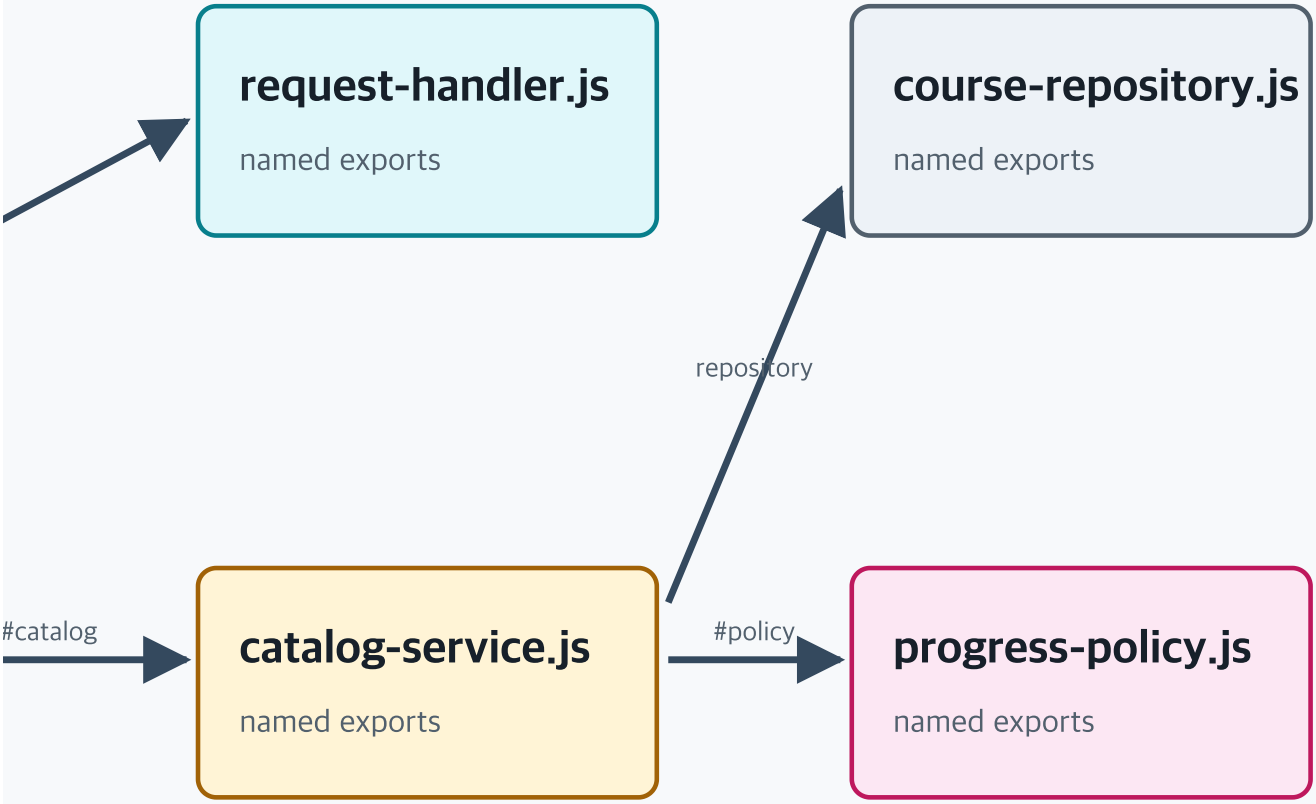


그림 5. import arrow는 importer가 imported module의 public export에 의존한다는 뜻입니다. 실제 function 호출 순서와는 다릅니다.


cli.js
named exports
#config
load-config.js
named exports
#app
handler
server.js
named exports
#app
app.js
named exports
import graph ≠ runtime call graph · import는 시



사용 가능성, call trace는 실제 실행 순서를 보여 준다

5.1. module system marker

Node.js package 문서에서 nearest parent `package.json`의 다음 field는 `.js`를 ESM으로 해석하게 합니다.

```
{
  "type": "module"
}
```

marker	Node.js 해석
`.mjs`	ESM
`.cjs`	CommonJS
`.js` + nearest `type: module`	ESM
`.js` + nearest `type: commonjs`	CommonJS

명시 marker가 없는 ambiguous file은 current Node가 syntax detection을 할 수 있지만, package author는 module type을 명시하는 편이 tool·loader·미래 변경에 더 분명합니다.

5.2. import와 export

```
// importer
import { evaluateProgress } from '#policy';

// imported module public export
export function evaluateProgress(input, config) {
  // ...
}
```

관찰	질문
import specifier	어떤 resolver 규칙을 쓰는가
named·default import	어떤 public symbol을 기대하는가
export	외부 module에 허용한 surface는 무엇인가
top-level code	import 순간 실행되는 side effect가 있는가

관찰	질문
dynamic import	static graph에서 빠질 path가 있는가
cycle	A→B→A 초기화 순서가 문제가 되는가

5.3. 실습 import inventory

```
cli.js     → #app, #config
server.js  → node:http, #app, #config, request-handler.js
app.js     → #catalog, course-repository.js, project-path.js
catalog-service.js → #policy
course-repository.js → node:fs/promises
load-config.js → node:fs/promises, node:util, schema.js, project-path.js
```

`node:`는 Node built-in module을 명시합니다. external package가 없어도 file system·HTTP·process·OS에 의존할 수 있습니다.

6. specifier별 module resolution을 구분합니다

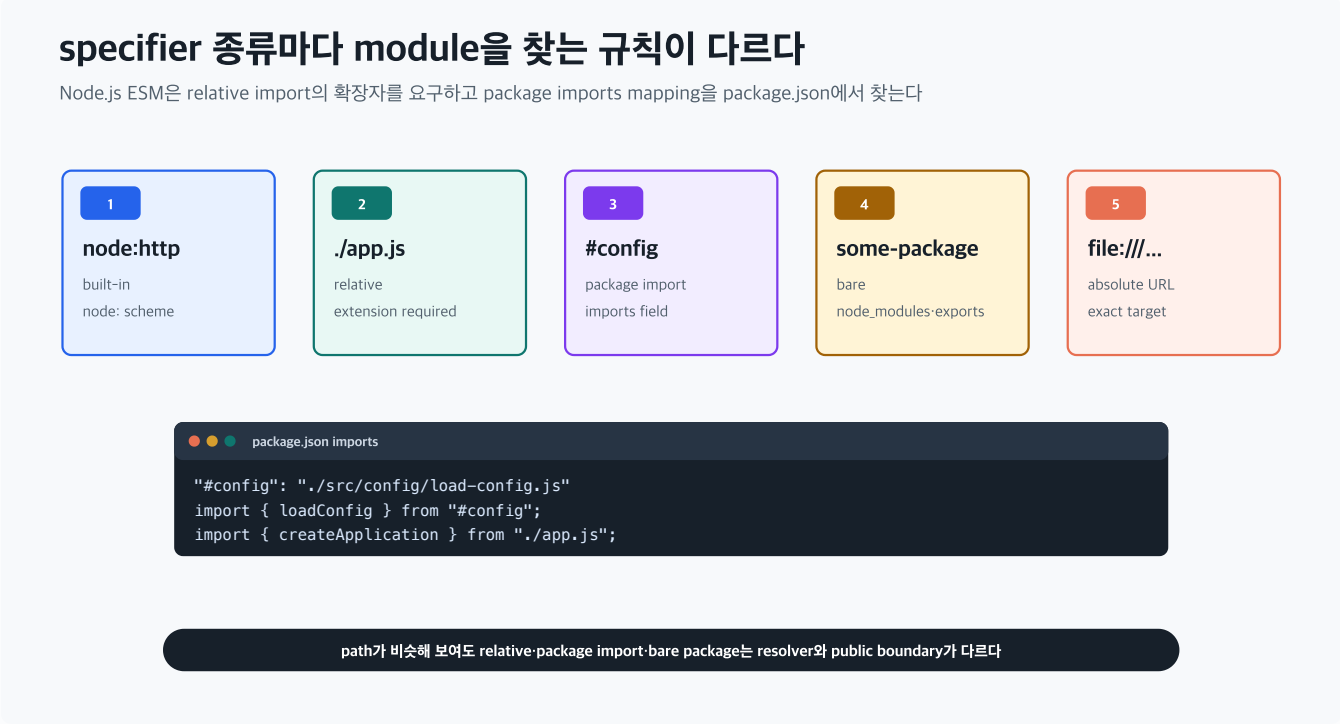


그림 6. 문자열 모양에 따라 built-in·relative·package imports·bare package·absolute URL의 resolver가 달라집니다.

1

node:http

built-in

node: scheme

2

./app.js

relative

extension required

3

#config

package impo

imports field

package.json imports

```
"#config": "./src/config/load-config.js"
import { loadConfig } from "#config";
import { createApplication } from "./app.js";
```

path가 비슷해 보여도 relative·package import·bar

ort

4

some-package

bare

node_modules·exports

5

file:///...

absolute URL

exact target

e package는 resolver와 public boundary가 다르다

6.1. 다섯 specifier

예	종류	찾는 기준
`node:http`	built-in	Node.js built-in module
`./schema.js`	relative	importing file 기준 URL, extension 필요
`../shared/project-path.js`	relative parent	importing file 기준, exact path
`#config`	package imports	nearest package.json의 `imports` mapping
`some-package`	bare	package resolution·`exports`·node_modules
`file:///app/x.js`	absolute URL	exact file URL

Node.js ESM 공식 문서는 relative·absolute specifier에 file extension을 요구합니다. directory index도 완전한 path로 지정합니다.

6.2. package imports

```
"imports": {
  "#app": "./src/app.js",
  "#config": "./src/config/load-config.js",
  "#catalog": "./src/catalog/catalog-service.js",
  "#policy": "./src/catalog/progress-policy.js"
}
```

Node.js의 `imports` field는 package 내부 private mapping이며 entry key는 external package specifier와 구분되도록 `#`로 시작합니다. `#config`라는 이름만 보고 file 위치를 추측하지 않고 mapping을 확인합니다.

6.3. `exports`와 public boundary

library package의 `exports`는 consumer가 사용할 public entrypoint를 제한할 수 있습니다. 내부 file이 존재한다고 `pkg/private.js`를 import할 수 있는 것은 아닙니다. `exports` 도입은 기존 deep import를 막아 breaking change가 될 수 있습니다.

30초 확인 2

`import { loadConfig } from '#config'`를 봤습니다. 어느 file을 확인해야 target path를 증명할 수 있습니까? `#config.js`를 찾는 것으로 충분합니까?

7. import graph와 runtime call graph를 구분합니다

import graph가 답하는 질문

```
어느 module이 어느 module의 public symbol을 사용할 수 있는가?
```

call graph가 답하는 질문

```
이 command·input에서 어느 function이 실제로 다음 function을 호출했는가?
```

예를 들어 `cli.js`가 `#app`을 import해도 `inspect` command는 `app.simulate()`를 호출하지 않습니다. import edge는 존재하지만 그 실행 path에는 call edge가 없습니다.

차이	import graph	call graph
기준	source declaration	runtime control flow
node	module·file	function·method·callback
edge	import dependency	actual·possible call
입력 영향	비교적 적음	branch·event·data에 크게 의존
static 한계	dynamic import·generated code	reflection·framework callback·remote call

framework에서는 entry가 숨을 수 있습니다

route decorator, dependency injection container, plugin registry, event subscription, queue consumer, build-generated route는 direct call text가 없을 수 있습니다. 이때는 다음을 함께 봅니다.

- framework bootstrap command
- route·handler registry
- annotation·decorator metadata
- dependency container binding
- configuration-driven module list
- generated artifact·source map
- runtime trace·log·test

8. call flow와 data flow를 다른 색으로 그립니다

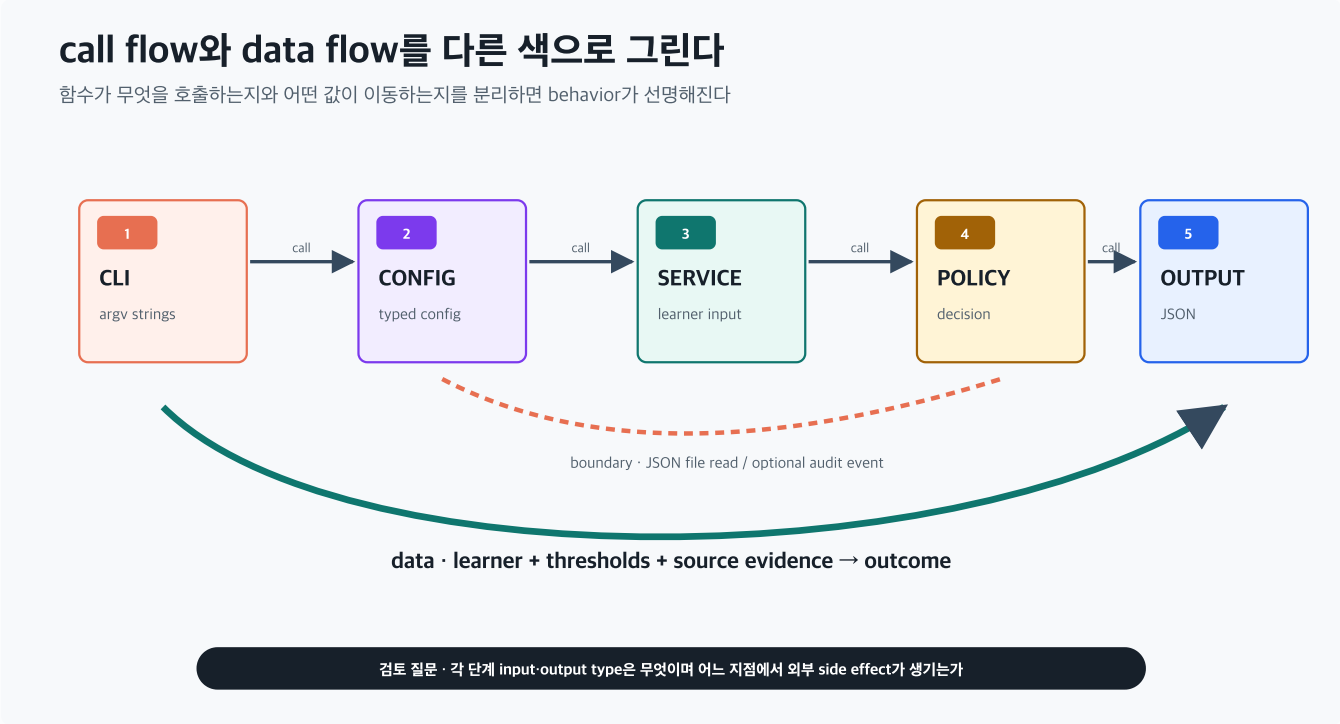
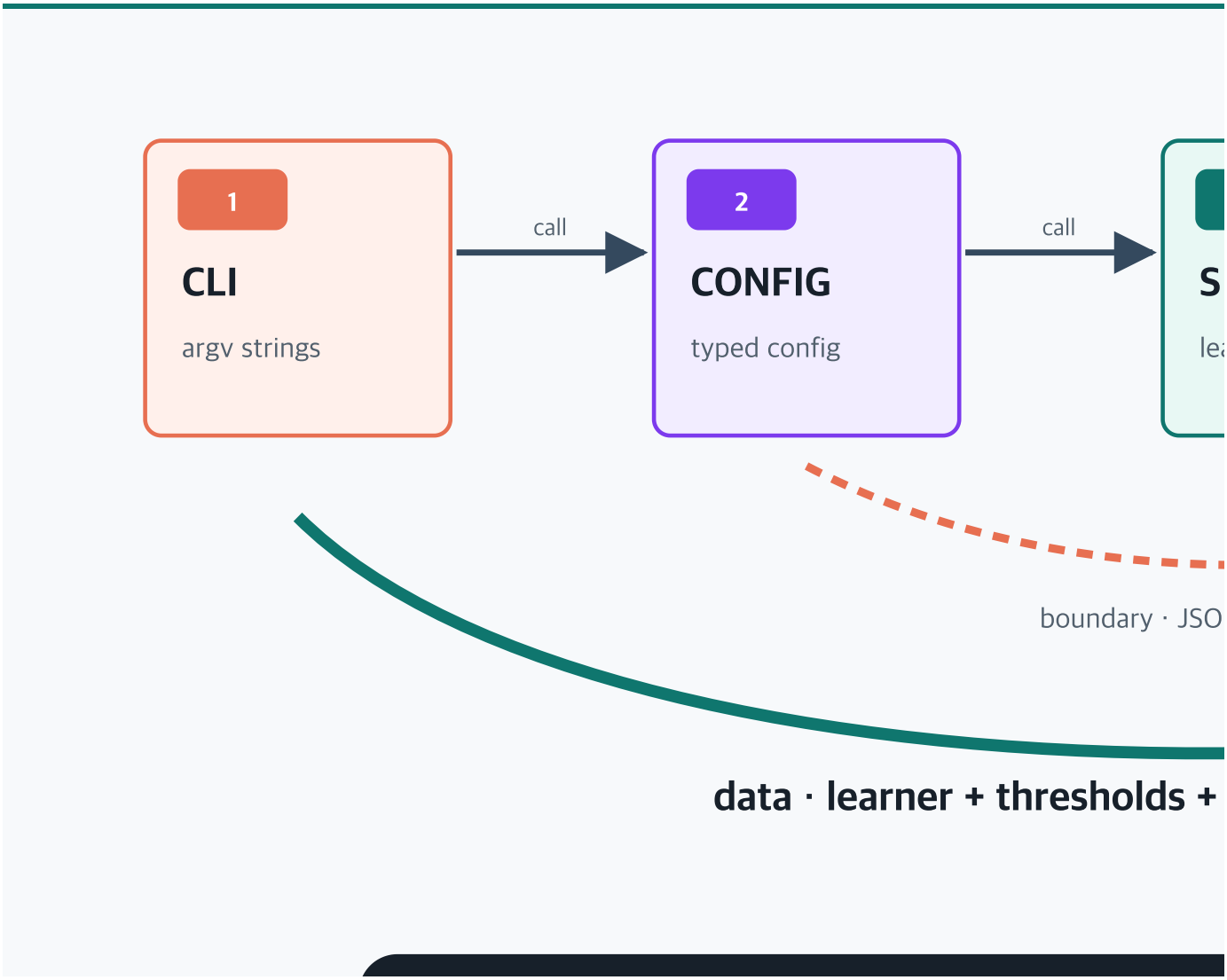
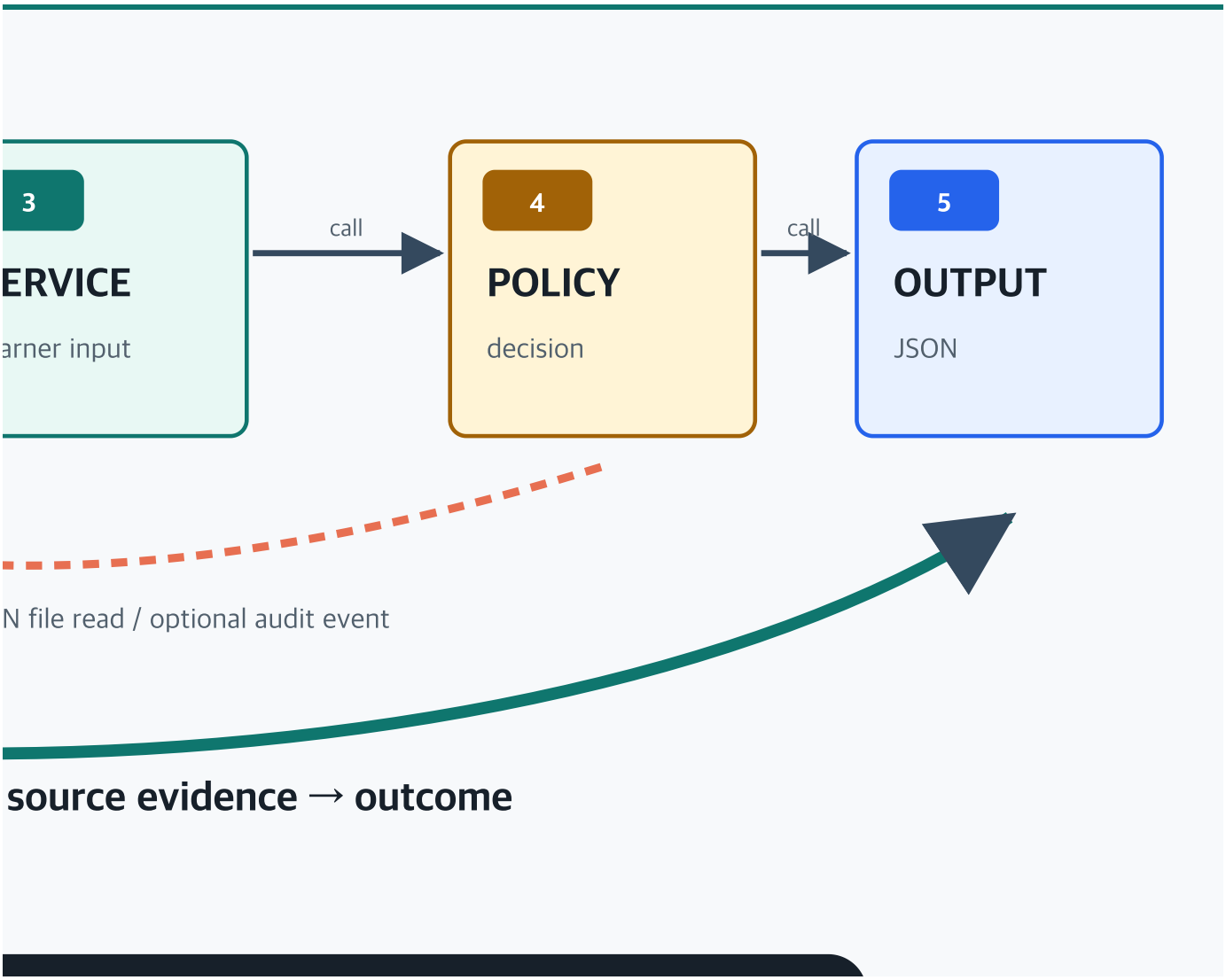


그림 7. function 호출 방향과 value 이동 방향을 분리합니다. 같은 단계라도 control과 data의 질문은 다릅니다.


1
CLI
argv strings
call
2
CONFIG
typed config
call
boundary · JSO
data · learner + thresholds +


3
ERVICE
arner input
call
4
POLICY
decision
call
5
OUTPUT
JSON
N file read / optional audit event
source evidence → outcome

8.1. call flow

```
main()
  → loadConfig()
  → createApplication()
  → catalogService.simulate()
  → repository.loadCatalog()
  → evaluateProgress()
  → console.log()
```

8.2. data flow

```
argv '--score=74'              string
  → parseRuntimeInput          number 74
config/service.json 75         JSON number
  → selected passingScore      number 75 · source file
catalog learner 8/10           JSON numbers
  → evaluateProgress input
  → progressPercent 80
  → scorePassed false
  → outcome IN_PROGRESS
  → JSON string stdout
```

8.3. 각 단계 data contract

stage	input	output	failure
parseArgs	string array	values·positionals	unknown option
choose	four optional values	value·source	none
normalize	raw mixed types	typed frozen config	invalid type·range
repository	file path	catalog object	read·JSON·shape error
policy	learner numbers·config	decision object	invalid count·score
CLI presenter	result object	JSON text·exit	serialization·stdout error

좋은 flow diagram에는 function name만 아니라 주요 data shape·type과 failure edge가 있습니다.

9. I/O boundary와 pure core를 나눕니다

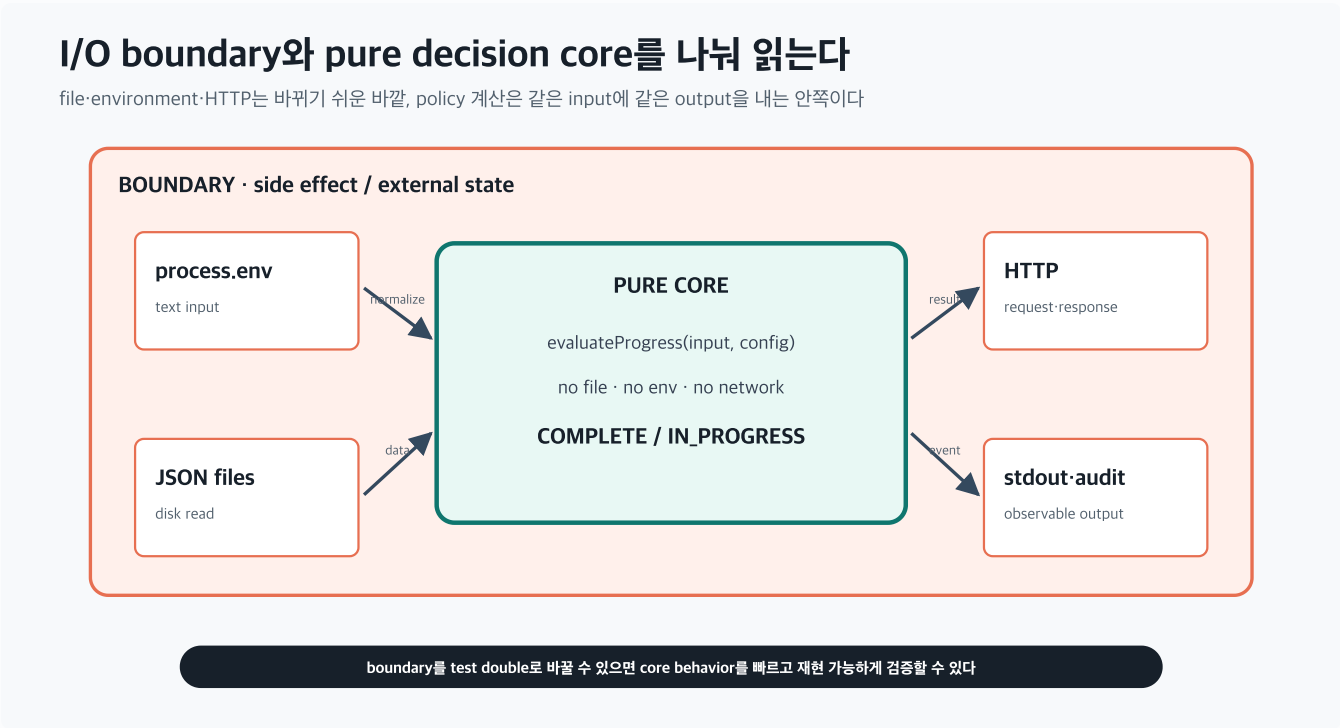
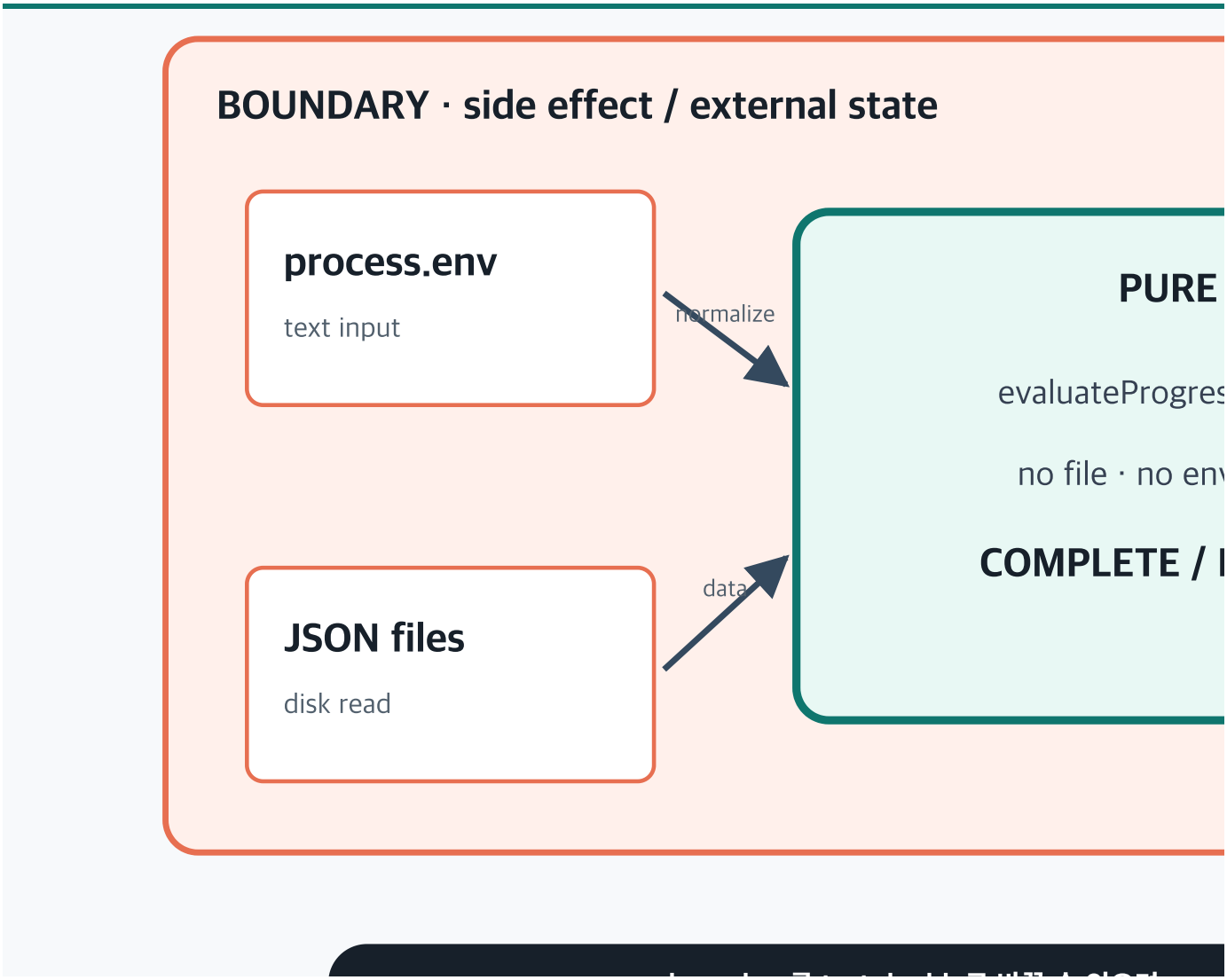
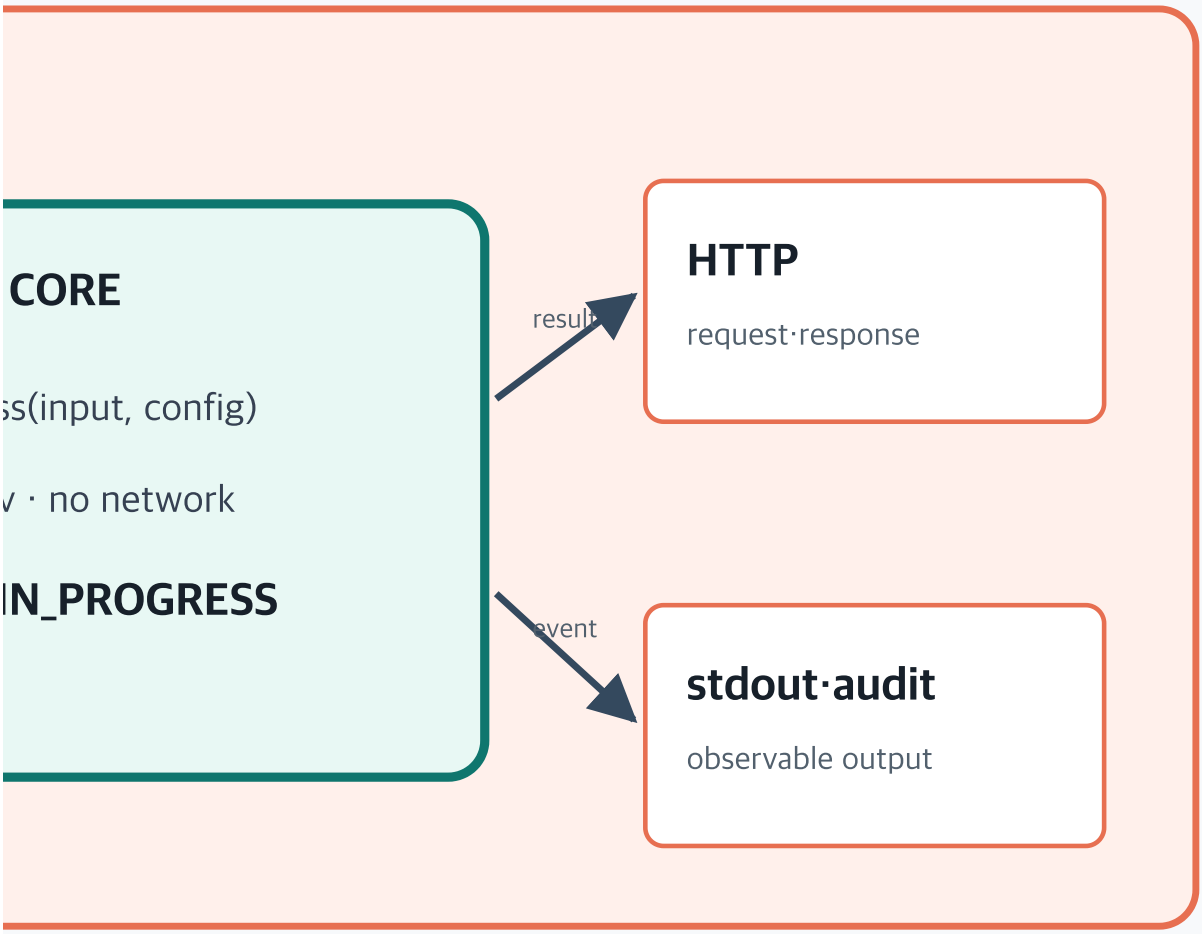


그림 8. 바깥의 file·environment·HTTP는 external state를 다루고, 안쪽의 pure policy는 같은 input에서 같은 output을 계산합니다.


BOUNDARY · side effect / external state
process.env
text input
normalize
JSON files
disk read
data
PURE
evaluateProgres
no file · no env
COMPLETE / I


CORE
ss(input, config)
v · no network
IN_PROGRESS
result
event
HTTP
request·response
stdout·audit
observable output

9.1. boundary inventory

boundary	module	effect	test strategy
environment	`load-config.js`	process state read	env object 주입
config file	`load-config.js`	file read·JSON parse	readText stub
catalog file	`course-repository.js`	file read·JSON parse	readText stub·sample
stdout	`cli.js`	observable text output	process execution capture
stderr audit	`cli.js` · `server.js`	event output	audit function 주입
HTTP listen	`server.js`	TCP port open	isolated port·cleanup
HTTP response	`request-handler.js`	status·header·body	handler·integration test

9.2. pure policy

```
export function evaluateProgress(input, config) {
  const progressPercent = Math.round(
    (input.completedLessons / input.totalLessons) * 100
  );
  const progressPassed = progressPercent >= config.completionPercent;
  const scorePassed = input.score >= config.passingScore;
  return {
    progressPercent,
    progressPassed,
    scorePassed,
    outcome: progressPassed && scorePassed ? 'COMPLETE' : 'IN_PROGRESS'
  };
}
```

이 function은 file·environment·network·current time을 읽지 않습니다. 따라서 threshold·rounding·boundary case를 빠르게 test할 수 있습니다.

9.3. pure가 무조건 좋은 것은 아닙니다

모든 function을 pure로 만들 필요는 없습니다. service는 결국 file·database·network·clock을 사용합니다. 중요한 것은 side effect가 어디서 시작하고 어떤 interface로 core에 전달되는지 보이는 것입니다.

10. 설정 우선순위를 source evidence로 읽습니다

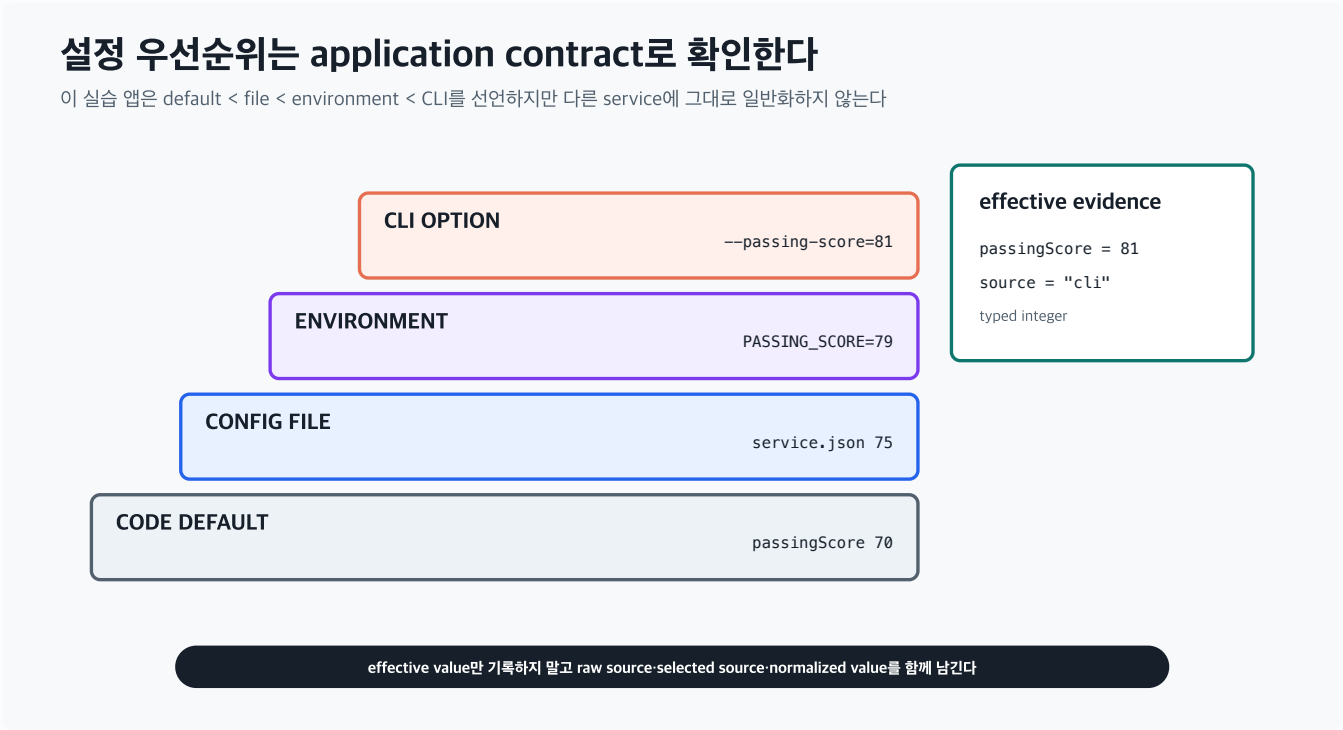
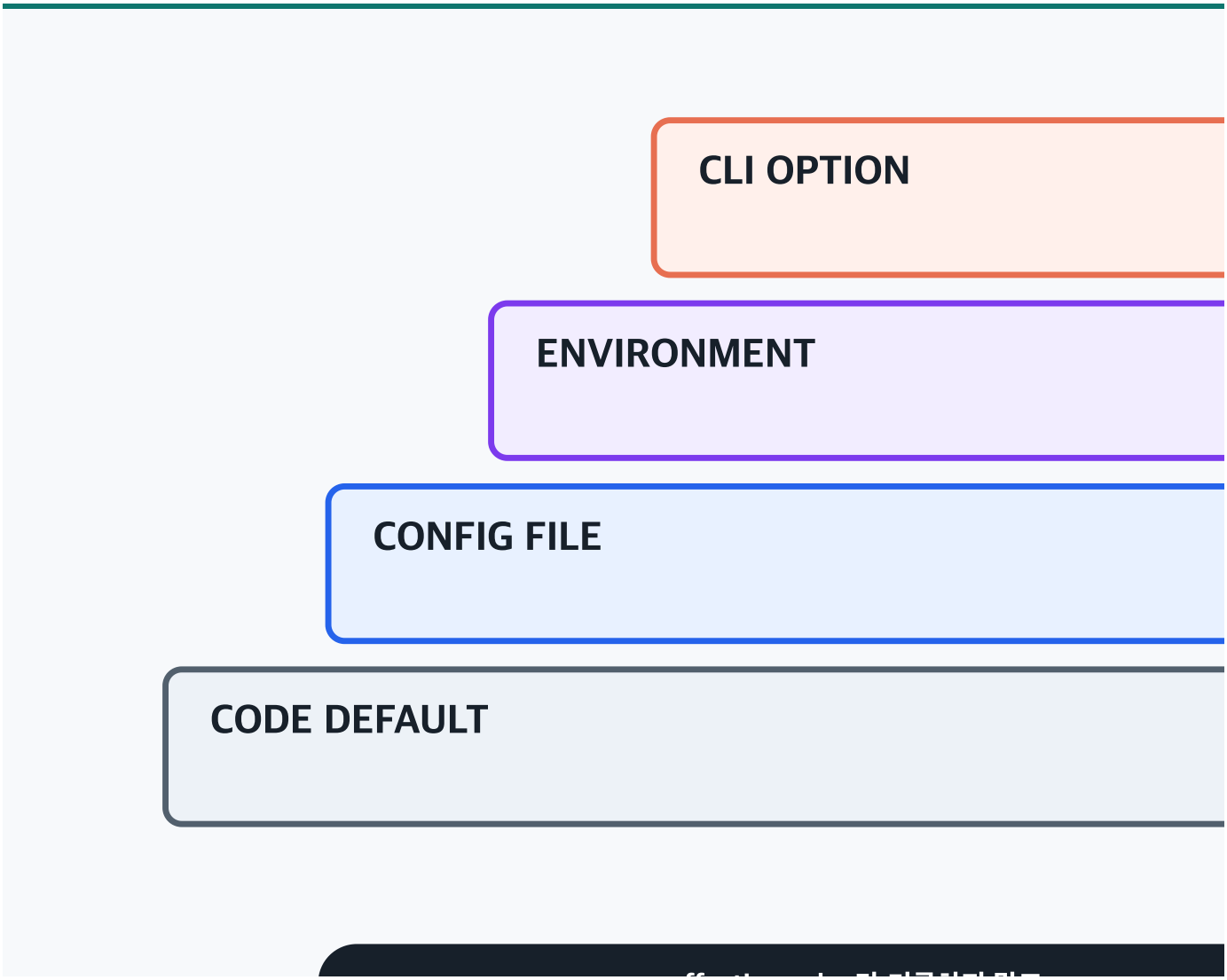
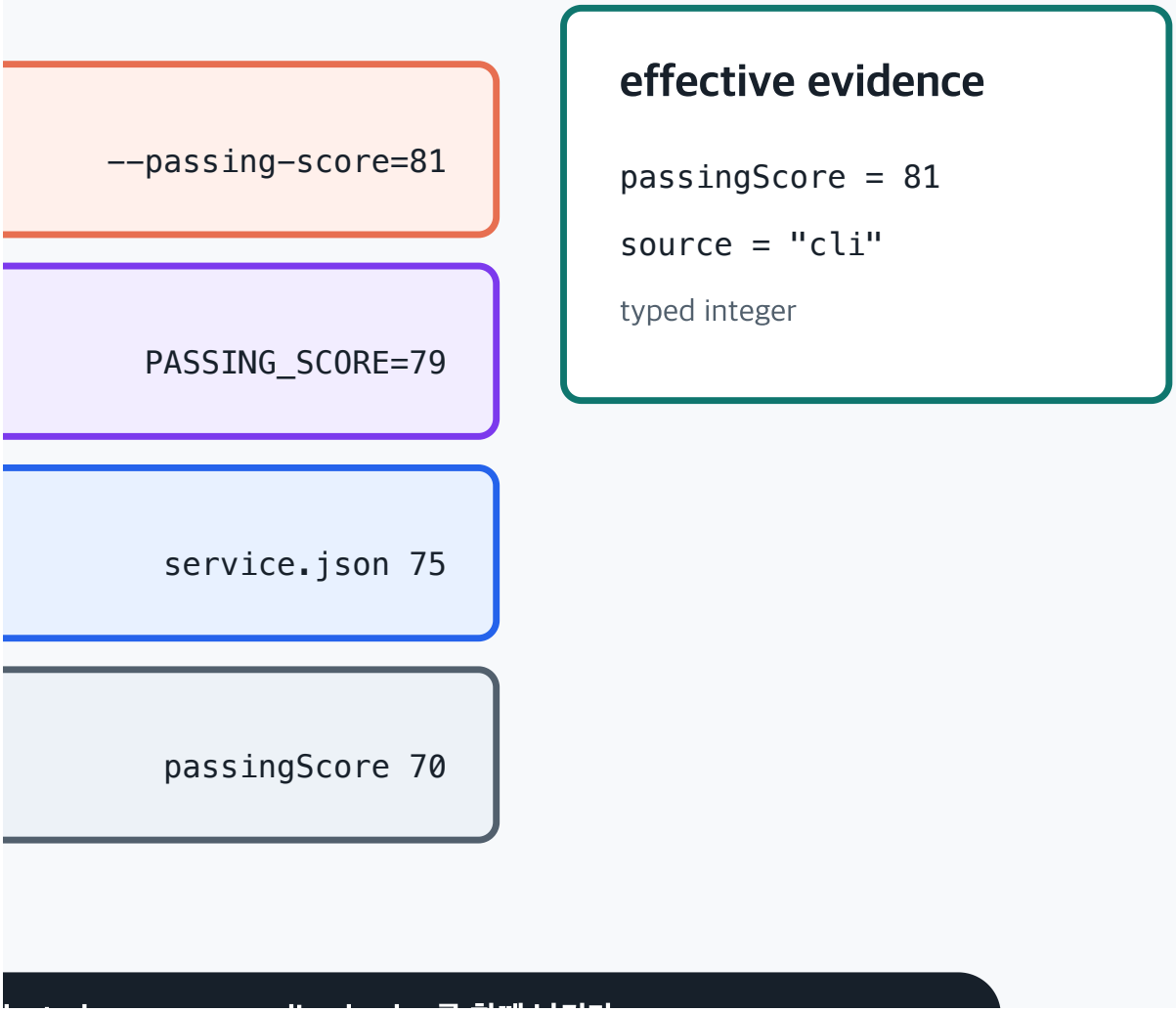


그림 9. 실습 앱은 네 source를 선언합니다. effective value만 아니라 selected source를 함께 반환합니다.


CLI OPTION
ENVIRONMENT
CONFIG FILE
CODE DEFAULT


--passing-score=81
PASSING_SCORE=79
service.json 75
passingScore 70
effective evidence
passingScore = 81
source = "cli"
typed integer

10.1. 이 실습 앱의 contract

```
낮은 우선순위                                    높은 우선순위
code default < selected config file < environment < CLI option
```

이 순서는 Node.js 전체의 보편 규칙이 아닙니다. 실습 `load-config.js`의 `choose()`가 정의한 application rule입니다.

```
function choose(cliValue, envValue, fileValue, defaultValue) {
  if (cliValue !== undefined) return { value: cliValue, source: 'cli' };
  if (envValue !== undefined) return { value: envValue, source: 'env' };
  if (fileValue !== undefined) return { value: fileValue, source: 'file' };
  return { value: defaultValue, source: 'default' };
}
```

10.2. 네 실제 결과

command·source	passingScore	source	config selector
`--config=config/minimal.json`	70	default	CLI가 file 선택
baseline `config/service.json`	75	file	default path
`PASSING_SCORE=79`	79	env	default path
env 79 + `--passing-score=81`	81	CLI	default path

`configSelectorSource`는 어떤 config file을 골랐는지 말하고, `sources.passingScore`는 그 file까지 읽은 뒤 effective score가 어느 layer에서 왔는지 말합니다. 서로 다른 질문입니다.

10.3. Node `--env-file`과 application precedence

Node.js 24.10부터 `--env-file`은 stable CLI option입니다.

```
node --env-file=.env src/cli.js inspect
```

Node CLI가 `.env`를 `process.env`에 채우는 규칙과 application이 `process.env`·JSON·CLI option 중 effective value를 고르는 규칙을 분리합니다.

- 동일 variable이 OS environment와 env file에 있으면 Node의 actual environment가 우선합니다.
- 여러 `--env-file`은 뒤 file이 앞 file 값을 덮을 수 있습니다.
- 그 뒤 application code가 `process.env`와 자체 CLI option을 다시 비교합니다.

```
OS env + Node --env-file merge
  → process.env strings
  → application choose(cli option, env, JSON, code default)
  → normalized typed config
```

11. raw 설정을 typed config로 정규화합니다

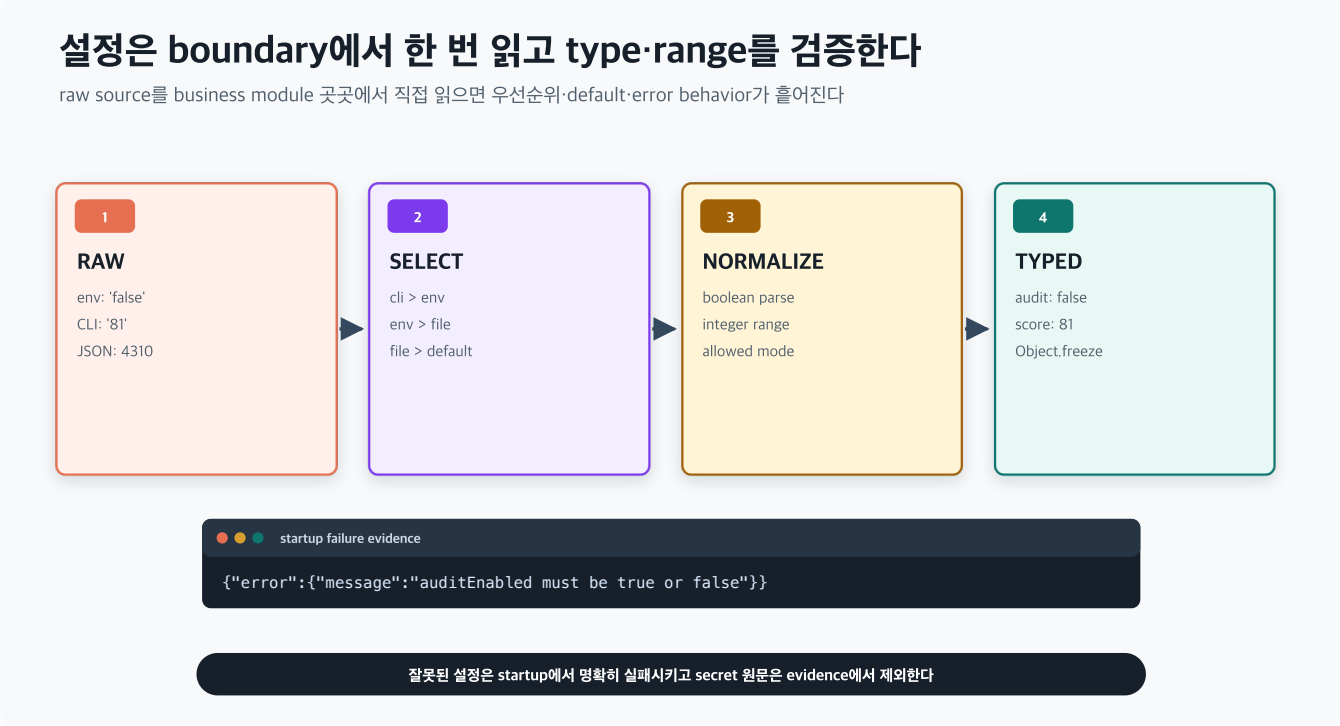
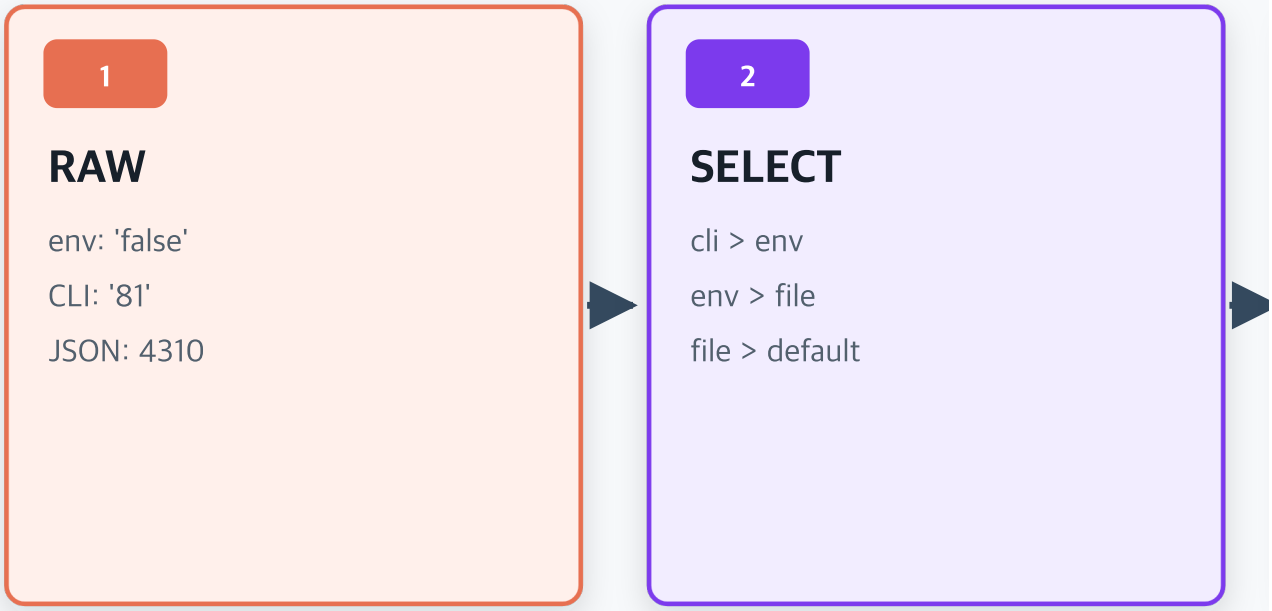
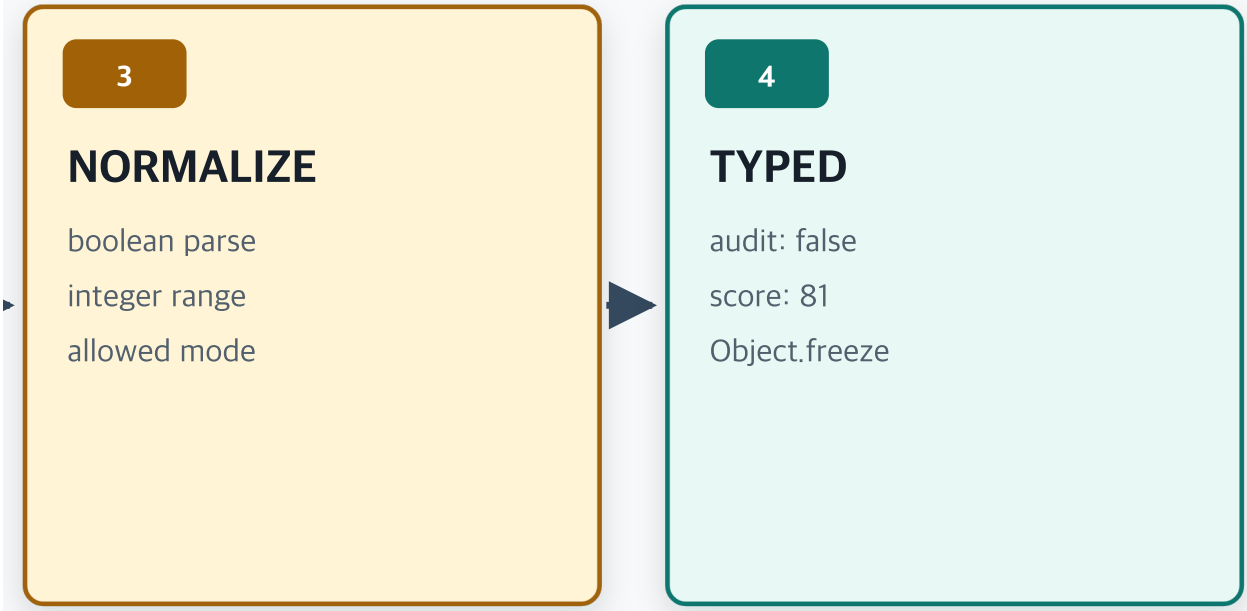


그림 10. raw source 선택 뒤 type·range·allowed value를 검증하고, business module에는 typed config 하나를 전달합니다.



startup failure evidence

```
{"error":{"message":"auditEnabled must be true o
```


3
NORMALIZE
boolean parse
integer range
allowed mode
4
TYPED
audit: false
score: 81
Object.freeze

```
r false"}}
```

11.1. 네 단계

```
RAW → SELECT SOURCE → NORMALIZE·VALIDATE → TYPED CONFIG
```

단계	해야 할 일	evidence
raw	original type·presence 확인	`'79'`, `undefined`, JSON 75
select	precedence 적용	`value=81, source=cli`
normalize	boolean·integer·enum·path 변환	`Number`, explicit parser
validate	range·allowed value·path boundary	0~100, allowed modes
typed	downstream single contract	frozen config object

11.2. startup failure

```
AUDIT_ENABLED=yes node src/cli.js inspect
```

```
{"error":{"message":"auditEnabled must be true or false"}}
```

exit code는 1입니다. invalid configuration을 silently coerce하거나 warning만 남기고 계속 실행하지 않습니다.

11.3. path boundary

실습 `resolveProjectPath()`는 config·catalog path를 project root 기준으로 resolve하고 `..`로 project 밖을 벗어나면 실패합니다.

```
relative candidate
  → absolute resolved path
  → project root와 relative 비교
  → outside면 error
```

이 방어가 모든 file security를 해결하지는 않습니다. symbolic link, permission, file ownership, race, sensitive content, container mount를 별도 검토합니다.

12. 환경 변수의 string 함정을 피합니다

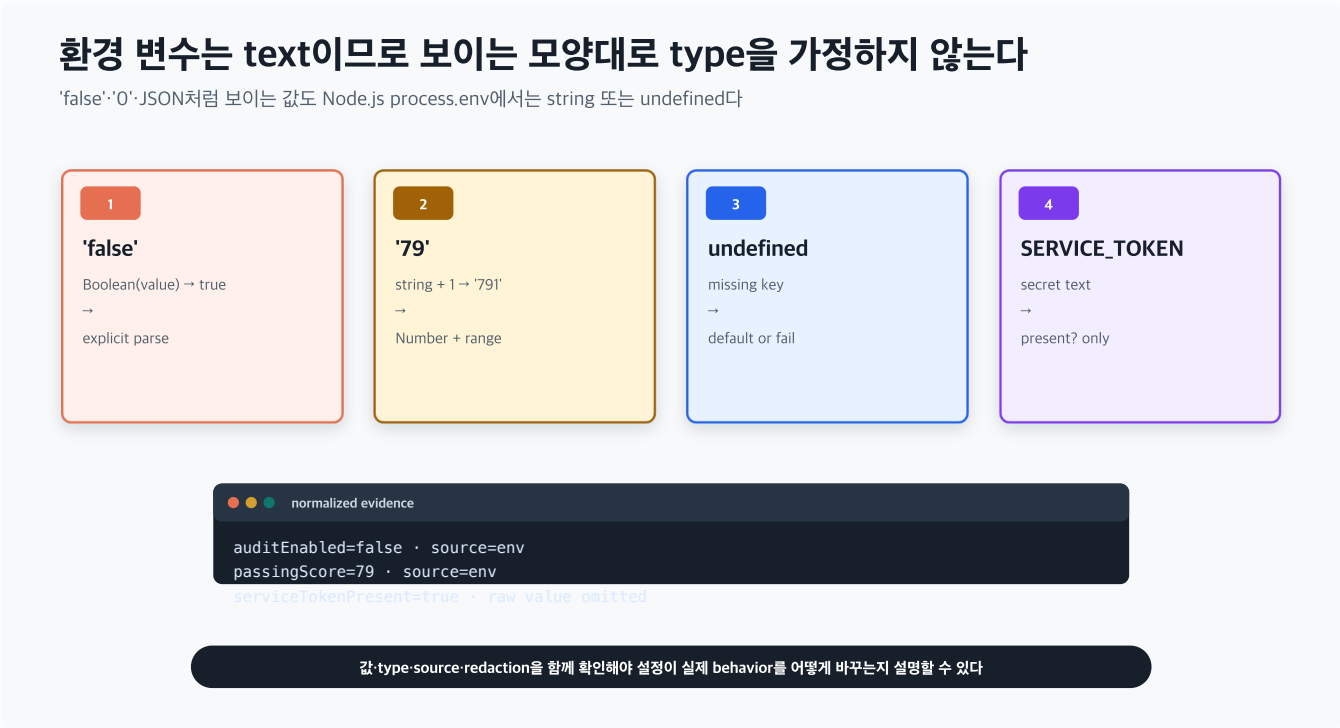


그림 11. 환경 변수는 text입니다. boolean·number처럼 보이는 값을 명시적으로 parse하고, secret은 존재 여부만 증거에 남깁니다.

1

'false'

Boolean(value) → true

→

explicit parse

2

'79'

string + 1 → '791'

→

Number + range

normalized evidence

```
auditEnabled=false · source=env
passingScore=79 · source=env
serviceTokenPresent=true · raw value omitted
```

3

undefined

missing key

→

default or fail

4

SERVICE_TOKEN

secret text

→

present? only

Node.js 공식 environment variable 문서는 `.env`의 `0`, `true`, JSON처럼 보이는 값도 Node 안에서 literal string이 된다고 설명합니다.

12.1. `false` 함정

```
Boolean('false') === true
```

비어 있지 않은 string이기 때문입니다.

```
function parseBoolean(value) {
  if (value === 'true') return true;
  if (value === 'false') return false;
  throw new Error('must be true or false');
}
```

12.2. number 함정

```
'79' + 1        // '791'
Number('79')    // 79
Number('x')     // NaN
```

변환 뒤 `Number.isInteger`, minimum·maximum을 확인합니다. `Number('')`는 0이 될 수 있으므로 empty input policy도 명시합니다.

12.3. missing과 empty

raw	의미 후보	필요한 contract
`undefined`	variable 없음	default / required error
`''`	빈 값	허용 / missing 취급 / error
`'0'`	text zero	number 0이 유효한가
`'false'`	text false	explicit boolean parse
whitespace	사용자 입력 오류	trim 여부·empty 판정

12.4. secret redaction

실습은 `SERVICE_TOKEN` 원문을 output에 넣지 않습니다.

```
{
  "serviceTokenPresent": true
}
```

존재 여부조차 보안 정보가 될 수 있는 환경에서는 그 field도 제한합니다. 실제 secret은 source·PDF·screenshot·log·ticket에 붙이지 않습니다.

13. 설정 파일 형식과 behavior를 구분합니다

13.1. JSON

RFC 8259는 JSON을 structured data를 위한 text format으로 정의합니다. object·array·string·number·boolean·null을 표현합니다.

```
{
  "mode": "study",
  "passingScore": 75,
  "auditEnabled": false
}
```

- 표준 JSON에는 comment syntax가 없습니다.
- trailing comma를 허용하지 않습니다.
- object member order에 behavior를 의존하지 않습니다.
- duplicate member name은 parser 간 상호운용 문제가 있으므로 허용하지 않습니다.
- parse 성공은 schema·business validity를 증명하지 않습니다.

13.2. `.env`

```
APP_MODE=pilot
PASSING_SCORE=79
AUDIT_ENABLED=false
```

`.env`는 typed schema가 아니라 environment variable text를 표현합니다. `.env.example`은 name·sample을 공유할 수 있지만 실제 credential을 담지 않습니다.

13.3. YAML

```
environment:
  PASSING_SCORE: "75"
  AUDIT_ENABLED: "true"
secretReferences:
  SERVICE_TOKEN: course-service-token
```

YAML file이 값을 선언해도 실제 platform이 어떤 schema·default·merge rule로 해석하는지 공식 문서가 필요합니다. `"75"`를 quote한 이유, secret reference가 실제 secret injection으로 연결되는지 확인합니다.

13.4. package.json과 lockfile

file	주 질문
package.json	name·version·type·engines·scripts·dependencies·imports는
package-lock.json	어떤 resolved dependency tree를 고정하는가
`.npmrc`	registry·auth·install·script policy에 무엇을 적용하는가

실습 lockfile에는 external dependency가 없습니다. 실제 project에서는 manifest intent와 resolved lock delta를 함께 봅니다.

13.5. configuration inventory

key	source names	raw type	normalized type	default	allowed·range	secret	consumer
mode	`mode` · `APP_MODE` · `--mode`	string	enum string	study	study·pilot·production		app·health
passingScore	JSON· `PASSING_SCORE` · `--passing-score`	number/string	integer	70	0~100		policy
completionPercent	JSON·env·CLI	number/string	integer	80	1~100		policy
auditEnabled	JSON·env·CLI	boolean/string	boolean	false	true·false		service

key	source names	raw type	normalized type	default	allowed·range	secret	consumer
port	JSON· `PORT` · `--port`	number/string	integer	4310	1024~65535		server
catalogPath	JSON·env·CLI	string	project path	data/catalog.json	inside project		repository
serviceToken	`SERVICE_TOKEN`	string	presence·secret handle	none	deployment policy	✓	external auth 예정

14. source·build·runtime·deploy를 분리합니다

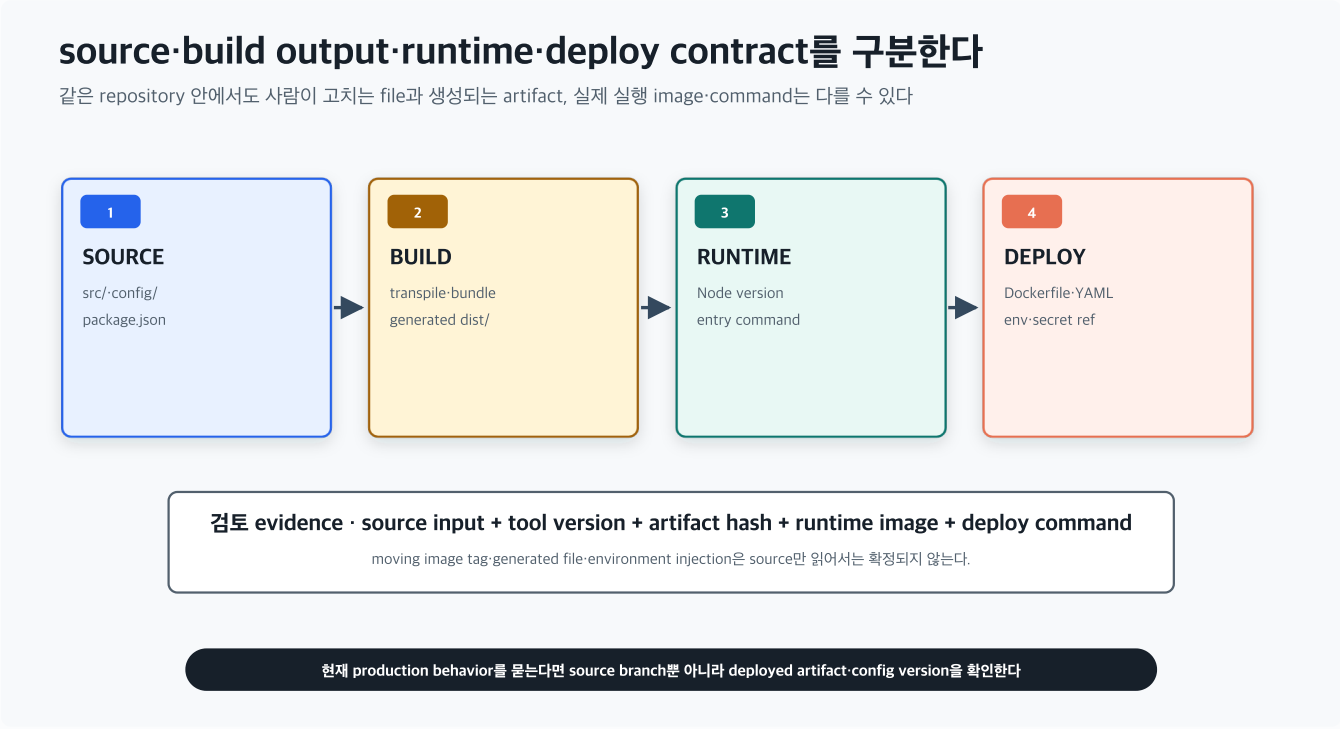
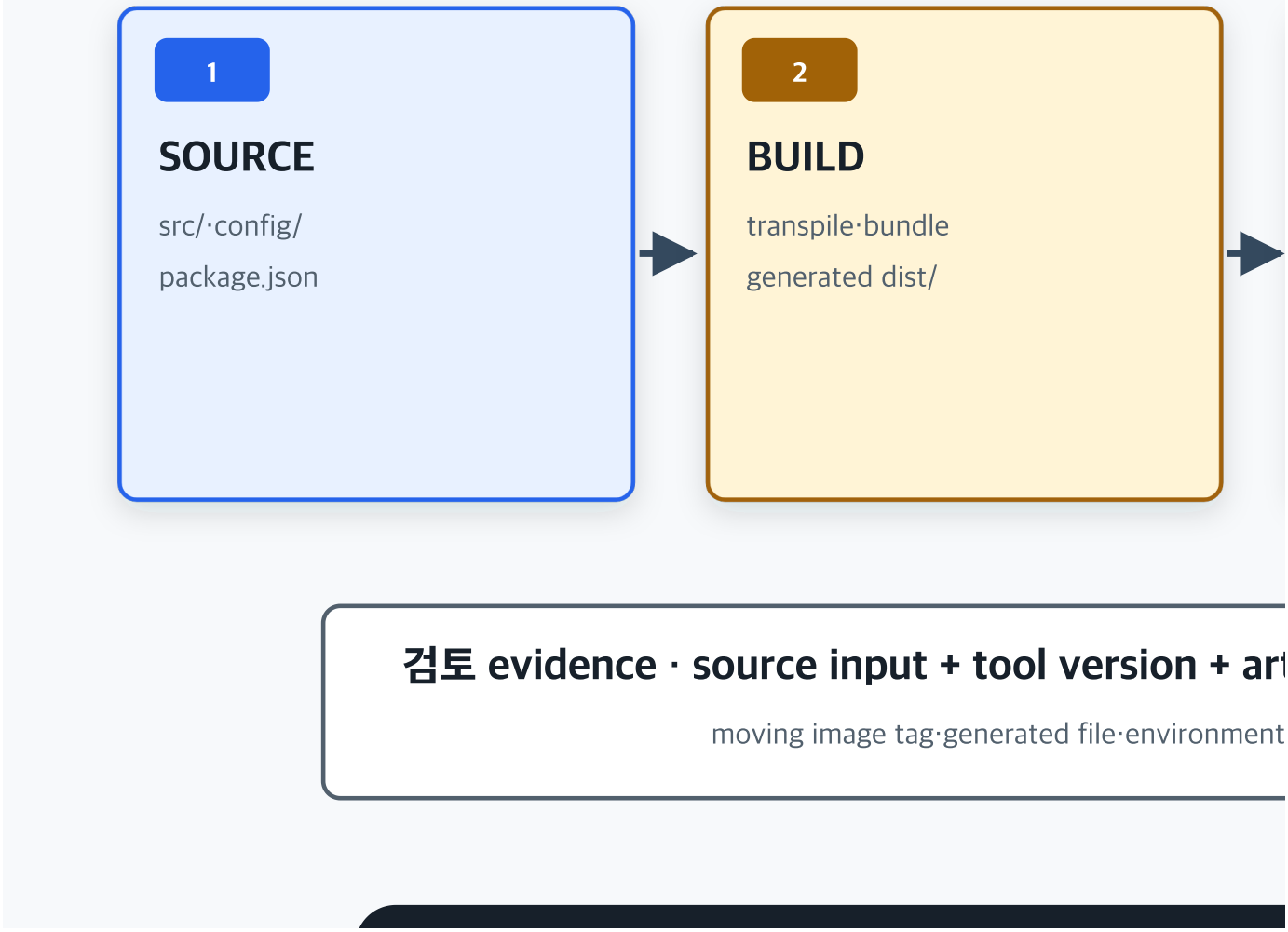
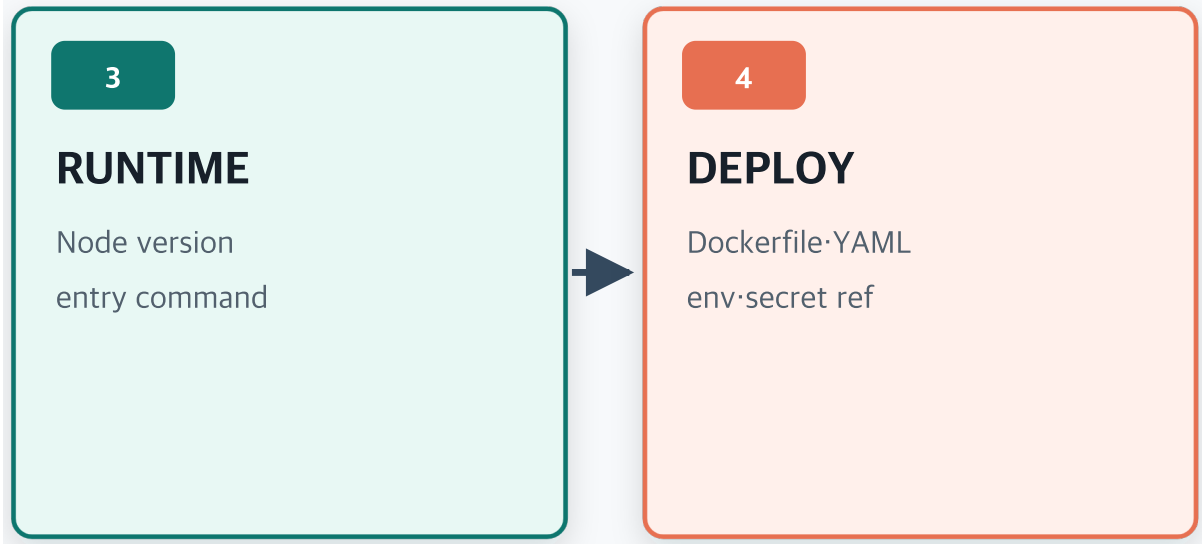


그림 12. source code를 읽었다고 production artifact·runtime·effective deployment까지 확인한 것은 아닙니다.


1
SOURCE
src/·config/
package.json
2
BUILD
transpile·bundle
generated dist/
검토 evidence · source input + tool version + art
moving image tag·generated file·environment



tifact hash + runtime image + deploy command

injection은 source만 읽어서는 확정되지 않는다.

14.1. 네 층

층	핵심 evidence	대표 gap
source	commit OID·manifest·code·config	local change·generated source
build	exact command·tool version·artifact hash	transform·minify·compile flag
runtime	executable·version·image digest·working dir	local과 production version 차이
deploy	platform manifest·env·secret·replica·route	override·rollout·stale config

14.2. moving image tag

```
FROM node:24-alpine
```

이 tag는 특정 digest가 아니므로 다른 시각에 다른 image를 가리킬 수 있습니다. 학습 예시에서는 읽기 쉽도록 사용했지만 production evidence에는 resolved digest·SBOM·provenance·scan·build record가 필요합니다.

14.3. current production 질문

"현재 production의 passing score는 75인가?"라는 질문에 source `config/service.json` 만 보고 답하지 않습니다.

```
deployed artifact OID·digest =
deploy revision =
effective environment =
config service·secret version =
runtime inspect evidence =
observed response =
```

source는 후보 contract이고 production observation은 별도 evidence입니다.

15. 실행 위험을 한 단계씩 올립니다

그림 13. 관찰·syntax check·test·server·install·build는 side effect 범위가 다릅니다. exact script를 읽고 단계적으로 실행합니다.

OBSERVE

먼저 읽기

root·version

manifest·tree

imports·config

deploy command

CHECK

제한 실행

node --check

syntax only

test in isolation

expected·actual

node --check는 syntax만 검사하며 import 존재

실행 전 scripts·hook·install lifecycle·ne

RUN

side effect 승인

npm run·test

server port

install·build

file·network·secret

·runtime behavior·test pass를 증명하지 않는다

twork·credential boundary를 확인한다.

15.1. observe

```
node --version
git rev-parse --show-toplevel
git rev-parse HEAD
sed -n '1,180p' package.json
find src config test deploy -maxdepth 3 -type f
rg '^import |^export ' src test
```

command 자체도 environment와 alias에 영향을 받을 수 있습니다. 출력에 secret·개인정보 path가 있는지 확인합니다.

15.2. syntax check

Node.js 공식 CLI 문서에서 `node --check`는 script를 실행하지 않고 syntax를 검사합니다.

```
node --check src/cli.js
node --check src/server.js
```

`--check`가 증명하지 않는 것:

- import target 존재·resolution
- JSON·environment load
- type·business behavior
- test pass
- server startup·port
- network·database 연결
- deploy configuration

15.3. test

```
node --test
```

test는 code를 실행합니다. 실습에서는 built-in test runner로 10개 test가 통과합니다.

```
tests 10
pass 10
fail 0
```

증명 범위:

- code default·file·environment·CLI selection
- string boolean·integer normalization
- invalid configuration failure
- progress threshold boundary
- repository data와 policy의 service orchestration

증명하지 않은 범위:

- Docker image build
- deployment YAML 적용
- production secret injection
- production file·network
- complete HTTP route matrix

15.4. server

```
node src/server.js --port=4313
curl http://127.0.0.1:4313/health
```

server는 종료 전까지 process와 port를 유지합니다. isolated local port·test data·cleanup을 준비합니다.

15.5. install·build

실습에는 외부 dependency가 없어 install이 필요 없습니다. 일반 project에서는 다음을 확인합니다.

- lockfile과 package manager version
- registry·proxy·credential
- lifecycle·prepare·postinstall script
- native build·platform dependency
- network egress
- generated file·workspace mutation

- license·provenance·vulnerability policy

16. 재현 가능한 코드 탐색 evidence를 만듭니다

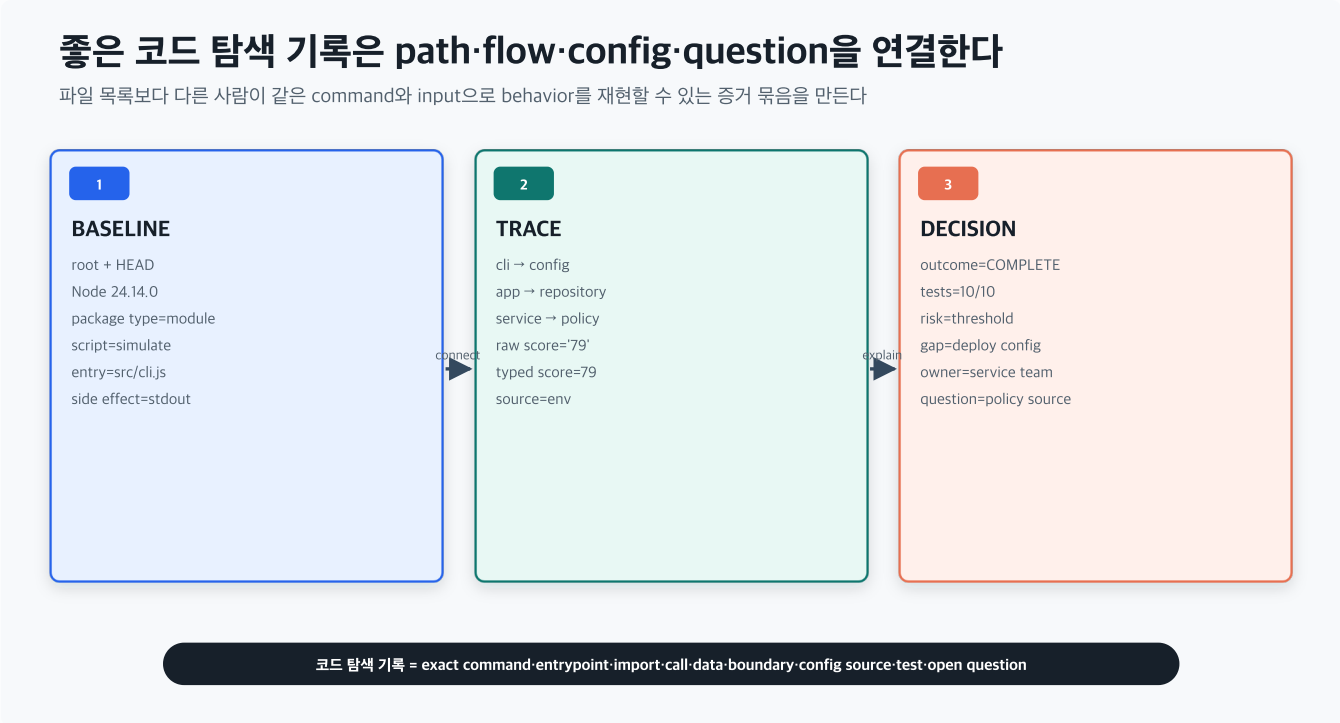
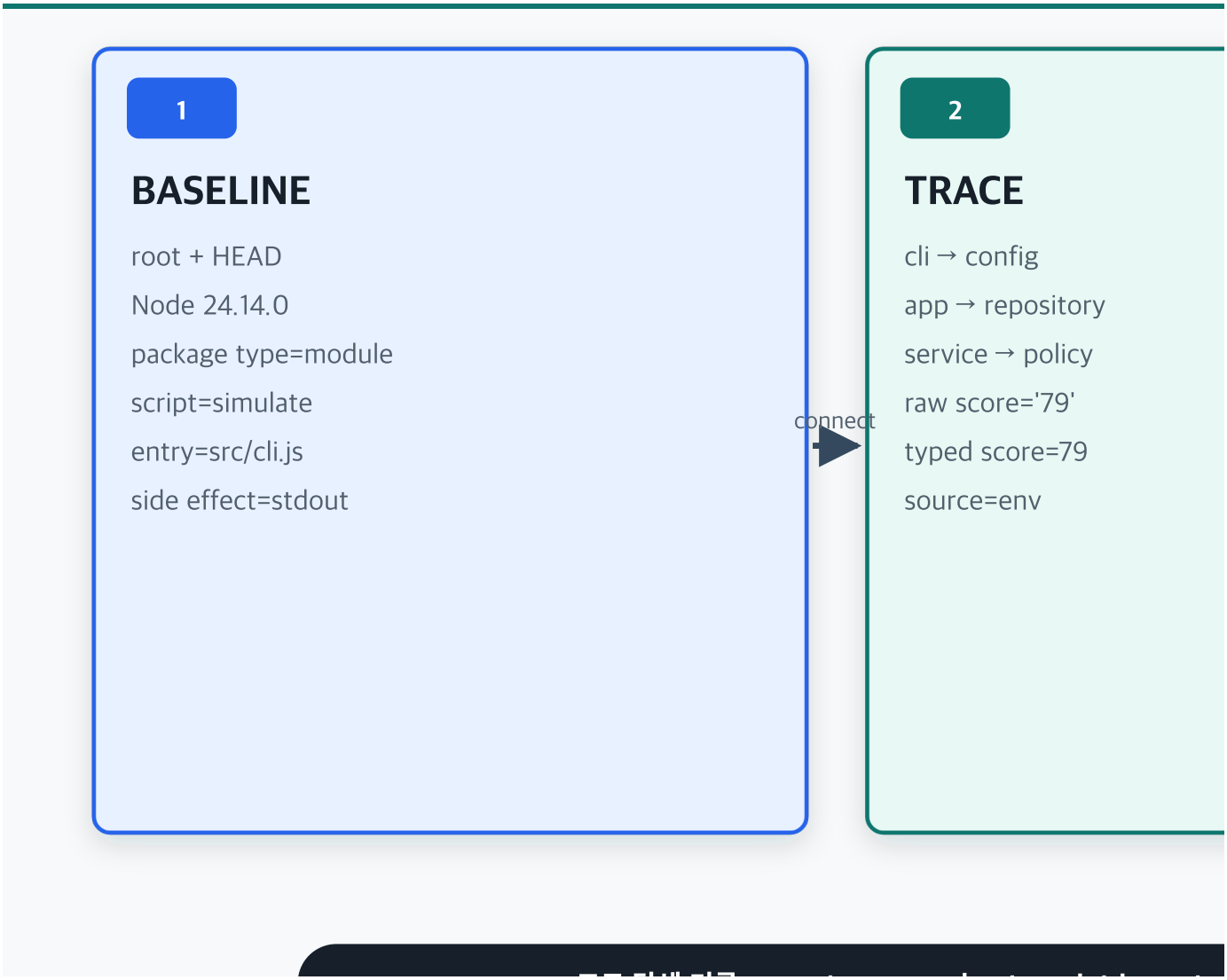
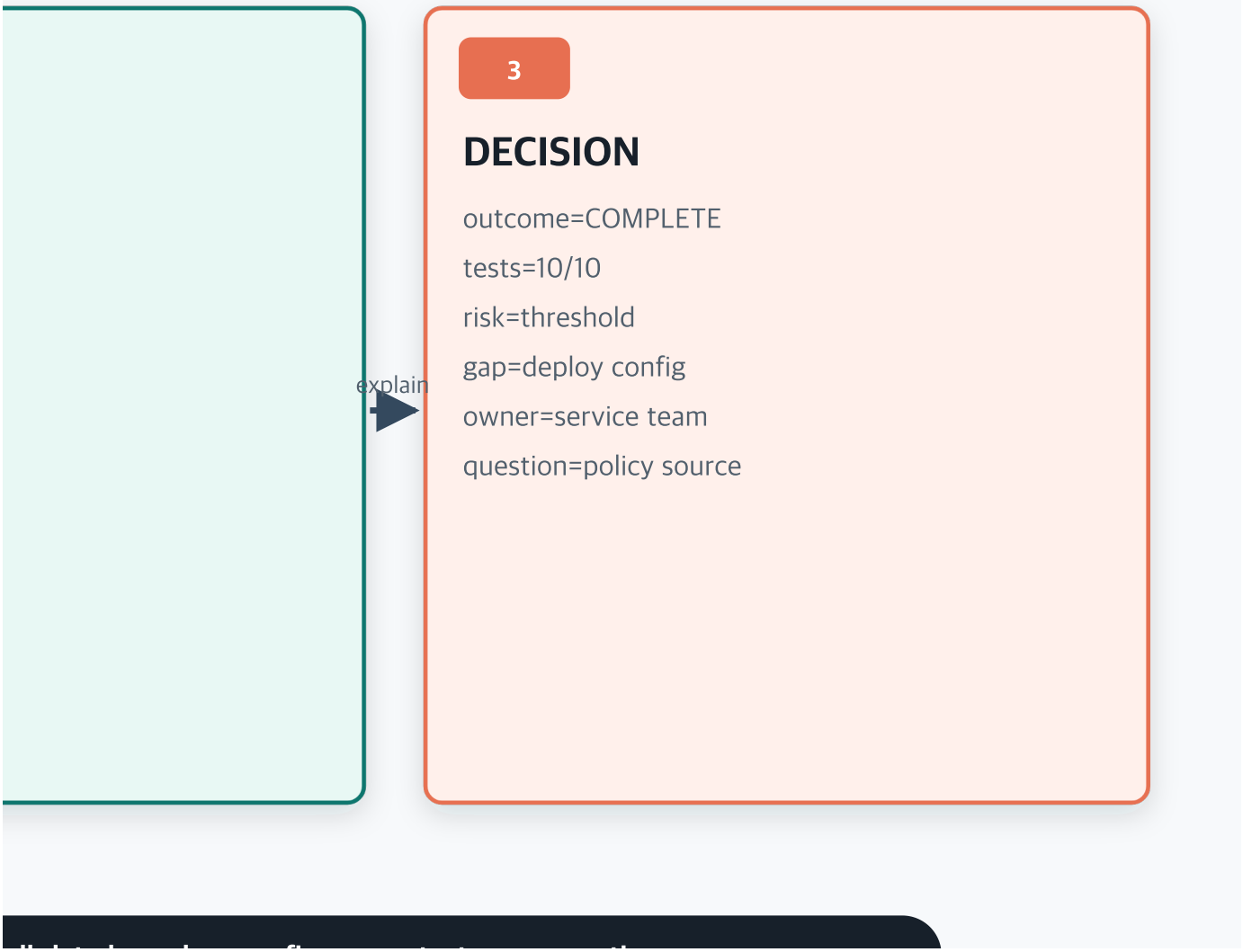


그림 14. baseline·trace·decision 세 묶음에 exact command, path, value source, test와 open question을 연결합니다.


1
BASELINE
root + HEAD
Node 24.14.0
package type=module
script=simulate
entry=src/cli.js
side effect=stdout
connect
2
TRACE
cli → config
app → repository
service → policy
raw score='79'
typed score=79
source=env


explain
3
DECISION
outcome=COMPLETE
tests=10/10
risk=threshold
gap=deploy config
owner=service team
question=policy source

16.1. baseline

```
root = /practice/L06-02
HEAD = [full OID]
Node = v24.14.0
package = yeoncore-code-reading-lab@0.2.0
module system = ESM
task = CLI simulate
```

16.2. trace

```
command = node src/cli.js simulate --learner=planning --score=74
entry = src/cli.js
call = main → loadConfig → createApplication → service → repository → policy
data = argv strings → typed config + catalog → decision object → JSON text
boundary = config read + catalog read + stdout
```

16.3. decision

```
effective passingScore = 75
source = file
input score = 74
outcome = IN_PROGRESS
tests = 10 pass / 0 fail
gap = deploy config·image·secret not proven
question = approved policy source?
```

16.4. evidence의 최소 조건

항목	반드시 기록할 것
source	root·HEAD·status·capture time
runtime	executable·version·OS
command	exact option·argument·working directory
entry	file·symbol·trigger
dependency	import target·public export·call edge

항목	반드시 기록할 것
data	input·output shape·type·source
boundary	file·env·network·stdout·secret
config	raw·selected source·normalized value·validation
result	output·exit code·side effect·cleanup
gap	not run·assumption·open question·owner

17. 코드 경로 탐색기 실습

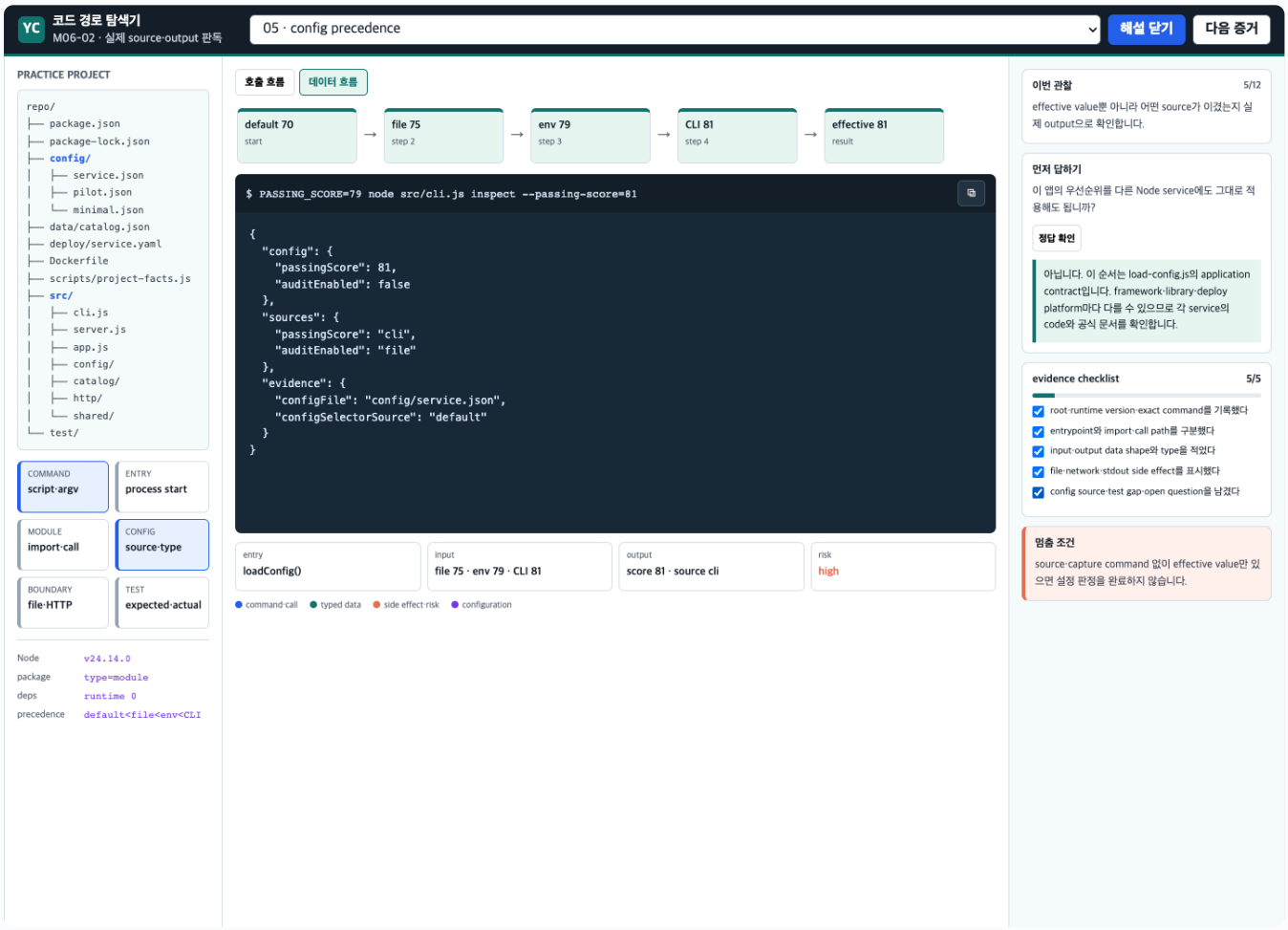


그림 15. 왼쪽에서 command·entry·module·config·boundary·test zone을 가리키고, 가운데 실제 source·output과 오른쪽 검토 질문을 연결합니다.

17.1. 12개 evidence

#	evidence	먼저 말할 것
1	project facts	runtime·module·scripts·deps
2	script→entrypoint	executable·file·argv
3	ESM entry module	imports·orchestration·output
4	structured argument parsing	raw type·option spec
5	config precedence	values·selected source
6	normalization failure	input·error·exit
7	composition root	dependency construction
8	file boundary	read·parse·shape gap
9	pure policy	threshold·operator·outcome
10	end-to-end	command→data→result
11	HTTP entry	port·route·response·cleanup
12	test·deploy gap	proven·not proven

17.2. 탐색기는 source를 대신하지 않습니다

화면의 code와 output은 제공된 생성기를 Node.js 24.14.0에서 실행한 기록입니다. 내 환경의 version·path·duration은 달라질 수 있습니다. 실제 결과는 단계별 실습 안내서를 따라 내 evidence에 기록합니다.

18. 60분 실전 과제

Mission A · 10분 · baseline·tree

```
node --version
git rev-parse --show-toplevel 2>/dev/null || true
node scripts/project-facts.js
find . -maxdepth 3 -type f | sort
```

runtime, package, scripts, imports, root file을 기록합니다. 실습 folder가 Git repository가 아닐 수 있으므로 Git command 실패도 사실대로 적습니다.

Mission B · 10분 · entrypoint matrix

`package.json`, `Dockerfile`, `deploy/service.yaml`을 읽고 CLI·server·test·deploy 네 row를 작성합니다.

```
task | command | entry | argv·event | first side effect | evidence
```

Mission C · 15분 · import·call·data trace

```
rg -n '^import |^export ' src test
sed -n '1,160p' src/cli.js
sed -n '1,200p' src/app.js
sed -n '1,200p' src/catalog/catalog-service.js
sed -n '1,160p' src/catalog/progress-policy.js
```

import graph 1개, call flow 1개, data flow 1개를 그립니다.

Mission D · 15분 · config precedence

```
node src/cli.js inspect --config=config/minimal.json
node src/cli.js inspect
PASSING_SCORE=79 AUDIT_ENABLED=false node src/cli.js inspect
PASSING_SCORE=79 node src/cli.js inspect --passing-score=81
```

70·75·79·81과 `default·file·env·cli` source가 맞는지 표로 비교합니다.

Mission E · 10분 · safe execution·gap

```
node --check src/cli.js
node --check src/server.js
node --test
node src/cli.js simulate --learner=planning --score=74
```

syntax, 10개 test, IN_PROGRESS result를 기록하고 Docker·deploy·production에서 아직 증명하지 않은 것을 세 개 적습니다.

19. 자주 실패하는 읽기 12가지

실패	왜 틀리는가	바꿀 evidence
folder 이름으로 역할 확정	convention은 증거가 아님	command·import·call
`index.js` 를 시작점으로 추측	task별 entry가 다름	script·CMD·deploy
script label만 읽음	value가 실제 command	executable·file·argv
import graph를 call graph로 봄	import돼도 branch에서 미호출	call path·runtime trace
함수 이름만 기록	data·type·failure 누락	input·output contract
`false` 를 boolean으로 간주	environment는 string	explicit parse evidence
effective 값만 기록	source·precedence 재현 불가	raw·selected source
config file 하나를 production 값으로 봄	env·deploy override 가능	deployed effective config
JSON parse 성공을 validity로 봄	schema·business rule 별도	type·range·invariant
`node --check` 를 test로 봄	syntax만 검사	import·test·runtime
test pass를 deploy pass로 봄	image·secret·platform 미검증	not-proven register
`npm run` 을 안전한 관찰로 봄	arbitrary script 실행	script·side effect·isolation

20. 셀프 테스트 · 먼저 답을 적으세요

문제 1 · baseline

코드 탐색을 시작할 때 Node version 외에 반드시 기록할 repository 기준 세 가지를 쓰세요.

답: _____

문제 2 · command

`"simulate": "node src/cli.js simulate --learner=planning"` 을
executable·entrypoint·positional·option으로 나누세요.

답: _____

문제 3 · entrypoint

한 repository에서 CLI·server·test entrypoint가 다른 이유를 한 문장으로 쓰세요.

답: _____

문제 4 · module

`#config`가 어느 file인지 확인하려면 무엇을 읽어야 합니까?

답: _____

문제 5 · graph

import graph와 call graph의 차이를 쓰세요.

답: _____

문제 6 · flow

call flow와 data flow를 따로 그리는 이유는 무엇입니까?

답: _____

문제 7 · boundary

실습에서 pure core 밖 I/O boundary 네 가지를 쓰세요.

답: _____

문제 8 · precedence

file 75, environment 79, CLI 81일 때 실습의 effective passing score와 source는 무엇입니까?

답: _____

문제 9 · environment

`Boolean('false')`의 결과와 올바른 처리 원칙을 쓰세요.

답: _____

문제 10 · validation

JSON parse 성공이 config validity를 증명하지 않는 이유를 쓰세요.

답: _____

문제 11 · check·test

`node --check`와 `node --test`가 각각 증명하는 범위의 차이를 쓰세요.

답: _____

문제 12 · production

source config가 75라고 production도 75라고 단정할 수 없는 이유와 추가 evidence 두 가지를 쓰세요.

답: _____

21. 셀프 테스트 정답·해설

답 1 · baseline

repository top-level, full HEAD OID, working tree status입니다. capture time·OS도 함께 기록하면 재현성이 높아집니다.

답 2 · command

executable은 `node`, entrypoint는 `src/cli.js`, positional command는 `simulate`, option은 `--learner=planning`입니다.

답 3 · entrypoint

task의 trigger와 side effect가 다르기 때문입니다. CLI는 argv·stdout, server는 port·request, test는 runner·assertion으로 시작합니다.

답 4 · module

nearest `package.json`의 `imports` field에서 `#config` mapping을 읽습니다. 실습 target은 `./src/config/load-config.js`입니다.

답 5 · graph

import graph는 module이 어떤 public symbol을 사용할 수 있는지 보여 주고, call graph는 특정 input·branch에서 function이 실제 또는 가능한 순서로 무엇을 호출하는지 보여 줍니다.

답 6 · flow

control 이동과 value·type 이동은 다르기 때문입니다. function을 호출해도 같은 data가 전달되지 않을 수 있고, 한 value가 여러 단계에서 변환될 수 있습니다.

답 7 · boundary

environment read, config·catalog file read, stdout·stderr output, HTTP listen·response가 대표적입니다.

답 8 · precedence

81, source `cli` 입니다. 이 답은 실습 application의 `choose()` contract에만 적용됩니다.

답 9 · environment

`true` 입니다. 비어 있지 않은 string이므로 truthy입니다. accepted literal을 명시적으로 비교해 true·false로 변환하고 그 밖의 값은 실패시킵니다.

답 10 · validation

parse는 JSON grammar만 확인합니다. required key, type, range, allowed value, path boundary, cross-field·business invariant는 별도 schema·code가 검증해야 합니다.

답 11 · check·test

`node --check` 는 target script syntax를 실행 없이 검사합니다. `node --test` 는 test file과 application module을 실행해 assertion의 expected·actual을 판정하지만 test가 없는 deploy 영역까지 증명하지는 않습니다.

답 12 · production

environment·deploy platform·config service가 source file을 override할 수 있고 다른 artifact가 배포됐을 수 있습니다. deployed artifact OID·image digest, deploy revision, effective environment, runtime inspect·response 중 두 가지 이상을 확인합니다.

22. 완료 체크리스트

baseline·command

- [ ] repository root·HEAD·status·capture time을 기록했다.
- [ ] runtime·package manager version을 기록했다.

- [ ] task별 exact command·working directory를 기록했다.
- [ ] script label과 command value를 구분했다.
- [ ] executable·entry file·positional·option을 나눴다.

structure·flow

- [ ] directory 역할을 command·import·call로 검증했다.
- [ ] CLI·server·test·deploy entrypoint matrix를 만들었다.
- [ ] module system과 specifier 종류를 기록했다.
- [ ] import graph와 call graph를 구분했다.
- [ ] call flow와 data flow를 따로 그렸다.
- [ ] main path와 failure path를 적었다.
- [ ] file·environment·network·output boundary를 표시했다.

configuration

- [ ] config key별 raw source name을 기록했다.
- [ ] default·file·env·CLI precedence를 code로 확인했다.
- [ ] effective value와 selected source를 함께 기록했다.
- [ ] string·number·boolean·missing·empty를 구분했다.
- [ ] range·allowed value·path·cross-field validation을 확인했다.
- [ ] secret 원문을 evidence에 남기지 않았다.
- [ ] invalid config의 error·exit behavior를 확인했다.

execution·evidence

- [ ] syntax check와 behavior test를 구분했다.
- [ ] 실행 command의 file·network·credential side effect를 검토했다.
- [ ] test expected·actual·count·environment를 기록했다.
- [ ] server·process·port cleanup을 기록했다.
- [ ] source·build·runtime·deploy gap을 나눴다.
- [ ] not run·assumption·open question·owner를 적었다.
- [ ] 다른 사람이 exact command와 source version으로 재현할 수 있다.

23. 공식 참고 자료

Node.js 24 LTS current documentation

- Modules: Packages
- ECMAScript modules
- CommonJS modules
- Environment variables and DotEnv
- process.env and process.argv
- Command-line API
- util.parseArgs
- File system
- HTTP
- Test runner

npm·data format

- npm package.json
- npm scripts
- npm package-lock.json
- npm .npmrc
- RFC 8259 · JSON

버전 기준일: 2026-07-15. 공식 Node.js 24 LTS 문서는 24.18.x 계열, npm current 문서는 11 계열을 확인했습니다. 실습 생성기·14개 source syntax·10개 test·네 config precedence·CLI 두 outcome·HTTP health·simulation은 bundled Node.js 24.14.0에서 직접 검증했습니다. Node.js·npm·OS·shell·framework·deployment platform이 다르면 command·resolution·precedence·output이 달라질 수 있으므로 exact version과 official documentation을 함께 기록합니다.