Finding the change in a large codebase

Key takeaways

1. Start from one reproducible wrong artifact, not the repository root.
2. Follow exact names and values until one causal slice explains it.
3. A cause earns trust only by predicting what could have disproved it.

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: deck 097 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Invariant under test: if a source that contributes to a PDF changes, the PDF target must become out of date.

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:...
```

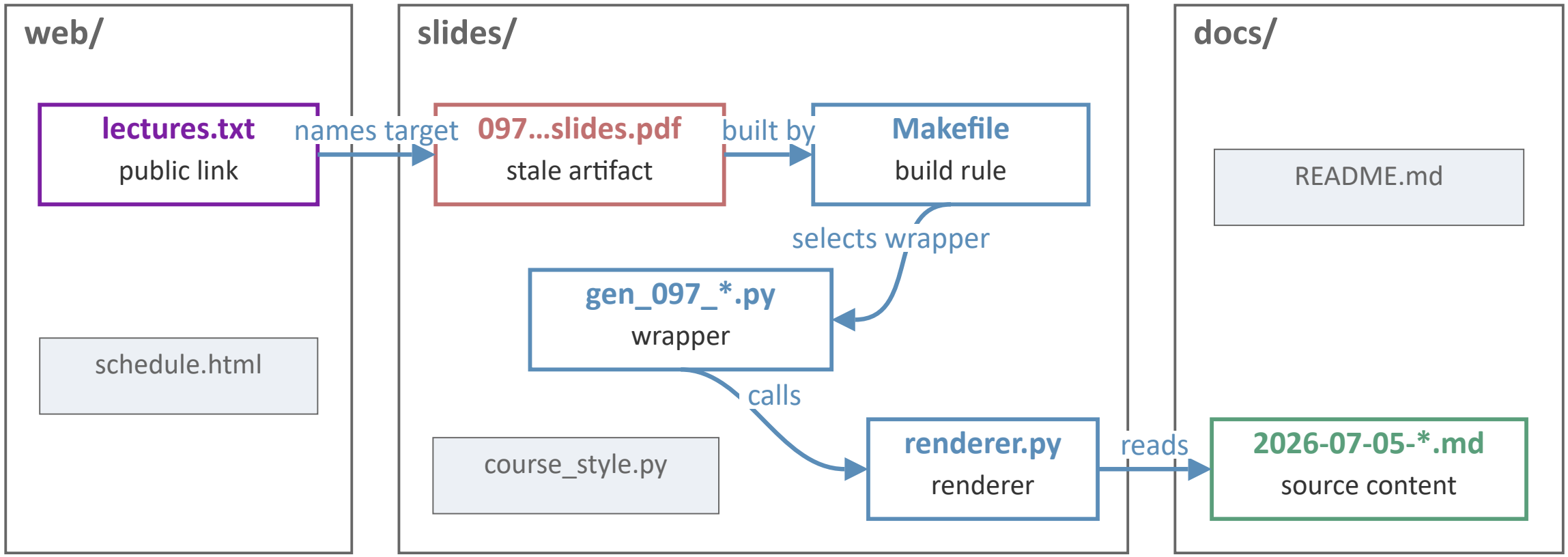
Trace one complete dependency slice

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

source: web/lectures.txt; slides/Makefile; wrapper; renderer; review outline

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.

source: this repository: web/lectures.txt, slides/Makefile, wrapper, renderer, and outline

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 reproducible wrong artifact, not from the repository root.
2. Follow exact names, calls, and values until you have one connected causal slice.
3. A diagnosis earns trust by predicting observations that could have disproved it.

Find the missing dependency

10 min

- A build has `report.pdf: gen_report.py`; `gen_report.py` calls `render_report.py`, which reads `outline.md`. Touching `outline.md` leaves `make -n report.pdf` saying “up to date.”
- Alone, draw the artifact → rule → entry point → input slice and write two competing causes for the stale result.
- In pairs, choose the cheapest command that distinguishes the causes; write its predicted output before running it.
- Swap slices with another pair: they must name a missing edge or accept the diagnosis. Deliverable: cited slice, test, prediction, and verdict.