
Finding the change in a large codebase

Key takeaways

1. Begin at one wrong artifact you can reproduce on demand, not at the repository root. A reliable symptom tells you which code could possibly explain it.
2. Follow exact filenames, build targets, symbols, and values until a handful of files form one connected slice from the symptom to the source of truth.
3. A cause earns trust only when it predicts observations that could have disproved it. State those predictions, then run the cheap test that would have failed.

Common misunderstandings of large-repository navigation

Which claim would you defend right now? Choose before the evidence.

“Understanding a large repository starts by reading its top-level structure.”

“Searching for architectural concepts is faster than following exact filenames.”

“A plausible explanation becomes a diagnosis once it fits the observed failure.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

A changed source that never rebuilds

Historical worked example: a deck's shared source changed upstream; Make served a stale PDF

Recorded before the fix: same artifact, two different answers

```
$ cd slides
$ touch ../docs/2026-07-05-course-structure-review.md
$ make -n 097_react_architectures-slides.pdf
make: '097_react_architectures-slides.pdf' is up to date.

$ touch gen_097_react_architectures.py
$ make -n 097_react_architectures-slides.pdf
./gen_097_react_architectures.py
```

Do not begin by reading the repository. Begin by making one wrong behavior reproducible, then follow only the code that can explain it.

Reproduce before you navigate

Minimal experiment: input, operation, expected observation

```
cd slides
target=097_react_architectures-slides.pdf
input=../docs/2026-07-05-course-structure-review.md

make "$target"           # establish a clean baseline
touch "$input"           # change one claimed dependency
make --dry-run "$target" # should print the rebuild command

# Failure: Make prints "up to date".
```

Invariant under test: if a source that contributes to a PDF changes, the PDF target must become out of date.

Instapoll

Instapoll · Review · Oct 20 B

You have the failure reproduced: make reports the PDF up to date after its source changed.
You have never opened this repository before. What do you search for first?

- A. The words 'dependency' and 'rebuild' across the build scripts
- B. The exact PDF filename that make just printed, as a literal string
- C. The top-level Makefile, read from the top down until it makes sense
- D. Nothing yet — ask the agent which module decides staleness here

Start at the artifact boundary

The public schedule names the artifact. That literal is a better entry point than guessing which Python module ‘sounds related.’

The observed consumer

```
# web/lectures.txt
- title: "ReAct pattern & agent
architectures"
  slides: lectures/
    097_react_architectures-slides.pdf
```

Search the exact artifact

```
$ rg -n
'097_react_architectures-slides.pdf'

web/lectures.txt:405:    slides: ...
slides/gen_097_react_architectures.py:...
```

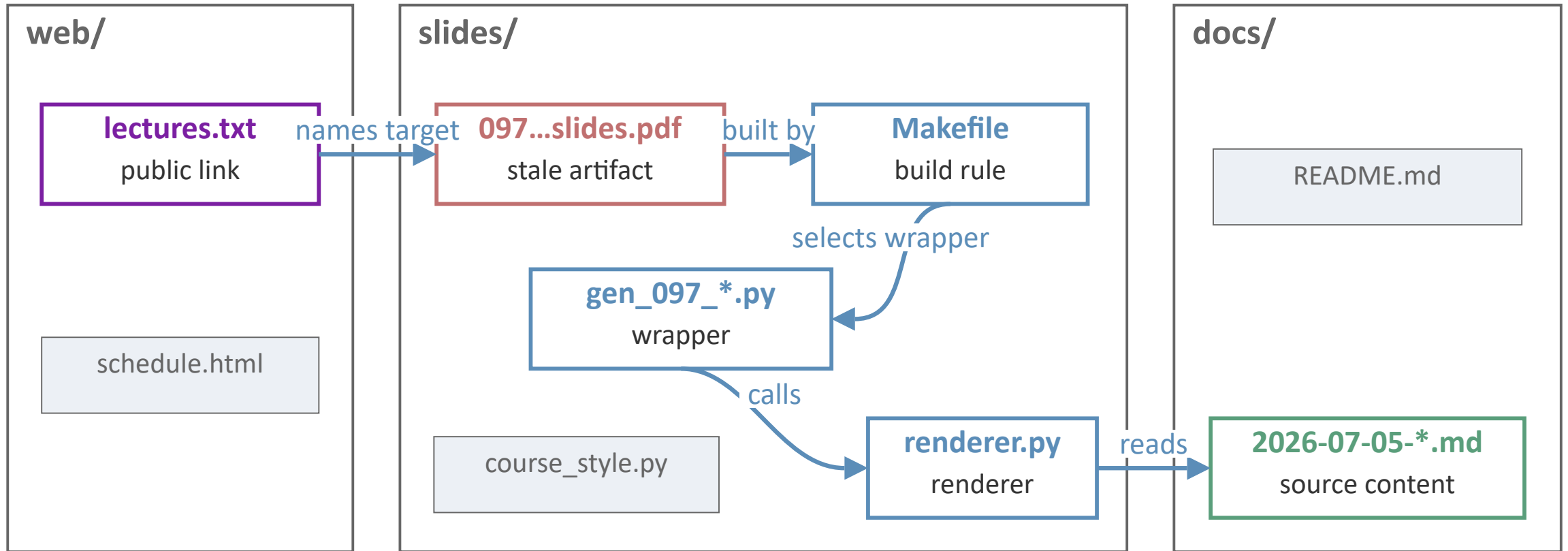
Trace one complete dependency slice

Five files are enough to explain the stale PDF. Everything else in the repository is temporarily irrelevant.

step	file	evidence that points to the next step
1 public link	web/lectures.txt	names the PDF target
2 build rule	slides/Makefile	maps target stem to gen_*.py
3 wrapper	gen_097_*.py	calls render_review_deck(...)
4 renderer	review_deck_renderer.py	reads REVIEW_PATH
5 content	docs/2026-07-05-*.md	owns the provisional outline

A code map is not a directory tour. It is the smallest connected slice from symptom to source of truth.

Follow one dependency slice across directories



Blue edges: followed evidence • Gray files: ignored for this diagnosis

Search literals before concepts

Names that cross boundaries survive refactors better than your guess at the architecture. Search the filename, target stem, and command first.

Breadth with rg; depth with a bounded read

```
$ rg -n '097_react_architectures' web slides docs
$ rg -n '%-slides\.pdf|GEN_SCRIPTS|PDF_FILES' slides/Makefile
$ rg -n 'render_review_deck|REVIEW_PATH' slides

# Narrow only after the first hit:
$ sed -n '400,410p' web/lectures.txt
$ sed -n '1,40p' slides/gen_097_react_architectures.py
$ sed -n '185,260p' slides/review_deck_renderer.py
```

Follow symbols to ownership

Wrapper: configuration, not content

```
render_review_deck(  
    review_id="097",  
    output_name=(  
        "097_react_architectures-slides.pdf"  
    ),  
    accent=ACCENT,  
    title="ReAct pattern & agent architectures",  
)
```

Renderer: content owner is elsewhere

```
COURSE_ROOT =  
Path(__file__).resolve().parents[1]  
REVIEW_PATH = COURSE_ROOT / "docs" / (  
    "2026-07-05-course-structure-review.md"  
)  
  
spec = _load_deck(review_id, ...)  
...  
save_pdf(deck, out_path)
```

The call says what varies; the callee reveals who owns the shared behavior and data.

Trace data, not just calls

One value crosses four boundaries

```
review_id = "097"                # wrapper
  ↓
parsed = _parse_review()          # reads REVIEW_PATH
spec = parsed[review_id]         # selects one outline
  ↓
_title_slide(...); _section_slide(...) # normalizes content
  ↓
save_pdf(deck, out_path)         # emits target
```

The outline contributes to spec, so it is a build dependency even though the wrapper never names its path.

State a cause that can lose

A diagnosis is useful only if another observation could disprove it.

Rule before the repair

```
# historical Make rule
%-slides.pdf: gen_%.py
    ./${<

# only the wrapper timestamp matters
```

Hypothesis + predictions

```
cause = "missing prerequisite edges"

predict(touch(wrapper)) == REBUILD
predict(touch(renderer)) == NO_REBUILD
predict(touch(outline)) == NO_REBUILD

# all three predictions matched
```

Cause: Make cannot see the wrapper's imported renderer or the renderer's outline read, so it serves a stale target exactly as instructed.

Separate observations from inferences

evidence	observation	inference
old Make rule	only gen_%.py was a prerequisite	imports were invisible
wrapper	calls shared renderer	renderer changes affect output
renderer	reads a fixed outline path	outline affects output
touch tests	wrapper rebuilds; shared files do not	cause predicts behavior
rg count	12 wrappers shared the renderer then	blast radius was broad

Keep this ledger in the prompt or a scratch file. It prevents a plausible story from quietly becoming a fact.

Ask the agent a bounded, testable question

Weak: invites a repository tour

Explain how the slide system works and fix anything wrong with it.

Strong: names evidence and deliverable

```
`make -n TARGET` stays up to date after  
OUTLINE changes. Trace only the dependency  
path that can explain that result. Return:  
1. file:line evidence for every edge,  
2. a falsifiable cause,  
3. tests that distinguish two candidate  
fixes.  
Do not edit files.
```

The prompt controls context by controlling the question. A smaller causal slice usually produces a better explanation.

You vs. the Agent: repository navigation

Your Responsibility

Choose the symptom and expected behavior

Reject a story that lacks disconfirming evidence

Decide when the causal slice is complete

Protect scope: investigation first, mutation later

Agent's Responsibility

Search literals, files, symbols, and callers quickly

Extract a compact dependency and data-flow map

Run read-only experiments and record exact outputs

Propose competing causes and distinguishing tests

From stale artifact to causal slice

1. Start from one wrong artifact you can reproduce on demand, not from the repository root. Name the input, the operation, and the expected observation first: that experiment is what tells you which code could explain the failure.
2. Follow exact names, calls, and values instead of architectural concepts, because a filename or build target that crosses a boundary survives refactors. Stop when a few files form a connected slice from symptom to source of truth.
3. A diagnosis earns trust by predicting observations that could have disproved it. Keep a ledger that separates what you observed from what you inferred, so a plausible story cannot quietly become a fact.