mdbook-listings

Managed code listings for mdbook

Paul Adamson

Table of Contents

Introduction

`mdbook-listings` is a small `mdbook` preprocessor and companion command line interface (CLI) for authors writing technical prose about code. The tool and the book you are reading now were co-developed. This book is `mdbook-listings`'s documentation, the record of the methodology used to build it, and the tool's primary dog food. Several chapters in this book describe building an `mdbook-listings` primitive, and the tool is used to manage the listings in the book, so a regression in the tool breaks the book build.

A PDF version of this book is available at mdbook-listings.pdf.

How books drift from code

A book quotes code from the project it documents. Months later the code has been refactored — a function is renamed, a config file gains keys, an example stops compiling. The book quotes don't change with the code. A chapter cites a function that no longer exists; a `{{#include}}` directive picks up different bytes than the surrounding prose describes; a sample fails to compile against the current project.

`mdbook-listings` provides four primitives to keep the book coherent with the code:

1. **Stability** — embed a code listing in the book with a guarantee

that the rendered output does not change when the source file changes. (*Freeze a Listing*, ch. 2.)
1. **Evolution** — show how code evolves across a chapter's slices

without repeating the full file contents each time. (*Show Diffs Between Slices*, ch. 3.)
1. **Annotation** — attach prose to specific lines of an embedded

listing, and reference those attachment points stably from surrounding prose. (*Render Inline Callouts*, ch. 4.)
1. **Synchronization** — warn the author when a listing that is

supposed to track the current source has fallen out of sync. (*Verify Sync with Source*, ch. 5.) Ch. 1 (*Install the Preprocessor*) ships the one-shot onboarding command an author runs before they can use any of the four primitives — it's the entry point of the user's journey through the tool. Everything else that isn't a primitive — small ergonomics, recipes, troubleshooting — lives in ch. 6.

What's planned beyond what's shipped (v0.2.0 themes onward) lives in `ROADMAP.md` at the repo root, not in this book. The book documents shipped stories; the roadmap documents intended ones.

Scope boundaries — things the tool deliberately does not do

Worth stating up front so readers aren't surprised:

- **Replacement of a shipped tag is not a primitive.** Once a frozen

tag has been referenced by a shipped chapter, its bytes are immutable. `--force` exists as an escape hatch for pre-ship corrections (typos, missed formatting, accidental debug prints), but using it on a tag readers have already seen silently changes what they see. If you need different bytes, make a new tag. If you need to retire an old tag nobody references anymore, hand-edit `listings.toml` and delete the file.
- **The tool does not parse your source code.** Freezing is a

byte-level copy; diffing is a byte-level diff; CALLOUT markers are matched by substring within comment lines, not by a language-aware parser. One consequence: a CALLOUT marker inside

a multi-line string literal in Rust *looks* like a callout and will be rendered as one. Write your callouts in comments.

- **The tool does not run your code.** No compile check, no test

execution, no "does the listing still typecheck." Deep verify (eventually — not in the initial release) might add a compile check as an opt-in. For now, if your listing's code rots, verify catches the byte-level drift but not the semantic drift.

How the book is organized

Each content chapter is a **user story**: a single slice of value stated in the first person, from the point of view of a book author using the tool. A chapter takes the reader through:

1. **The story.** One sentence in user-story form — *as a book author I want X so that Y*.

2. **Acceptance criteria (AC).** Statements of the behavior the

implementation must exhibit for the story to be "done." Each AC is verified by one or more tests in the crate's `tests/` directory. Each user story has its own integration-test file (`tests/<story>.rs`), with shared helpers in `tests/common/mod.rs`, so a chapter's frozen test listing focuses on the story it documents rather than re-freezing a growing monolith. The tests are the executable form; the AC are the specification.

1. **Outside-in narrative.** A sequence of slice-sized sub-sections

walking through the implementation (see below). Each slice that modifies a source file — *including test files* — freezes the new state under a fresh `<file>-vN` tag and embeds the listing in the sub-section. Even tiny changes (e.g. removing a `#[ignore]` attribute) get a new version: predictability beats negotiating "is this change substantive enough?" every time, and the diff primitive (ch. 3) lets later chapters render compact diffs between consecutive tags rather than full files.

Until the diff primitive ships, each slice's narrative must explicitly describe in prose *what changed* in the new version relative to the previous one. The reader sees two full file listings across consecutive sub-sections; the prose tells them where to look. After diffs ship, the prose can cede that work to a diff block between consecutive `-vN` tags.

1. **Design decisions** *(optional)*. The rationale for choices the

tests and implementation can't show on their own — *why* this approach and not the alternatives. Include when the story made non-obvious choices that a future maintainer would need to reconstruct from scratch; omit when nothing about the story needed defending.

1. **Final state** *(optional)*. `{{#include}}`s of the frozen

listings of every file in the slice at their end-of-story state — the latest tag per file. Include when the narrative sub-sections didn't already show the latest tag for every file (which can happen in retrospective chapters or chapters where only some slices re-froze). Omit when every file in the slice has its latest tag embedded somewhere in the narrative above — adding Final state in that case just duplicates the bytes.

1. **What this slice does not solve** *(optional)*. The deliberate

edges of the slice — features the author *knew* they wanted but deferred — with forward references to the stories or chores that will pick them up. Include when there is a backlog worth surfacing; omit when the slice closes the loop on its own.

Outside-in TDD and slices

Stories are implemented outside-in:

1. **Slice 1 — the failing integration test.** Write an acceptance

test at the outermost layer (typically a CLI-level test that invokes the binary). It fails because the code it needs doesn't exist. Commit.
1. **Slices 2..N — inner unit tests, one per piece.** Drop down a

level. Identify the first piece of code the integration test needs. Write a unit test for it (red). Write the minimum code that greens it. Commit. Repeat: next piece, another red-then- green pair, another commit. Continue until the integration test goes green.
1. **Slice N + 1 — refactor (optional).** Tidy up while everything is

green. Commit. Each slice is its own commit. Commit messages name the slice (`Show Diffs slice 3/6: unit test + impl for diff_between`). The chapter's outside-in-narrative section walks through the slice sequence in reading order, quoting snippets from each.

Commits and stories are not always the same thing

Most commits ship a slice of a story. Some commits are chores (scaffolding, dependency bumps, supply-chain exemptions) or narrow bug fixes that don't belong to any story. Those commits don't get their own chapter — they live in `git log` where anyone who needs them can find them. What the chapter count tracks is *stories shipped*, not commits.

Early chapters don't fully illustrate the methodology

There's a chicken-and-egg problem at the start of this book. Several of the tool's features (diffs between slices, inline callouts) are exactly the features that would make an outside-in narrative compact — and those features don't exist yet when the stories that build them are shipped.

Specifically:

• **Ch. 1 (Install the Preprocessor)** was built fully outside-in

across 8 slices plus a refactor — install doesn't depend on either diffs or callouts, so the chapter has no chicken-and-egg constraint and serves as the cleanest example of the methodology in the book.

• **Ch. 2 (Freeze a Listing)** is reconstructed retrospectively.

The freeze work landed in a single commit before the book adopted this methodology; there is no slice-by-slice sequence to walk through.

• **Ch. 3 (Show Diffs Between Slices)** is built outside-in, but it

can't use the diff primitive in its own narrative because the diff primitive is what the story builds.

• **Ch. 4 (Render Inline Callouts)** can use diffs from ch. 3 but

can't annotate its own code with callouts for the same reason.

From ch. 5 onward every story has both diffs and callouts available, and the outside-in narrative settles into its compact shape. Early chapters will be noticeably longer — we quote whole intermediate file states where later chapters quote diffs.

Reading in order vs skipping around

Each story depends only on the ones before it — the slices stack. If you want to rebuild the tool from scratch by following the chapters, you can; each one leaves the crate in a compiling, shipping state. If you just want to learn how to use `mdbook-listings`, skim the **Story** and **Acceptance criteria** blocks and the **Final state** listings (when the chapter has one) and ignore the narratives. If you're here for the methodology, read the narratives and skim the reference content.

Install the Preprocessor

> **This chapter has shipped**
>
> The story shipped across eight slices plus a small refactor and a wrap-up chore. Every Acceptance criterion is exercised by at least one test in the suite. Read the chapter top-to-bottom for the methodology view; the **Outside-in narrative** sub-sections embed each frozen tag at the slice that introduced it, so the latest version of each file is in the slice that touched it last (Refactor for `src/install.rs`, slice 8 for `tests/install.rs`, slice 6 for `src/main.rs`, slice 3 for `Cargo.toml`, slice 2 for `src/lib.rs` and `assets/mdbook-listings.css`).

Story

> As a book author, I want one command that wires mdbook-listings into my book so that I don't have to hand-edit configuration or hunt down assets to start using the tool.

Acceptance criteria

1. After install runs successfully against a book, building that book

invokes mdbook-listings as a preprocessor without further author intervention.
1. After install runs successfully against a book, the HTML build

picks up the CSS asset that styles mdbook-listings's output.
1. Install is idempotent: a second run on an already-installed book

makes no further changes and confirms to the author that nothing changed.
1. Install preserves the rest of the book's existing configuration —

comments, formatting, and the order of any already-registered preprocessors and outputs are untouched; only entries relevant to mdbook-listings are added.
1. Install run in a directory without a valid book configuration is

rejected with a diagnostic identifying what was expected and not found.
1. If mdbook-admonish is also registered in the book, install places

mdbook-listings *before* it in the preprocessor chain so the callout → admonish-note pipeline produces correctly styled PDF output.

The slice — outside-in narrative outline

The story shipped as eight slices plus a refactor:

Slice	What it adds
1/8	Failing integration test asserting ACs 1 + 2 via post-install disk state: a minimal fixture book's `book.toml` gains a `[preprocessor.listings]` entry, references the bundled CSS asset in `[output.html].additional-css`, and the asset itself is written to the book root. Fails because install is a stub. (Asserting AC 1 by actually running `mdbook build` is deferred — it would couple the test to having mdbook on PATH at test time.)

2/8	Bundle the CSS asset into the binary at compile time (`include_bytes!`). Unit test: asset is non-empty + matches an expected sentinel. CSS contents stay a placeholder until ch. 4 (Callouts) settles the badge styling.
3/8	TOML round-trip primitive (read `book.toml`, mutate, write back preserving comments + ordering, via `toml_edit`). Unit-tested on synthetic input strings — no filesystem.
4/8	Add the `[preprocessor.listings]` registration. Unit test for AC 3 (idempotency) on top of slice 3.
5/8	Copy the CSS asset to `<book-root>/mdbook-listings.css` and add it to `[output.html].additional-css`. Unit test for the additional-css addition (AC 2 in the synthetic-config form).
6/8	Wire slices 2–5 into the `install` CLI handler. Slice 1's integration test now passes for ACs 1 + 2. AC 3 (idempotency) is pinned by slice 4's unit test.
7/8	Reject missing book config with a diagnostic (AC 5). New integration test.
8/8	Enforce ordering relative to mdbook-admonish if present (AC 6). Unit test on synthetic configs with admonish present / absent / already-correctly-ordered. Integration test in a fixture book with admonish registered after a stub preprocessor.
refactor	Optional.

Outside-in narrative

Slice 1 — failing integration test

The first slice introduces a CLI-level integration test that drives `install` against a minimal fixture book. The test body delegates setup and assertions to a `MinimalFixtureBook` helper so it reads as the scenario rather than the mechanics:

```
//! Integration tests for the Install the Preprocessor story (ch. 1).

use std::fs;
use std::path::{Path, PathBuf};

use tempfile::TempDir;

mod common;
use common::mdbook_listings;

#[test]
#[ignore = "passes once the install command is wired up to do real work"]
fn install_registers_preprocessor_and_writes_css() {
    let book = MinimalFixtureBook::new();

    mdbook_listings()
```

```
        .args(["install", "--book-root"])
        .arg(book.root())
        .assert()
        .success();

    book.assert_preprocessor_registered();
    book.assert_css_asset_present();
}

/// The smallest mdbook that's still a valid book: a `book.toml` declaring
/// just the `[book]` table with a title, materialised in a TempDir whose
/// lifetime is tied to this struct so the filesystem clean-up is automatic.
struct MinimalFixtureBook {
    _tmp: TempDir,
    root: PathBuf,
}

impl MinimalFixtureBook {
    fn new() -> Self {
        let tmp = TempDir::new().expect("tempdir");
        let root = tmp.path().to_path_buf();
        fs::write(root.join("book.toml"), "[book]\ntitle =
\"Test\"\n").unwrap();
        Self { _tmp: tmp, root }
    }

    fn root(&self) -> &Path {
        &self.root
    }

    fn assert_preprocessor_registered(&self) {
        let book_toml =
fs::read_to_string(self.root.join("book.toml")).unwrap();
        assert!(
            book_toml.contains("[preprocessor.listings]"),
            "book.toml should register the preprocessor; got:\n{book_toml}",
        );
        assert!(
            book_toml.contains("mdbook-listings.css"),
            "book.toml should reference the CSS asset; got:\n{book_toml}",
        );
    }

    fn assert_css_asset_present(&self) {
        assert!(
            self.root.join("mdbook-listings.css").exists(),
            "CSS asset should be written to the book root",
        );
    }
}
```

The test is `#[ignore]`'d so the green-build pre-commit chain stays passing while `install` is still a stub. It was run once locally first and confirmed to fail at the install invocation (`error: 'mdbook-listings install' is not yet implemented`); the ignore reason names the condition for unskipping. A later slice wires up the install handler and removes the ignore.

Slice 2 — bundle the CSS asset

Slice 2 introduces the first piece of code the integration test will eventually need: the CSS bytes that `install` will copy to the book root. The asset is compiled into the binary via [`include_bytes!`] so a `cargo install mdbook-listings` produces a self-contained binary with nothing external to fetch.

A new `install` module declares the constant and a sentinel string that unit tests assert is present in the bundled bytes (so a build that strips or replaces the asset fails loudly):

```
//! `install` subcommand: configures an existing book to use mdbook-listings.

/// Compiled in so `cargo install mdbook-listings` produces a self-contained
/// binary with nothing external to fetch at install time.
pub const CSS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.css");

/// Catches builds that stripped or replaced the asset — a missing sentinel
/// means the bundled bytes are not the expected build-time asset.
pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v1";

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn css_asset_is_non_empty() {
        assert!(!CSS_ASSET.is_empty(), "bundled CSS asset must not be empty");
    }

    #[test]
    fn css_asset_contains_sentinel() {
        let contents = std::str::from_utf8(CSS_ASSET).expect("CSS asset must be
UTF-8");
        assert!(
            contents.contains(CSS_ASSET_SENTINEL),
            "bundled CSS asset must contain sentinel `{CSS_ASSET_SENTINEL}`;
got:\n{contents}",
        );
    }
}
```

The asset itself is intentionally a placeholder — real callout styling depends on choices the **Render Inline Callouts** story (ch. 4) hasn't made yet. The placeholder carries only the sentinel string the unit tests look for:

```
/* mdbook-listings — placeholder CSS.
 *
 * The Render Inline Callouts story (ch. 4) will replace this with the real
 * styling for callout badges and <details> blocks. Until then the asset
 * exists only so the install pipeline has something to ship and so the
 * build-time bundling can be smoke-tested.
 *
 * Sentinel string used by unit tests to confirm the bundled bytes are the
```

```
 * expected build-time asset: mdbook-listings-css-v1
 */
```

`src/lib.rs` gains one line — `pub mod install;` — so the rest of the crate can reach the new module:

```
//! Managed code listings for mdbook.

pub mod freeze;
pub mod install;
pub mod manifest;
```

The unit tests run as part of the regular suite and pass; the integration test from slice 1 is still `#[ignore]`'d because `install` doesn't yet do anything with the bundled asset.

Slice 3 — TOML round-trip primitive

Slice 3 stands up the primitive that lets later slices mutate `book.toml` while preserving its formatting: a `BookConfig` newtype around `toml_edit::DocumentMut`. Two unit tests pin the guarantees the wrapper has to keep — round-tripping a config without mutation is byte-identical to the input (preserving comments and entry ordering), and invalid TOML is rejected with a diagnostic.

`Cargo.toml` gains `toml_edit` as a runtime dep:

```
[package]
name = "mdbook-listings"
version = "0.1.0"
edition = "2024"
rust-version = "1.88"
license = "MIT"
description = "Managed code listings for mdbook: inline callouts, freezing, and
verification"
repository = "https://github.com/padamson/mdbook-listings"
categories = ["command-line-utilities", "text-processing"]
keywords = ["mdbook", "preprocessor", "documentation", "code-listing"]

[dependencies]
anyhow = "1"
clap = { version = "4", features = ["derive"] }
serde = { version = "1", features = ["derive"] }
sha2 = "0.11"
toml = "1.1"
toml_edit = "0.22"

[dev-dependencies]
assert_cmd = "2"
predicates = "3"
tempfile = "3"
```

The install module now declares the primitive alongside the CSS asset bundling from slice 2. **What's new in `install-v2` compared to `install-v1`:** the `BookConfig` struct (with `#[derive(Debug)]` so test failures format readably), its `parse` and `render`

methods, two new tests (`book_config_round_trip_preserves_comments_and_ordering` and `book_config_parse_rejects_invalid_toml`), and the imports those need (`anyhow::{Context, Result}`, `toml_edit::DocumentMut`). Everything else — the CSS constants and their tests — is unchanged from `install-v1`.

```rust
//! `install` subcommand: configures an existing book to use mdbook-listings.

use anyhow::{Context, Result};
use toml_edit::DocumentMut;

/// Compiled in so `cargo install mdbook-listings` produces a self-contained
/// binary with nothing external to fetch at install time.
pub const CSS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.css");

/// Catches builds that stripped or replaced the asset — a missing sentinel
/// means the bundled bytes are not the expected build-time asset.
pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v1";

/// Newtype over [`toml_edit::DocumentMut`] so future install methods
/// (register preprocessor, add additional-css) have a domain type to
/// attach to and so callers don't depend on `toml_edit` directly.
#[derive(Debug)]
pub struct BookConfig(DocumentMut);

impl BookConfig {
    pub fn parse(s: &str) -> Result<Self> {
        s.parse::<DocumentMut>()
            .map(BookConfig)
            .context("book config is not valid TOML")
    }

    pub fn render(&self) -> String {
        self.0.to_string()
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn css_asset_is_non_empty() {
        assert!(!CSS_ASSET.is_empty(), "bundled CSS asset must not be empty");
    }

    #[test]
    fn css_asset_contains_sentinel() {
        let contents = std::str::from_utf8(CSS_ASSET).expect("CSS asset must be
UTF-8");
        assert!(
            contents.contains(CSS_ASSET_SENTINEL),
            "bundled CSS asset must contain sentinel `{CSS_ASSET_SENTINEL}`;
got:\n{contents}",
        );
    }
```

```
    #[test]
    fn book_config_round_trip_preserves_comments_and_ordering() {
        let input = "\
# top comment
[book]
title = \"Test\"

# preprocessor comment
[preprocessor.admonish]
command = \"mdbook-admonish\"

[output.html]
";
        let cfg = BookConfig::parse(input).expect("parse");
        assert_eq!(cfg.render(), input);
    }

    #[test]
    fn book_config_parse_rejects_invalid_toml() {
        let err = BookConfig::parse("[book\nbroken = ").unwrap_err();
        let msg = format!("{err:#}");
        assert!(
            msg.contains("not valid TOML"),
            "diagnostic should name the failure mode; got: {msg}"
        );
    }
}
```

The integration test from slice 1 is still `#[ignore]`'d. `BookConfig` is plumbing — slice 4 wires it up to add the `[preprocessor.listings]` registration that satisfies the test's first assertion.

Slice 4 — register the `[preprocessor.listings]` entry

Slice 4 adds the `BookConfig` method that satisfies the chunk of AC 1 visible from `book.toml`: a `[preprocessor.listings]` entry with `command = "mdbook-listings"`. Two unit tests pin (a) that the entry is added with the right command value and (b) that the operation is idempotent — a second call on an already-registered config produces identical rendered output (this is the unit-test form of AC 3).

What's new in `install-v3` compared to `install-v2`: the `register_listings_preprocessor` method on `BookConfig`, the `Item`, `Table` imports it needs from `toml_edit`, and two new tests (`book_config_register_listings_preprocessor_adds_entry` and `book_config_register_listings_preprocessor_is_idempotent`). Everything else — the CSS constants, the `BookConfig` parse and render methods, and their tests — is unchanged from `install-v2`.

```
//! `install` subcommand: configures an existing book to use mdbook-listings.

use anyhow::{Context, Result};
use toml_edit::{DocumentMut, Item, Table};

/// Compiled in so `cargo install mdbook-listings` produces a self-contained
```

```rust
/// binary with nothing external to fetch at install time.
pub const CSS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.css");

/// Catches builds that stripped or replaced the asset — a missing sentinel
/// means the bundled bytes are not the expected build-time asset.
pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v1";

/// Newtype over [`toml_edit::DocumentMut`] so future install methods
/// (register preprocessor, add additional-css) have a domain type to
/// attach to and so callers don't depend on `toml_edit` directly.
#[derive(Debug)]
pub struct BookConfig(DocumentMut);

impl BookConfig {
    pub fn parse(s: &str) -> Result<Self> {
        s.parse::<DocumentMut>()
            .map(BookConfig)
            .context("book config is not valid TOML")
    }

    pub fn render(&self) -> String {
        self.0.to_string()
    }

    /// Add (or confirm the presence of) `[preprocessor.listings]` with
    /// `command = "mdbook-listings"`. Idempotent — a second call on an
    /// already-registered config produces identical rendered output.
    pub fn register_listings_preprocessor(&mut self) {
        let preprocessor = self
            .0
            .as_table_mut()
            .entry("preprocessor")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor] must be a table");
        let listings = preprocessor
            .entry("listings")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor.listings] must be a table");
        listings["command"] = toml_edit::value("mdbook-listings");
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn css_asset_is_non_empty() {
        assert!(!CSS_ASSET.is_empty(), "bundled CSS asset must not be empty");
    }

    #[test]
    fn css_asset_contains_sentinel() {
```

```
        let contents = std::str::from_utf8(CSS_ASSET).expect("CSS asset must be
UTF-8");
        assert!(
            contents.contains(CSS_ASSET_SENTINEL),
            "bundled CSS asset must contain sentinel `{CSS_ASSET_SENTINEL}`;
got:\n{contents}",
        );
    }

    #[test]
    fn book_config_round_trip_preserves_comments_and_ordering() {
        let input = "\
# top comment
[book]
title = \"Test\"

# preprocessor comment
[preprocessor.admonish]
command = \"mdbook-admonish\"

[output.html]
";
        let cfg = BookConfig::parse(input).expect("parse");
        assert_eq!(cfg.render(), input);
    }

    #[test]
    fn book_config_parse_rejects_invalid_toml() {
        let err = BookConfig::parse("[book\nbroken = ").unwrap_err();
        let msg = format!("{err:#}");
        assert!(
            msg.contains("not valid TOML"),
            "diagnostic should name the failure mode; got: {msg}"
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[preprocessor.listings]"),
            "rendered config should declare [preprocessor.listings]; got:
\n{rendered}",
        );
        assert!(
            rendered.contains(r#"command = "mdbook-listings""#),
            "rendered config should set command = \"mdbook-listings\"; got:
\n{rendered}",
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_is_idempotent() {
```

```
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_preprocessor();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_preprocessor();
        let after_second = cfg2.render();

        assert_eq!(after_first, after_second, "register must be idempotent");
    }
}
```

The integration test from slice 1 is still `#[ignore]`'d. The register method handles the `[preprocessor.listings]` half of the post-install disk state; slice 5 adds the matching `additional-css` registration for the CSS asset, and slice 6 wires both into the install handler so the integration test goes green.

Slice 5 — copy the CSS asset and register it

Slice 5 covers the other half of AC 2: `[output.html]` gets `additional-css = ["./mdbook-listings.css"]` so mdbook's HTML build picks the asset up, and the on-disk copy of the asset itself lands at `<book-root>/mdbook-listings.css`.

The TOML mutation is a `BookConfig::register_listings_css` method on the same newtype as the preprocessor registration; the file copy is a free function `write_css_asset(book_root)` that writes `[CSS_ASSET]` (from slice 2) to the conventional filename. A new `CSS_ASSET_FILENAME` constant ties the two together so they can't drift out of sync. Two unit tests pin the additional-css side: the entry is added with the right relative path, and the operation is idempotent (no duplicate entries on a second call).

What's new in `install-v4` compared to `install-v3`: the `CSS_ASSET_FILENAME` constant, the `write_css_asset` free function, the `register_listings_css` method on `BookConfig`, two new tests (`book_config_register_listings_css_adds_entry`, `book_config_register_listings_css_is_idempotent`), and the imports they need (`std::fs`, `std::path::Path`, plus `toml_edit::{Array, Value}` added to the existing import line). Everything else — the CSS constants, the parse/render methods, the preprocessor-registration method, and their tests — is unchanged from `install-v3`.

```
//! `install` subcommand: configures an existing book to use mdbook-listings.

use std::fs;
use std::path::Path;

use anyhow::{Context, Result};
use toml_edit::{Array, DocumentMut, Item, Table, Value};

/// Compiled in so `cargo install mdbook-listings` produces a self-contained
/// binary with nothing external to fetch at install time.
pub const CSS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.css");

/// Catches builds that stripped or replaced the asset — a missing sentinel
/// means the bundled bytes are not the expected build-time asset.
```

```rust
pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v1";

/// Filename the CSS asset is written under at install time. Shared between
/// the file-copy side and the `[output.html].additional-css` entry so the
/// two can't drift.
pub const CSS_ASSET_FILENAME: &str = "mdbook-listings.css";

/// Write the bundled [`CSS_ASSET`] to `<book_root>/<CSS_ASSET_FILENAME>`,
/// creating or overwriting the file. Existing content is replaced
/// unconditionally — this is the "ship the version this binary was
/// built with" semantics the install command wants.
pub fn write_css_asset(book_root: &Path) -> Result<()> {
    let path = book_root.join(CSS_ASSET_FILENAME);
    fs::write(&path, CSS_ASSET).with_context(|| format!("writing CSS asset to
{}", path.display()))
}

/// Newtype over [`toml_edit::DocumentMut`] so future install methods
/// (register preprocessor, add additional-css) have a domain type to
/// attach to and so callers don't depend on `toml_edit` directly.
#[derive(Debug)]
pub struct BookConfig(DocumentMut);

impl BookConfig {
    pub fn parse(s: &str) -> Result<Self> {
        s.parse::<DocumentMut>()
            .map(BookConfig)
            .context("book config is not valid TOML")
    }

    pub fn render(&self) -> String {
        self.0.to_string()
    }

    /// Add (or confirm the presence of) `[preprocessor.listings]` with
    /// `command = "mdbook-listings"`. Idempotent — a second call on an
    /// already-registered config produces identical rendered output.
    pub fn register_listings_preprocessor(&mut self) {
        let preprocessor = self
            .0
            .as_table_mut()
            .entry("preprocessor")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor] must be a table");
        let listings = preprocessor
            .entry("listings")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor.listings] must be a table");
        listings["command"] = toml_edit::value("mdbook-listings");
    }

    /// Add `./<CSS_ASSET_FILENAME>` to `[output.html].additional-css`,
    /// creating the section + array as needed. Idempotent — duplicate
```

```
    /// entries are not appended.
    pub fn register_listings_css(&mut self) {
        let entry = format!("./{CSS_ASSET_FILENAME}");
        let array = self
            .0
            .as_table_mut()
            .entry("output")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[output] must be a table")
            .entry("html")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[output.html] must be a table")
            .entry("additional-css")
            .or_insert_with(|| Item::Value(Value::Array(Array::new())))
            .as_value_mut()
            .expect("additional-css must be a value")
            .as_array_mut()
            .expect("additional-css must be an array");
        if !array.iter().any(|v| v.as_str() == Some(entry.as_str())) {
            array.push(entry);
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn css_asset_is_non_empty() {
        assert!(!CSS_ASSET.is_empty(), "bundled CSS asset must not be empty");
    }

    #[test]
    fn css_asset_contains_sentinel() {
        let contents = std::str::from_utf8(CSS_ASSET).expect("CSS asset must be
UTF-8");
        assert!(
            contents.contains(CSS_ASSET_SENTINEL),
            "bundled CSS asset must contain sentinel `{CSS_ASSET_SENTINEL}`;
got:\n{contents}",
        );
    }

    #[test]
    fn book_config_round_trip_preserves_comments_and_ordering() {
        let input = "\
# top comment
[book]
title = \"Test\"

# preprocessor comment
[preprocessor.admonish]
```

```
command = \"mdbook-admonish\"

[output.html]
";
        let cfg = BookConfig::parse(input).expect("parse");
        assert_eq!(cfg.render(), input);
    }

    #[test]
    fn book_config_parse_rejects_invalid_toml() {
        let err = BookConfig::parse("[book\nbroken = ").unwrap_err();
        let msg = format!("{err:#}");
        assert!(
            msg.contains("not valid TOML"),
            "diagnostic should name the failure mode; got: {msg}"
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[preprocessor.listings]"),
            "rendered config should declare [preprocessor.listings]; got:
\n{rendered}",
        );
        assert!(
            rendered.contains(r#"command = "mdbook-listings""#),
            "rendered config should set command = \"mdbook-listings\"; got:
\n{rendered}",
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_preprocessor();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_preprocessor();
        let after_second = cfg2.render();

        assert_eq!(after_first, after_second, "register must be idempotent");
    }

    #[test]
    fn book_config_register_listings_css_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_css();
        let rendered = cfg.render();
        assert!(
```

```
            rendered.contains("[output.html]"),
            "rendered config should declare [output.html]; got:\n{rendered}",
        );
        assert!(
            rendered.contains(r#"additional-css = ["./mdbook-listings.css"]"#),
            "rendered config should reference the CSS asset; got:\n{rendered}",
        );
    }

    #[test]
    fn book_config_register_listings_css_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_css();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_css();
        let after_second = cfg2.render();

        assert_eq!(
            after_first, after_second,
            "register_listings_css must be idempotent"
        );
    }
}
```

The integration test from slice 1 is still `#[ignore]`'d. Slice 5 finishes the building blocks; slice 6 sequences `BookConfig::parse → register_listings_preprocessor → register_listings_css → render → write back to book.toml`, plus a `write_css_asset` call, behind the `install` CLI handler — at which point the integration test passes for ACs 1 + 2.

Slice 6 — wire the install handler

Slice 6 sequences the building blocks from slices 2–5 behind the CLI: an `install(book_root)` orchestrator function in `src/install.rs` reads `book.toml`, parses it, calls `register_listings_preprocessor` and `register_listings_css`, writes the file back if anything changed, and writes the CSS asset if the on-disk copy differs from the bundled bytes. It returns an `InstallOutcome` enum (`Installed` or `Unchanged`) so the CLI can confirm to the author whether the run was a no-op (the user-visible half of AC 3).

`src/main.rs`'s `Command::Install` arm calls into the orchestrator and prints either "installed mdbook-listings into …" or "mdbook-listings already installed in …; nothing changed" based on the outcome.

`tests/install.rs` drops the `#[ignore]` attribute on `install_registers_preprocessor_and_writes_css`. The test now runs as part of the regular suite and passes — confirming ACs 1 + 2 end-to-end.

What's new in `install-v5` compared to `install-v4`: the `install` orchestrator function and the `InstallOutcome` enum. The bodies of constants, `write_css_asset`, the `BookConfig` methods, and all the existing tests are unchanged. Doc comments throughout were trimmed to a why-only style — restating function names or describing the body in prose is dropped — but the code itself is the same as `install-v4`.

```rust
//! `install` subcommand: configures an existing book to use mdbook-listings.

use std::fs;
use std::path::Path;

use anyhow::{Context, Result};
use toml_edit::{Array, DocumentMut, Item, Table, Value};

/// Compiled in so `cargo install mdbook-listings` produces a self-contained
/// binary with nothing external to fetch at install time.
pub const CSS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.css");

/// Catches builds that stripped or replaced the asset — a missing sentinel
/// means the bundled bytes are not the expected build-time asset.
pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v1";

/// Shared between [`write_css_asset`] and
/// [`BookConfig::register_listings_css`] so the two can't drift.
pub const CSS_ASSET_FILENAME: &str = "mdbook-listings.css";

/// Always overwrites — install ships the bundled bytes, not whatever a
/// stale on-disk copy happens to contain.
pub fn write_css_asset(book_root: &Path) -> Result<()> {
    let path = book_root.join(CSS_ASSET_FILENAME);
    fs::write(&path, CSS_ASSET).with_context(|| format!("writing CSS asset to
{}", path.display()))
}

/// Idempotent: book.toml and the CSS asset on disk are only rewritten if
/// they differ from what install would produce.
pub fn install(book_root: &Path) -> Result<InstallOutcome> {
    let book_toml_path = book_root.join("book.toml");
    let original = fs::read_to_string(&book_toml_path)
        .with_context(|| format!("reading book config at {}",
book_toml_path.display()))?;
    let mut config = BookConfig::parse(&original)?;
    config.register_listings_preprocessor();
    config.register_listings_css();
    let new = config.render();

    let css_path = book_root.join(CSS_ASSET_FILENAME);
    let css_already_correct = fs::read(&css_path)
        .ok()
        .is_some_and(|bytes| bytes.as_slice() == CSS_ASSET);

    let toml_changed = new != original;
    if toml_changed {
        fs::write(&book_toml_path, new)
            .with_context(|| format!("writing book config at {}",
book_toml_path.display()))?;
    }
    if !css_already_correct {
        write_css_asset(book_root)?;
    }
```

```
    Ok(if toml_changed || !css_already_correct {
        InstallOutcome::Installed
    } else {
        InstallOutcome::Unchanged
    })
}

/// Lets the CLI tell the author whether a re-install was a no-op (AC 3).
#[derive(Debug, PartialEq, Eq)]
pub enum InstallOutcome {
    Installed,
    Unchanged,
}

/// Newtype over [`toml_edit::DocumentMut`] so callers don't depend on
/// `toml_edit` directly and the `register_*` methods have a domain type
/// to attach to.
#[derive(Debug)]
pub struct BookConfig(DocumentMut);

impl BookConfig {
    pub fn parse(s: &str) -> Result<Self> {
        s.parse::<DocumentMut>()
            .map(BookConfig)
            .context("book config is not valid TOML")
    }

    pub fn render(&self) -> String {
        self.0.to_string()
    }

    /// Idempotent: a second call on an already-registered config is a no-op
    /// in the rendered output.
    pub fn register_listings_preprocessor(&mut self) {
        let preprocessor = self
            .0
            .as_table_mut()
            .entry("preprocessor")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor] must be a table");
        let listings = preprocessor
            .entry("listings")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor.listings] must be a table");
        listings["command"] = toml_edit::value("mdbook-listings");
    }

    /// Idempotent: duplicate entries are not appended.
    pub fn register_listings_css(&mut self) {
        let entry = format!("./{CSS_ASSET_FILENAME}");
        let array = self
            .0
            .as_table_mut()
```

```
            .entry("output")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[output] must be a table")
            .entry("html")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[output.html] must be a table")
            .entry("additional-css")
            .or_insert_with(|| Item::Value(Value::Array(Array::new())))
            .as_value_mut()
            .expect("additional-css must be a value")
            .as_array_mut()
            .expect("additional-css must be an array");
        if !array.iter().any(|v| v.as_str() == Some(entry.as_str())) {
            array.push(entry);
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn css_asset_is_non_empty() {
        assert!(!CSS_ASSET.is_empty(), "bundled CSS asset must not be empty");
    }

    #[test]
    fn css_asset_contains_sentinel() {
        let contents = std::str::from_utf8(CSS_ASSET).expect("CSS asset must be
UTF-8");
        assert!(
            contents.contains(CSS_ASSET_SENTINEL),
            "bundled CSS asset must contain sentinel `{CSS_ASSET_SENTINEL}`;
got:\n{contents}",
        );
    }

    #[test]
    fn book_config_round_trip_preserves_comments_and_ordering() {
        let input = "\
# top comment
[book]
title = \"Test\"

# preprocessor comment
[preprocessor.admonish]
command = \"mdbook-admonish\"

[output.html]
";
        let cfg = BookConfig::parse(input).expect("parse");
        assert_eq!(cfg.render(), input);
```

```
    }

    #[test]
    fn book_config_parse_rejects_invalid_toml() {
        let err = BookConfig::parse("[book\nbroken = ").unwrap_err();
        let msg = format!("{err:#}");
        assert!(
            msg.contains("not valid TOML"),
            "diagnostic should name the failure mode; got: {msg}"
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[preprocessor.listings]"),
            "rendered config should declare [preprocessor.listings]; got:
\n{rendered}",
        );
        assert!(
            rendered.contains(r#"command = "mdbook-listings""#),
            "rendered config should set command = \"mdbook-listings\"; got:
\n{rendered}",
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_preprocessor();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_preprocessor();
        let after_second = cfg2.render();

        assert_eq!(after_first, after_second, "register must be idempotent");
    }

    #[test]
    fn book_config_register_listings_css_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_css();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[output.html]"),
            "rendered config should declare [output.html]; got:\n{rendered}",
        );
        assert!(
            rendered.contains(r#"additional-css = ["./mdbook-listings.css"]"#),
            "rendered config should reference the CSS asset; got:\n{rendered}",
```

```
        );
    }

    #[test]
    fn book_config_register_listings_css_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_css();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_css();
        let after_second = cfg2.render();

        assert_eq!(
            after_first, after_second,
            "register_listings_css must be idempotent"
        );
    }
}
```

What's new in `main-v2` compared to `main-v1`: the `Command::Install` arm now calls `mdbook_listings::install::install`, branches on `InstallOutcome`, and prints one of two messages. The `use mdbook_listings::install::{InstallOutcome, install};` import is added. Everything else — the other subcommands (`Supports`, `Freeze`, `Verify`, the no-subcommand preprocessor stub) and the `main`/`run`/`supports` functions — is unchanged from `main-v1`.

```
use std::path::PathBuf;
use std::process;

use anyhow::Result;
use clap::{Parser, Subcommand};
use mdbook_listings::freeze::{FreezeOptions, FreezeOutcome, freeze};
use mdbook_listings::install::{InstallOutcome, install};

/// Managed code listings for mdbook: inline callouts, freezing, and
verification.
#[derive(Parser)]
#[command(version, about, long_about = None)]
struct Cli {
    #[command(subcommand)]
    command: Option<Command>,
}

#[derive(Subcommand)]
enum Command {
    /// Check whether a renderer is supported by this preprocessor.
    ///
    /// Invoked by mdbook during the build to decide whether to pipe the book
    /// through this preprocessor for a given renderer. Exits 0 if supported,
    /// 1 otherwise.
    Supports {
        /// Name of the renderer mdbook is asking about (e.g. `html`, `typst-
```

```
pdf`).
        renderer: String,
    },

    /// Install preprocessor assets and register mdbook-listings in `book.toml`.
    Install {
        /// Root directory of the book (contains `book.toml`). Defaults to the
        /// current directory.
        #[arg(long)]
        book_root: Option<PathBuf>,
    },

    /// Freeze a source file into the book's listings directory and update
    /// the manifest.
    Freeze {
        /// Human-readable tag used as the frozen filename and as the manifest
        /// entry key. Should be unique within the book.
        #[arg(long)]
        tag: String,

        /// Root directory of the book. Defaults to the current directory.
        #[arg(long)]
        book_root: Option<PathBuf>,

        /// Overwrite an existing frozen copy with the same tag.
        #[arg(long)]
        force: bool,

        /// Path to the source file to freeze.
        source: PathBuf,
    },

    /// Verify consistency between the manifest, frozen listings, and
`{{#include}}`
    /// references in the book's markdown.
    Verify {
        /// Root directory of the book. Defaults to the current directory.
        #[arg(long)]
        book_root: Option<PathBuf>,
    },
}

fn main() {
    if let Err(err) = run() {
        eprintln!("error: {err:?}");
        process::exit(1);
    }
}

fn run() -> Result<()> {
    let cli = Cli::parse();
    match cli.command {
        None => preprocess(),
        Some(Command::Supports { renderer }) => supports(&renderer),
        Some(Command::Install { book_root }) => {
```

```
            let book_root = book_root.unwrap_or_else(|| PathBuf::from("."));
            match install(&book_root)? {
                InstallOutcome::Installed => {
                    println!("installed mdbook-listings into {}",
book_root.display());
                }
                InstallOutcome::Unchanged => {
                    println!(
                        "mdbook-listings already installed in {}; nothing
changed",
                        book_root.display(),
                    );
                }
            }
            Ok(())
        }
        Some(Command::Freeze {
            tag,
            book_root,
            force,
            source,
        }) => {
            let book_root = book_root.unwrap_or_else(|| PathBuf::from("."));
            let outcome = freeze(FreezeOptions {
                book_root: &book_root,
                tag: &tag,
                source: &source,
                force,
            })?;
            let verb = match outcome {
                FreezeOutcome::Created => "created",
                FreezeOutcome::Unchanged => "unchanged",
                FreezeOutcome::Replaced => "replaced",
            };
            println!("{verb}: {tag}");
            Ok(())
        }
        Some(Command::Verify { book_root: _ }) => {
            anyhow::bail!("`mdbook-listings verify` is not yet implemented")
        }
    }
}

/// Default mode: read an mdbook preprocessor JSON payload from stdin, emit the
/// transformed payload on stdout.
fn preprocess() -> Result<()> {
    anyhow::bail!("mdbook-listings preprocessor mode is not yet implemented")
}

/// Answer mdbook's renderer-support probe by exiting 0 (supported) or 1
/// (unsupported). We do not return from this function.
fn supports(renderer: &str) -> ! {
    let supported = matches!(renderer, "html" | "typst-pdf");
    process::exit(if supported { 0 } else { 1 });
}
```

What's new in `install-tests-v2` compared to `install-tests-v1`: the `#[ignore = "..."]` attribute is removed; the `#[test]` attribute and the body are unchanged.

```
//! Integration tests for the Install the Preprocessor story (ch. 1).

use std::fs;
use std::path::{Path, PathBuf};

use tempfile::TempDir;

mod common;
use common::mdbook_listings;

#[test]
fn install_registers_preprocessor_and_writes_css() {
    let book = MinimalFixtureBook::new();

    mdbook_listings()
        .args(["install", "--book-root"])
        .arg(book.root())
        .assert()
        .success();

    book.assert_preprocessor_registered();
    book.assert_css_asset_present();
}

/// The smallest mdbook that's still a valid book: a `book.toml` declaring
/// just the `[book]` table with a title, materialised in a TempDir whose
/// lifetime is tied to this struct so the filesystem clean-up is automatic.
struct MinimalFixtureBook {
    _tmp: TempDir,
    root: PathBuf,
}

impl MinimalFixtureBook {
    fn new() -> Self {
        let tmp = TempDir::new().expect("tempdir");
        let root = tmp.path().to_path_buf();
        fs::write(root.join("book.toml"), "[book]\ntitle =
\"Test\"\n").unwrap();
        Self { _tmp: tmp, root }
    }

    fn root(&self) -> &Path {
        &self.root
    }

    fn assert_preprocessor_registered(&self) {
        let book_toml =
fs::read_to_string(self.root.join("book.toml")).unwrap();
        assert!(
            book_toml.contains("[preprocessor.listings]"),
            "book.toml should register the preprocessor; got:\n{book_toml}",
        );
```

```
        assert!(
            book_toml.contains("mdbook-listings.css"),
            "book.toml should reference the CSS asset; got:\n{book_toml}",
        );
    }

    fn assert_css_asset_present(&self) {
        assert!(
            self.root.join("mdbook-listings.css").exists(),
            "CSS asset should be written to the book root",
        );
    }
}
```

The integration test from slice 1 is no longer ignored. Slices 7 and 8 add the remaining ACs (missing-config diagnostic for AC 5, mdbook-admonish ordering for AC 6).

Slice 7 — reject missing book config

Slice 7 makes the missing-`book.toml` case fail with a helpful diagnostic instead of a generic "reading book config" wrapper. Before slice 7, running install in a directory with no `book.toml` produced something like:

```
error: reading book config at /path/to/book.toml
Caused by: No such file or directory (os error 2)
```

After slice 7:

```
error: book.toml not found at /path/to/book.toml — install requires
an existing mdbook book directory; run `mdbook init` first.
```

A new integration test `install_rejects_missing_book_config` runs install against a fresh `TempDir` (no `book.toml`) and asserts the binary exits non-zero with `book.toml not found` in stderr.

What's new in `install-v6` compared to `install-v5`: the `fs::read_to_string` call inside `install` is now a `match` that special-cases `io::ErrorKind::NotFound` with a tailored diagnostic; other I/O errors still go through the existing `with_context` path. Everything else — every other function, constant, struct, and test — is unchanged.

```
//! `install` subcommand: configures an existing book to use mdbook-listings.

use std::fs;
use std::path::Path;

use anyhow::{Context, Result};
use toml_edit::{Array, DocumentMut, Item, Table, Value};

/// Compiled in so `cargo install mdbook-listings` produces a self-contained
/// binary with nothing external to fetch at install time.
pub const CSS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.css");
```

```
/// Catches builds that stripped or replaced the asset — a missing sentinel
/// means the bundled bytes are not the expected build-time asset.
pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v1";

/// Shared between [`write_css_asset`] and
/// [`BookConfig::register_listings_css`] so the two can't drift.
pub const CSS_ASSET_FILENAME: &str = "mdbook-listings.css";

/// Always overwrites — install ships the bundled bytes, not whatever a
/// stale on-disk copy happens to contain.
pub fn write_css_asset(book_root: &Path) -> Result<()> {
    let path = book_root.join(CSS_ASSET_FILENAME);
    fs::write(&path, CSS_ASSET).with_context(|| format!("writing CSS asset to
{}", path.display()))
}

/// Idempotent: book.toml and the CSS asset on disk are only rewritten if
/// they differ from what install would produce.
pub fn install(book_root: &Path) -> Result<InstallOutcome> {
    let book_toml_path = book_root.join("book.toml");
    let original = match fs::read_to_string(&book_toml_path) {
        Ok(s) => s,
        Err(e) if e.kind() == std::io::ErrorKind::NotFound => {
            anyhow::bail!(
                "book.toml not found at {} — install requires an existing mdbook
book directory; run `mdbook init` first.",
                book_toml_path.display(),
            );
        }
        Err(e) => {
            return Err(anyhow::Error::from(e))
                .with_context(|| format!("reading book config at {}",
book_toml_path.display()));
        }
    };
    let mut config = BookConfig::parse(&original)?;
    config.register_listings_preprocessor();
    config.register_listings_css();
    let new = config.render();

    let css_path = book_root.join(CSS_ASSET_FILENAME);
    let css_already_correct = fs::read(&css_path)
        .ok()
        .is_some_and(|bytes| bytes.as_slice() == CSS_ASSET);

    let toml_changed = new != original;
    if toml_changed {
        fs::write(&book_toml_path, new)
            .with_context(|| format!("writing book config at {}",
book_toml_path.display()))?;
    }
    if !css_already_correct {
        write_css_asset(book_root)?;
    }
```

```
    Ok(if toml_changed || !css_already_correct {
        InstallOutcome::Installed
    } else {
        InstallOutcome::Unchanged
    })
}

/// Lets the CLI tell the author whether a re-install was a no-op (AC 3).
#[derive(Debug, PartialEq, Eq)]
pub enum InstallOutcome {
    Installed,
    Unchanged,
}

/// Newtype over [`toml_edit::DocumentMut`] so callers don't depend on
/// `toml_edit` directly and the `register_*` methods have a domain type
/// to attach to.
#[derive(Debug)]
pub struct BookConfig(DocumentMut);

impl BookConfig {
    pub fn parse(s: &str) -> Result<Self> {
        s.parse::<DocumentMut>()
            .map(BookConfig)
            .context("book config is not valid TOML")
    }

    pub fn render(&self) -> String {
        self.0.to_string()
    }

    /// Idempotent: a second call on an already-registered config is a no-op
    /// in the rendered output.
    pub fn register_listings_preprocessor(&mut self) {
        let preprocessor = self
            .0
            .as_table_mut()
            .entry("preprocessor")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor] must be a table");
        let listings = preprocessor
            .entry("listings")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor.listings] must be a table");
        listings["command"] = toml_edit::value("mdbook-listings");
    }

    /// Idempotent: duplicate entries are not appended.
    pub fn register_listings_css(&mut self) {
        let entry = format!("./{CSS_ASSET_FILENAME}");
        let array = self
            .0
            .as_table_mut()
```

```
            .entry("output")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[output] must be a table")
            .entry("html")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[output.html] must be a table")
            .entry("additional-css")
            .or_insert_with(|| Item::Value(Value::Array(Array::new())))
            .as_value_mut()
            .expect("additional-css must be a value")
            .as_array_mut()
            .expect("additional-css must be an array");
        if !array.iter().any(|v| v.as_str() == Some(entry.as_str())) {
            array.push(entry);
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn css_asset_is_non_empty() {
        assert!(!CSS_ASSET.is_empty(), "bundled CSS asset must not be empty");
    }

    #[test]
    fn css_asset_contains_sentinel() {
        let contents = std::str::from_utf8(CSS_ASSET).expect("CSS asset must be
UTF-8");
        assert!(
            contents.contains(CSS_ASSET_SENTINEL),
            "bundled CSS asset must contain sentinel `{CSS_ASSET_SENTINEL}`;
got:\n{contents}",
        );
    }

    #[test]
    fn book_config_round_trip_preserves_comments_and_ordering() {
        let input = "\
# top comment
[book]
title = \"Test\"

# preprocessor comment
[preprocessor.admonish]
command = \"mdbook-admonish\"

[output.html]
";
        let cfg = BookConfig::parse(input).expect("parse");
        assert_eq!(cfg.render(), input);
```

```
    }

    #[test]
    fn book_config_parse_rejects_invalid_toml() {
        let err = BookConfig::parse("[book\nbroken = ").unwrap_err();
        let msg = format!("{err:#}");
        assert!(
            msg.contains("not valid TOML"),
            "diagnostic should name the failure mode; got: {msg}"
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[preprocessor.listings]"),
            "rendered config should declare [preprocessor.listings]; got:
\n{rendered}",
        );
        assert!(
            rendered.contains(r#"command = "mdbook-listings""#),
            "rendered config should set command = \"mdbook-listings\"; got:
\n{rendered}",
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_preprocessor();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_preprocessor();
        let after_second = cfg2.render();

        assert_eq!(after_first, after_second, "register must be idempotent");
    }

    #[test]
    fn book_config_register_listings_css_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_css();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[output.html]"),
            "rendered config should declare [output.html]; got:\n{rendered}",
        );
        assert!(
            rendered.contains(r#"additional-css = ["./mdbook-listings.css"]"#),
            "rendered config should reference the CSS asset; got:\n{rendered}",
```

```
        );
    }

    #[test]
    fn book_config_register_listings_css_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_css();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_css();
        let after_second = cfg2.render();

        assert_eq!(
            after_first, after_second,
            "register_listings_css must be idempotent"
        );
    }
}
```

What's new in `install-tests-v3` compared to `install-tests-v2`: the `predicates::str::contains` import and the new `install_rejects_missing_book_config` test. The existing test (`install_registers_preprocessor_and_writes_css`) and the `MinimalFixtureBook` helper are unchanged.

```
//! Integration tests for the Install the Preprocessor story (ch. 1).

use std::fs;
use std::path::{Path, PathBuf};

use predicates::str::contains;
use tempfile::TempDir;

mod common;
use common::mdbook_listings;

#[test]
fn install_registers_preprocessor_and_writes_css() {
    let book = MinimalFixtureBook::new();

    mdbook_listings()
        .args(["install", "--book-root"])
        .arg(book.root())
        .assert()
        .success();

    book.assert_preprocessor_registered();
    book.assert_css_asset_present();
}

/// The smallest mdbook that's still a valid book: a `book.toml` declaring
/// just the `[book]` table with a title, materialised in a TempDir whose
```

```
/// lifetime is tied to this struct so the filesystem clean-up is automatic.
struct MinimalFixtureBook {
    _tmp: TempDir,
    root: PathBuf,
}

impl MinimalFixtureBook {
    fn new() -> Self {
        let tmp = TempDir::new().expect("tempdir");
        let root = tmp.path().to_path_buf();
        fs::write(root.join("book.toml"), "[book]\ntitle =
\"Test\"\n").unwrap();
        Self { _tmp: tmp, root }
    }

    fn root(&self) -> &Path {
        &self.root
    }

    fn assert_preprocessor_registered(&self) {
        let book_toml =
fs::read_to_string(self.root.join("book.toml")).unwrap();
        assert!(
            book_toml.contains("[preprocessor.listings]"),
            "book.toml should register the preprocessor; got:\n{book_toml}",
        );
        assert!(
            book_toml.contains("mdbook-listings.css"),
            "book.toml should reference the CSS asset; got:\n{book_toml}",
        );
    }

    fn assert_css_asset_present(&self) {
        assert!(
            self.root.join("mdbook-listings.css").exists(),
            "CSS asset should be written to the book root",
        );
    }
}

/// `install` against a directory with no `book.toml` exits non-zero with a
/// diagnostic identifying what was expected (AC 5).
#[test]
fn install_rejects_missing_book_config() {
    let tmp = TempDir::new().expect("tempdir");

    mdbook_listings()
        .args(["install", "--book-root"])
        .arg(tmp.path())
        .assert()
        .failure()
        .stderr(contains("book.toml not found"));
}
```

The suite now runs 24 tests (10 install-related, 14 from other modules). Slice 8 takes care of AC 6 (mdbook-admonish ordering).

Slice 8 — order before mdbook-admonish

Slice 8 satisfies AC 6: when `[preprocessor.admonish]` is already registered in `book.toml`, install adds `before = ["admonish"]` to the new `[preprocessor.listings]` entry so mdbook runs listings first. The callout → admonish-note pipeline (which the **Render Inline Callouts** story will rely on for PDF output) requires this ordering.

The behaviour is conditional: if admonish is absent, the `before` field is not added — keeping the registered listings entry minimal for books that don't use admonish.

Three new unit tests cover the synthetic-config cases (admonish present, admonish absent, idempotent re-run with admonish present); a new integration test `install_orders_before_admonish_when_admonish_is_registered` sets up a fixture book with `[preprocessor.admonish]`, runs install, and asserts the resulting `book.toml` has the `before = ["admonish"]` ordering plus both preprocessor entries.

What's new in `install-v7` compared to `install-v6`: the `register_listings_preprocessor` method now snapshots whether `"admonish"` is a key in the preprocessor table before adding listings, then appends `before = ["admonish"]` to the listings entry if so. Three new tests in the unit-test module. The method's existing behaviour (preprocessor entry with command) and idempotency are unchanged.

```
//! `install` subcommand: configures an existing book to use mdbook-listings.

use std::fs;
use std::path::Path;

use anyhow::{Context, Result};
use toml_edit::{Array, DocumentMut, Item, Table, Value};

/// Compiled in so `cargo install mdbook-listings` produces a self-contained
/// binary with nothing external to fetch at install time.
pub const CSS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.css");

/// Catches builds that stripped or replaced the asset — a missing sentinel
/// means the bundled bytes are not the expected build-time asset.
pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v1";

/// Shared between [`write_css_asset`] and
/// [`BookConfig::register_listings_css`] so the two can't drift.
pub const CSS_ASSET_FILENAME: &str = "mdbook-listings.css";

/// Always overwrites — install ships the bundled bytes, not whatever a
/// stale on-disk copy happens to contain.
pub fn write_css_asset(book_root: &Path) -> Result<()> {
    let path = book_root.join(CSS_ASSET_FILENAME);
    fs::write(&path, CSS_ASSET).with_context(|| format!("writing CSS asset to
{}", path.display()))
}

/// Idempotent: book.toml and the CSS asset on disk are only rewritten if
/// they differ from what install would produce.
```

```
pub fn install(book_root: &Path) -> Result<InstallOutcome> {
    let book_toml_path = book_root.join("book.toml");
    let original = match fs::read_to_string(&book_toml_path) {
        Ok(s) => s,
        Err(e) if e.kind() == std::io::ErrorKind::NotFound => {
            anyhow::bail!(
                "book.toml not found at {} — install requires an existing mdbook
book directory; run `mdbook init` first.",
                book_toml_path.display(),
            );
        }
        Err(e) => {
            return Err(anyhow::Error::from(e))
                .with_context(|| format!("reading book config at {}",
book_toml_path.display()));
        }
    };
    let mut config = BookConfig::parse(&original)?;
    config.register_listings_preprocessor();
    config.register_listings_css();
    let new = config.render();

    let css_path = book_root.join(CSS_ASSET_FILENAME);
    let css_already_correct = fs::read(&css_path)
        .ok()
        .is_some_and(|bytes| bytes.as_slice() == CSS_ASSET);

    let toml_changed = new != original;
    if toml_changed {
        fs::write(&book_toml_path, new)
            .with_context(|| format!("writing book config at {}",
book_toml_path.display()))?;
    }
    if !css_already_correct {
        write_css_asset(book_root)?;
    }

    Ok(if toml_changed || !css_already_correct {
        InstallOutcome::Installed
    } else {
        InstallOutcome::Unchanged
    })
}

/// Lets the CLI tell the author whether a re-install was a no-op (AC 3).
#[derive(Debug, PartialEq, Eq)]
pub enum InstallOutcome {
    Installed,
    Unchanged,
}

/// Newtype over [`toml_edit::DocumentMut`] so callers don't depend on
/// `toml_edit` directly and the `register_*` methods have a domain type
/// to attach to.
#[derive(Debug)]
```

```
pub struct BookConfig(DocumentMut);

impl BookConfig {
    pub fn parse(s: &str) -> Result<Self> {
        s.parse::<DocumentMut>()
            .map(BookConfig)
            .context("book config is not valid TOML")
    }

    pub fn render(&self) -> String {
        self.0.to_string()
    }

    /// Idempotent: a second call on an already-registered config is a no-op
    /// in the rendered output. If `[preprocessor.admonish]` is registered,
    /// the listings entry gets `before = ["admonish"]` so the
    /// callout → admonish-note pipeline produces correctly styled PDF
    /// output.
    pub fn register_listings_preprocessor(&mut self) {
        let preprocessor = self
            .0
            .as_table_mut()
            .entry("preprocessor")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor] must be a table");
        let admonish_present = preprocessor.contains_key("admonish");
        let listings = preprocessor
            .entry("listings")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[preprocessor.listings] must be a table");
        listings["command"] = toml_edit::value("mdbook-listings");
        if admonish_present {
            let mut before = Array::new();
            before.push("admonish");
            listings["before"] = toml_edit::value(before);
        }
    }

    /// Idempotent: duplicate entries are not appended.
    pub fn register_listings_css(&mut self) {
        let entry = format!("./{CSS_ASSET_FILENAME}");
        let array = self
            .0
            .as_table_mut()
            .entry("output")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[output] must be a table")
            .entry("html")
            .or_insert_with(|| Item::Table(Table::new()))
            .as_table_mut()
            .expect("[output.html] must be a table")
            .entry("additional-css")
```

```
            .or_insert_with(|| Item::Value(Value::Array(Array::new())))
            .as_value_mut()
            .expect("additional-css must be a value")
            .as_array_mut()
            .expect("additional-css must be an array");
        if !array.iter().any(|v| v.as_str() == Some(entry.as_str())) {
            array.push(entry);
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn css_asset_is_non_empty() {
        assert!(!CSS_ASSET.is_empty(), "bundled CSS asset must not be empty");
    }

    #[test]
    fn css_asset_contains_sentinel() {
        let contents = std::str::from_utf8(CSS_ASSET).expect("CSS asset must be
UTF-8");
        assert!(
            contents.contains(CSS_ASSET_SENTINEL),
            "bundled CSS asset must contain sentinel `{CSS_ASSET_SENTINEL}`;
got:\n{contents}",
        );
    }

    #[test]
    fn book_config_round_trip_preserves_comments_and_ordering() {
        let input = "\
# top comment
[book]
title = \"Test\"

# preprocessor comment
[preprocessor.admonish]
command = \"mdbook-admonish\"

[output.html]
";
        let cfg = BookConfig::parse(input).expect("parse");
        assert_eq!(cfg.render(), input);
    }

    #[test]
    fn book_config_parse_rejects_invalid_toml() {
        let err = BookConfig::parse("[book\nbroken = ").unwrap_err();
        let msg = format!("{err:#}");
        assert!(
            msg.contains("not valid TOML"),
            "diagnostic should name the failure mode; got: {msg}"
```

```
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[preprocessor.listings]"),
            "rendered config should declare [preprocessor.listings]; got:
\n{rendered}",
        );
        assert!(
            rendered.contains(r#"command = "mdbook-listings""#),
            "rendered config should set command = \"mdbook-listings\"; got:
\n{rendered}",
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_preprocessor();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_preprocessor();
        let after_second = cfg2.render();

        assert_eq!(after_first, after_second, "register must be idempotent");
    }

    #[test]
    fn book_config_register_listings_css_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_css();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[output.html]"),
            "rendered config should declare [output.html]; got:\n{rendered}",
        );
        assert!(
            rendered.contains(r#"additional-css = ["./mdbook-listings.css"]"#),
            "rendered config should reference the CSS asset; got:\n{rendered}",
        );
    }

    #[test]
    fn
book_config_register_listings_preprocessor_orders_before_admonish_when_present()
{
        let input = "[preprocessor.admonish]\ncommand = \"mdbook-admonish\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
```

```
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            rendered.contains(r#"before = ["admonish"]"#),
            "listings should declare before = [\"admonish\"]; got:\n{rendered}",
        );
        assert!(
            rendered.contains("[preprocessor.admonish]"),
            "admonish should still be registered; got:\n{rendered}",
        );
    }

    #[test]
    fn
book_config_register_listings_preprocessor_skips_before_when_admonish_absent() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            !rendered.contains("before"),
            "listings should not declare a before field when admonish is absent;
got:\n{rendered}",
        );
    }

    #[test]
    fn
book_config_register_listings_preprocessor_idempotent_with_admonish_present() {
        let input = "[preprocessor.admonish]\ncommand = \"mdbook-admonish\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_preprocessor();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_preprocessor();
        let after_second = cfg2.render();

        assert_eq!(
            after_first, after_second,
            "register must be idempotent when admonish is present"
        );
    }

    #[test]
    fn book_config_register_listings_css_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_css();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_css();
        let after_second = cfg2.render();

        assert_eq!(
```

```
            after_first, after_second,
            "register_listings_css must be idempotent"
        );
    }
}
```

What's new in `install-tests-v4` compared to `install-tests-v3`: the new `install_orders_before_admonish_when_admonish_is_registered` integration test. The other tests and helper struct are unchanged.

```
//! Integration tests for the Install the Preprocessor story (ch. 1).

use std::fs;
use std::path::{Path, PathBuf};

use predicates::str::contains;
use tempfile::TempDir;

mod common;
use common::mdbook_listings;

#[test]
fn install_registers_preprocessor_and_writes_css() {
    let book = MinimalFixtureBook::new();

    mdbook_listings()
        .args(["install", "--book-root"])
        .arg(book.root())
        .assert()
        .success();

    book.assert_preprocessor_registered();
    book.assert_css_asset_present();
}

/// The smallest mdbook that's still a valid book: a `book.toml` declaring
/// just the `[book]` table with a title, materialised in a TempDir whose
/// lifetime is tied to this struct so the filesystem clean-up is automatic.
struct MinimalFixtureBook {
    _tmp: TempDir,
    root: PathBuf,
}

impl MinimalFixtureBook {
    fn new() -> Self {
        let tmp = TempDir::new().expect("tempdir");
        let root = tmp.path().to_path_buf();
        fs::write(root.join("book.toml"), "[book]\ntitle =
\"Test\"\n").unwrap();
        Self { _tmp: tmp, root }
    }

    fn root(&self) -> &Path {
        &self.root
```

```
    }

    fn assert_preprocessor_registered(&self) {
        let book_toml =
fs::read_to_string(self.root.join("book.toml")).unwrap();
        assert!(
            book_toml.contains("[preprocessor.listings]"),
            "book.toml should register the preprocessor; got:\n{book_toml}",
        );
        assert!(
            book_toml.contains("mdbook-listings.css"),
            "book.toml should reference the CSS asset; got:\n{book_toml}",
        );
    }

    fn assert_css_asset_present(&self) {
        assert!(
            self.root.join("mdbook-listings.css").exists(),
            "CSS asset should be written to the book root",
        );
    }
}

/// `install` against a book that already has `[preprocessor.admonish]`
/// registers listings with `before = ["admonish"]` so the preprocessor
/// chain runs in the right order for PDF output (AC 6).
#[test]
fn install_orders_before_admonish_when_admonish_is_registered() {
    let tmp = TempDir::new().expect("tempdir");
    let book_root = tmp.path();
    fs::write(
        book_root.join("book.toml"),
        "[book]\ntitle = \"Test\"\n\n[preprocessor.admonish]\ncommand =
\"mdbook-admonish\"\n",
    )
    .unwrap();

    mdbook_listings()
        .args(["install", "--book-root"])
        .arg(book_root)
        .assert()
        .success();

    let book_toml = fs::read_to_string(book_root.join("book.toml")).unwrap();
    assert!(
        book_toml.contains(r#"before = ["admonish"]"#),
        "listings should be ordered before admonish; got:\n{book_toml}",
    );
    assert!(
        book_toml.contains("[preprocessor.listings]"),
        "listings preprocessor should still be registered; got:\n{book_toml}",
    );
    assert!(
        book_toml.contains("[preprocessor.admonish]"),
        "admonish should still be registered (untouched); got:\n{book_toml}",
```

```
    );
}

/// `install` against a directory with no `book.toml` exits non-zero with a
/// diagnostic identifying what was expected (AC 5).
#[test]
fn install_rejects_missing_book_config() {
    let tmp = TempDir::new().expect("tempdir");

    mdbook_listings()
        .args(["install", "--book-root"])
        .arg(tmp.path())
        .assert()
        .failure()
        .stderr(contains("book.toml not found"));
}
```

The suite now runs 27 tests (14 install-related, 13 from other modules). Every Acceptance criterion has at least one test covering it. The story is feature-complete; the optional refactor slice that follows tidies a small repetition that accumulated across slices 4–8, and the wrap-up chore after that promotes the remaining HTML-comment scaffold to chapter body.

Refactor

With every test green, the refactor commit tidies a small repetition that accumulated during slices 4–8 and was deliberately not addressed during the red-green slices: `register_listings_preprocessor` and `register_listings_css` both walked into nested `[preprocessor]` and `[output.html]` tables via the same six-line `entry().or_insert_with(...).as_table_mut().expect(...)` chain. Extracted into a `subtable_mut(parent, key)` helper so each call site shrinks from six lines to one.

The chapter has this section on purpose — the methodology in ch. 0 calls out the refactor commit as part of the outside-in cycle, and shipping a tiny one here makes that visible. Larger refactors are still possible later (e.g., splitting install.rs once it grows substantially); none felt load-bearing right now.

What's new in `install-v8` compared to `install-v7`: the private `subtable_mut` helper at module scope, and both `register_*` methods rewritten to use it. No public API change; no behaviour change. The full test suite (14 install tests, 27 overall) passes byte-for-byte the same as before.

```
//! `install` subcommand: configures an existing book to use mdbook-listings.

use std::fs;
use std::path::Path;

use anyhow::{Context, Result};
use toml_edit::{Array, DocumentMut, Item, Table, Value};

/// Compiled in so `cargo install mdbook-listings` produces a self-contained
/// binary with nothing external to fetch at install time.
pub const CSS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.css");
```

```
/// Catches builds that stripped or replaced the asset — a missing sentinel
/// means the bundled bytes are not the expected build-time asset.
pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v1";

/// Shared between [`write_css_asset`] and
/// [`BookConfig::register_listings_css`] so the two can't drift.
pub const CSS_ASSET_FILENAME: &str = "mdbook-listings.css";

/// Always overwrites — install ships the bundled bytes, not whatever a
/// stale on-disk copy happens to contain.
pub fn write_css_asset(book_root: &Path) -> Result<()> {
    let path = book_root.join(CSS_ASSET_FILENAME);
    fs::write(&path, CSS_ASSET).with_context(|| format!("writing CSS asset to
{}", path.display()))
}

/// Idempotent: book.toml and the CSS asset on disk are only rewritten if
/// they differ from what install would produce.
pub fn install(book_root: &Path) -> Result<InstallOutcome> {
    let book_toml_path = book_root.join("book.toml");
    let original = match fs::read_to_string(&book_toml_path) {
        Ok(s) => s,
        Err(e) if e.kind() == std::io::ErrorKind::NotFound => {
            anyhow::bail!(
                "book.toml not found at {} — install requires an existing mdbook
book directory; run `mdbook init` first.",
                book_toml_path.display(),
            );
        }
        Err(e) => {
            return Err(anyhow::Error::from(e))
                .with_context(|| format!("reading book config at {}",
book_toml_path.display()));
        }
    };
    let mut config = BookConfig::parse(&original)?;
    config.register_listings_preprocessor();
    config.register_listings_css();
    let new = config.render();

    let css_path = book_root.join(CSS_ASSET_FILENAME);
    let css_already_correct = fs::read(&css_path)
        .ok()
        .is_some_and(|bytes| bytes.as_slice() == CSS_ASSET);

    let toml_changed = new != original;
    if toml_changed {
        fs::write(&book_toml_path, new)
            .with_context(|| format!("writing book config at {}",
book_toml_path.display()))?;
    }
    if !css_already_correct {
        write_css_asset(book_root)?;
    }
```

```
    Ok(if toml_changed || !css_already_correct {
        InstallOutcome::Installed
    } else {
        InstallOutcome::Unchanged
    })
}

/// Lets the CLI tell the author whether a re-install was a no-op (AC 3).
#[derive(Debug, PartialEq, Eq)]
pub enum InstallOutcome {
    Installed,
    Unchanged,
}

/// Newtype over [`toml_edit::DocumentMut`] so callers don't depend on
/// `toml_edit` directly and the `register_*` methods have a domain type
/// to attach to.
#[derive(Debug)]
pub struct BookConfig(DocumentMut);

impl BookConfig {
    pub fn parse(s: &str) -> Result<Self> {
        s.parse::<DocumentMut>()
            .map(BookConfig)
            .context("book config is not valid TOML")
    }

    pub fn render(&self) -> String {
        self.0.to_string()
    }

    /// Idempotent: a second call on an already-registered config is a no-op
    /// in the rendered output. If `[preprocessor.admonish]` is registered,
    /// the listings entry gets `before = ["admonish"]` so the
    /// callout → admonish-note pipeline produces correctly styled PDF
    /// output.
    pub fn register_listings_preprocessor(&mut self) {
        let preprocessor = subtable_mut(self.0.as_table_mut(), "preprocessor");
        let admonish_present = preprocessor.contains_key("admonish");
        let listings = subtable_mut(preprocessor, "listings");
        listings["command"] = toml_edit::value("mdbook-listings");
        if admonish_present {
            let mut before = Array::new();
            before.push("admonish");
            listings["before"] = toml_edit::value(before);
        }
    }

    /// Idempotent: duplicate entries are not appended.
    pub fn register_listings_css(&mut self) {
        let entry = format!("./{CSS_ASSET_FILENAME}");
        let html = subtable_mut(subtable_mut(self.0.as_table_mut(), "output"),
"html");
        let array = html
            .entry("additional-css")
```

```
            .or_insert_with(|| Item::Value(Value::Array(Array::new())))
            .as_value_mut()
            .expect("additional-css must be a value")
            .as_array_mut()
            .expect("additional-css must be an array");
        if !array.iter().any(|v| v.as_str() == Some(entry.as_str())) {
            array.push(entry);
        }
    }
}

/// Get a mutable reference to `parent[key]` as a `Table`, creating the
/// child table if absent. Replaces the open-coded
/// `entry().or_insert_with(...).as_table_mut().expect(...)` chain.
fn subtable_mut<'a>(parent: &'a mut Table, key: &str) -> &'a mut Table {
    parent
        .entry(key)
        .or_insert_with(|| Item::Table(Table::new()))
        .as_table_mut()
        .expect("entry must be a table")
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn css_asset_is_non_empty() {
        assert!(!CSS_ASSET.is_empty(), "bundled CSS asset must not be empty");
    }

    #[test]
    fn css_asset_contains_sentinel() {
        let contents = std::str::from_utf8(CSS_ASSET).expect("CSS asset must be
UTF-8");
        assert!(
            contents.contains(CSS_ASSET_SENTINEL),
            "bundled CSS asset must contain sentinel `{CSS_ASSET_SENTINEL}`;
got:\n{contents}",
        );
    }

    #[test]
    fn book_config_round_trip_preserves_comments_and_ordering() {
        let input = "\
# top comment
[book]
title = \"Test\"

# preprocessor comment
[preprocessor.admonish]
command = \"mdbook-admonish\"

[output.html]
";
```

```
        let cfg = BookConfig::parse(input).expect("parse");
        assert_eq!(cfg.render(), input);
    }

    #[test]
    fn book_config_parse_rejects_invalid_toml() {
        let err = BookConfig::parse("[book\nbroken = ").unwrap_err();
        let msg = format!("{err:#}");
        assert!(
            msg.contains("not valid TOML"),
            "diagnostic should name the failure mode; got: {msg}"
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[preprocessor.listings]"),
            "rendered config should declare [preprocessor.listings]; got:
\n{rendered}",
        );
        assert!(
            rendered.contains(r#"command = "mdbook-listings""#),
            "rendered config should set command = \"mdbook-listings\"; got:
\n{rendered}",
        );
    }

    #[test]
    fn book_config_register_listings_preprocessor_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_preprocessor();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_preprocessor();
        let after_second = cfg2.render();

        assert_eq!(after_first, after_second, "register must be idempotent");
    }

    #[test]
    fn book_config_register_listings_css_adds_entry() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_css();
        let rendered = cfg.render();
        assert!(
            rendered.contains("[output.html]"),
            "rendered config should declare [output.html]; got:\n{rendered}",
        );
        assert!(
```

```
            rendered.contains(r#"additional-css = ["./mdbook-listings.css"]"#),
            "rendered config should reference the CSS asset; got:\n{rendered}",
        );
    }

    #[test]
    fn
book_config_register_listings_preprocessor_orders_before_admonish_when_present()
{
        let input = "[preprocessor.admonish]\ncommand = \"mdbook-admonish\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            rendered.contains(r#"before = ["admonish"]"#),
            "listings should declare before = [\"admonish\"]; got:\n{rendered}",
        );
        assert!(
            rendered.contains("[preprocessor.admonish]"),
            "admonish should still be registered; got:\n{rendered}",
        );
    }

    #[test]
    fn
book_config_register_listings_preprocessor_skips_before_when_admonish_absent() {
        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
        cfg.register_listings_preprocessor();
        let rendered = cfg.render();
        assert!(
            !rendered.contains("before"),
            "listings should not declare a before field when admonish is absent;
got:\n{rendered}",
        );
    }

    #[test]
    fn
book_config_register_listings_preprocessor_idempotent_with_admonish_present() {
        let input = "[preprocessor.admonish]\ncommand = \"mdbook-admonish\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_preprocessor();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_preprocessor();
        let after_second = cfg2.render();

        assert_eq!(
            after_first, after_second,
            "register must be idempotent when admonish is present"
        );
    }

    #[test]
```

```
    fn book_config_register_listings_css_is_idempotent() {
        let input = "[book]\ntitle = \"Test\"\n";
        let mut cfg = BookConfig::parse(input).unwrap();
        cfg.register_listings_css();
        let after_first = cfg.render();

        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
        cfg2.register_listings_css();
        let after_second = cfg2.render();

        assert_eq!(
            after_first, after_second,
            "register_listings_css must be idempotent"
        );
    }
}
```

The chapter is feature- and quality-complete. The wrap-up chore lifts the remaining scaffold sections out of the HTML comment.

Notes for implementers

- `toml_edit` is the standard crate for read-modify-write of a TOML

file while preserving comments and ordering. mdbook-admonish's own install is a good reference implementation; we studied it while writing the Freeze chapter.

- The CSS file content is a placeholder until the **Render Inline Callouts** story (ch. 4) settles the badge styling. The

`CSS_ASSET_SENTINEL` constant exists so a build that strips or replaces the asset fails loudly rather than silently shipping wrong content.

- **Listing overlap with ch. 2 (Freeze a Listing).** This story

modifies `src/lib.rs` and `src/main.rs` — files frozen by ch. 2 under `-v1` tags — and adds `src/install.rs`. Per the per-slice freeze discipline, each slice that touches one of these files freezes a new `-vN` tag (the latest are catalogued in **Final state** below). Until the **Show Diffs Between Slices** primitive ships, readers of ch. 1 and ch. 2 in sequence see overlapping full-file listings; the duplication goes away as a one-line cleanup once diffs are available.

What this slice does not solve

- No uninstall command. Authors who want to remove mdbook-listings

edit `book.toml` by hand.

- No upgrade flow. When the bundled CSS asset version bumps,

authors re-run install, which overwrites the asset.

- No detection of pre-existing conflicting configurations. If the

book already has a different preprocessor named `listings`, install silently overwrites its `command` value.

Freeze a Listing

> **This chapter is reconstructed retrospectively**
>
> The Freeze a Listing story landed in a single commit before this book adopted outside-in TDD as its development discipline. As a result, this chapter shows the story's **end state** — story, acceptance criteria, final listings, design decisions — without an outside-in narrative walking through slices. Chapters from ch. 2 onward have one narrative section per slice.

Story

> As a book author, I want to freeze a source file into my book under a memorable tag so that a later edit to the source file does not silently change what my chapter renders.

Acceptance criteria

1. When an author freezes a source file under a tag, the file's

bytes are preserved verbatim and become consumable from any chapter by that tag, using mdbook's existing include machinery (no additional wiring required).
1. Each freeze is recorded persistently. The record captures the

chosen tag, the original source path, the frozen location, and an integrity hash of the frozen bytes.
1. Re-freezing the same source under the same tag with no change

to its bytes does not modify disk state and confirms to the author that nothing changed.
1. Re-freezing under a tag that already exists, but with different

bytes — whether the original source was edited or a different source was supplied — is rejected. No disk state changes.
1. The author can opt in to overwriting an existing tag's frozen

copy with new bytes. Doing so updates the record and the frozen content, and confirms the replacement to the author.
1. Tags that could escape the listings area (e.g. containing path

separators or relative-path segments) are rejected before any disk state changes. Acceptance criteria 3, 4, and 5 together are the "idempotency discipline" that keeps this tool honest: the tag is the identity, the bytes are the content, and the author has to be explicit when they want to break the association.

The slice

The slice cuts top-to-bottom through the crate:

File	Role	Frozen tag
`tests/freeze.rs`	Acceptance criteria as CLI-level tests	`freeze-tests-v1`
`src/main.rs`	`clap` subcommand handler — the CLI adapter	`main-v1`

`src/freeze.rs`	Core logic: hash, decide, write, upsert	`freeze-v1`
`src/manifest.rs`	TOML load / save / upsert	`manifest-v1`
`src/lib.rs`	Public module registrations	`lib-v1`

Every file contributing to the slice is frozen as of this commit and embedded below. When later stories edit these files, they freeze a new tag (`freeze-v2`, etc.) and this chapter keeps pointing at `-v1`.

Design decisions

Four decisions shape the behaviour of `freeze`. They are called out explicitly here so that later slices that extend `freeze` know which invariants they are allowed to break and which they are not.

Manifest format: TOML

The manifest is TOML, matching mdbook's own `book.toml` and Cargo's `Cargo.toml`/`Cargo.lock`. The alternatives would have been YAML (more concise for nested structures but breaks ecosystem fit) or JSON (nicer for machines, worse for humans to hand-edit). TOML wins on ecosystem coherence.

Listing identifier: author-chosen tag plus content hash

Listings are named by an author-supplied tag (`manifest-v1`, `freeze-v1`, etc.) — the *human* identifier. The manifest also stores a SHA-256 of the frozen bytes, which is the *integrity* identifier. This is the pattern Git uses for refs and commits: humans remember the name, machines verify the hash.

Pure content-addressed identifiers (name the file by its SHA) were rejected because `{{#include listings/a1b2c3d4.rs}}` is unreadable. Auto-derived identifiers (chapter + section + index) were rejected because inserting or reordering a section silently renumbers everything.

Frozen directory layout: `<book-root>/src/listings/<tag>.<ext>`

Frozen files live under `src/listings/` inside the book so the built-in `{{#include}}` resolver finds them without any path gymnastics. The extension is inherited from the source so syntax highlighting works automatically.

Freeze trigger: manual CLI only (for now)

`mdbook-listings freeze` is the only way a listing gets frozen. Pre-commit hooks and tag-triggered freezes are on the backlog but deferred — they introduce policy questions (whose tag? which commit?) that are not worth answering until later stories reveal which of those policies authors actually want.

Final state

The four source files and the integration test, as of this commit, all frozen.

`tests/freeze.rs` — the acceptance criteria as tests

```
//! Integration tests for the Freeze a Listing story (ch. 2). These pin the
//! error-path acceptance criteria that aren't covered by the book's own use
//! of the freeze primitive on its own listings.
```

```
use std::fs;

use predicates::str::contains;
use tempfile::TempDir;

mod common;
use common::mdbook_listings;

/// `freeze` rejects re-running with the same tag but a now-different source
/// content unless `--force` is given. Without this guard, an author who edits
/// a source file and re-runs `freeze` would silently lose the previously
/// frozen bytes.
#[test]
fn freeze_rejects_conflicting_content_without_force() {
    let tmp = TempDir::new().expect("tempdir");
    let book_root = tmp.path().join("book");
    fs::create_dir_all(&book_root).unwrap();
    let source = tmp.path().join("compose.yaml");
    fs::write(&source, "a: 1\n").unwrap();

    mdbook_listings()
        .args(["freeze", "--tag", "t", "--book-root"])
        .arg(&book_root)
        .arg(&source)
        .assert()
        .success();

    fs::write(&source, "a: 2\n").unwrap();
    mdbook_listings()
        .args(["freeze", "--tag", "t", "--book-root"])
        .arg(&book_root)
        .arg(&source)
        .assert()
        .failure()
        .stderr(contains("already frozen"));
}

/// `freeze` rejects re-running with the same tag but content from an entirely
/// different source file, unless `--force` is given. The tag is the identity;
/// without this guard, an author who accidentally re-uses a tag for a new
/// source would clobber the previously frozen bytes silently.
#[test]
fn freeze_rejects_duplicate_tag_from_different_source() {
    let tmp = TempDir::new().expect("tempdir");
    let book_root = tmp.path().join("book");
    fs::create_dir_all(&book_root).unwrap();
    let source_a = tmp.path().join("a.yaml");
    let source_b = tmp.path().join("b.yaml");
    fs::write(&source_a, "a: 1\n").unwrap();
    fs::write(&source_b, "b: 2\n").unwrap();

    mdbook_listings()
        .args(["freeze", "--tag", "t", "--book-root"])
        .arg(&book_root)
```

```
        .arg(&source_a)
        .assert()
        .success();

    mdbook_listings()
        .args(["freeze", "--tag", "t", "--book-root"])
        .arg(&book_root)
        .arg(&source_b)
        .assert()
        .failure()
        .stderr(contains("already frozen"));
}
```

The `freeze_rejects_conflicting_content_without_force` and `freeze_rejects_duplicate_tag_from_different_source` tests together pin the two halves of **AC 4**, which is the one criterion the book itself can't exercise (because the book only drives the happy paths).

`src/main.rs` — the CLI adapter

```
use std::path::PathBuf;
use std::process;

use anyhow::Result;
use clap::{Parser, Subcommand};
use mdbook_listings::freeze::{FreezeOptions, FreezeOutcome, freeze};

/// Managed code listings for mdbook: inline callouts, freezing, and
verification.
#[derive(Parser)]
#[command(version, about, long_about = None)]
struct Cli {
    #[command(subcommand)]
    command: Option<Command>,
}

#[derive(Subcommand)]
enum Command {
    /// Check whether a renderer is supported by this preprocessor.
    ///
    /// Invoked by mdbook during the build to decide whether to pipe the book
    /// through this preprocessor for a given renderer. Exits 0 if supported,
    /// 1 otherwise.
    Supports {
        /// Name of the renderer mdbook is asking about (e.g. `html`, `typst-
pdf`).
        renderer: String,
    },

    /// Install preprocessor assets and register mdbook-listings in `book.toml`.
    Install {
        /// Root directory of the book (contains `book.toml`). Defaults to the
        /// current directory.
        #[arg(long)]
```

```
        book_root: Option<PathBuf>,
    },

    /// Freeze a source file into the book's listings directory and update
    /// the manifest.
    Freeze {
        /// Human-readable tag used as the frozen filename and as the manifest
        /// entry key. Should be unique within the book.
        #[arg(long)]
        tag: String,

        /// Root directory of the book. Defaults to the current directory.
        #[arg(long)]
        book_root: Option<PathBuf>,

        /// Overwrite an existing frozen copy with the same tag.
        #[arg(long)]
        force: bool,

        /// Path to the source file to freeze.
        source: PathBuf,
    },

    /// Verify consistency between the manifest, frozen listings, and
`{{#include}}`
    /// references in the book's markdown.
    Verify {
        /// Root directory of the book. Defaults to the current directory.
        #[arg(long)]
        book_root: Option<PathBuf>,
    },
}

fn main() {
    if let Err(err) = run() {
        eprintln!("error: {err:?}");
        process::exit(1);
    }
}

fn run() -> Result<()> {
    let cli = Cli::parse();
    match cli.command {
        None => preprocess(),
        Some(Command::Supports { renderer }) => supports(&renderer),
        Some(Command::Install { book_root: _ }) => {
            anyhow::bail!("`mdbook-listings install` is not yet implemented")
        }
        Some(Command::Freeze {
            tag,
            book_root,
            force,
            source,
        }) => {
            let book_root = book_root.unwrap_or_else(|| PathBuf::from("."));
```

```
            let outcome = freeze(FreezeOptions {
                book_root: &book_root,
                tag: &tag,
                source: &source,
                force,
            })?;
            let verb = match outcome {
                FreezeOutcome::Created => "created",
                FreezeOutcome::Unchanged => "unchanged",
                FreezeOutcome::Replaced => "replaced",
            };
            println!("{verb}: {tag}");
            Ok(())
        }
        Some(Command::Verify { book_root: _ }) => {
            anyhow::bail!("`mdbook-listings verify` is not yet implemented")
        }
    }
}

/// Default mode: read an mdbook preprocessor JSON payload from stdin, emit the
/// transformed payload on stdout.
fn preprocess() -> Result<()> {
    anyhow::bail!("mdbook-listings preprocessor mode is not yet implemented")
}

/// Answer mdbook's renderer-support probe by exiting 0 (supported) or 1
/// (unsupported). We do not return from this function.
fn supports(renderer: &str) -> ! {
    let supported = matches!(renderer, "html" | "typst-pdf");
    process::exit(if supported { 0 } else { 1 });
}
```

The no-subcommand arm (`preprocess()`) is a stub that errors — the preprocessor pipeline belongs to the **Show Diffs Between Slices** story and isn't implemented yet. Likewise `install` and `verify` are stubs that will fill in later. The `supports` arm is real and was shipped by the CLI-scaffolding chore, not this story; it's here because the dispatch table has to mention every subcommand.

`src/freeze.rs` — the freeze logic

```
//! `mdbook-listings freeze`: snapshot a source file into the book-local
//! listings directory and record it in the manifest.

use std::fs;
use std::path::{Path, PathBuf};

use anyhow::{Context, Result, anyhow, bail};
use sha2::{Digest, Sha256};

use crate::manifest::{Listing, Manifest};

/// Relative path from a book root to the frozen-listings directory.
pub const LISTINGS_SUBDIR: &str = "src/listings";
```

```
/// Options accepted by [`freeze`]. Mirrors the CLI flags 1:1 so the binary
/// layer stays a thin adapter.
#[derive(Debug)]
pub struct FreezeOptions<'a> {
    pub book_root: &'a Path,
    pub tag: &'a str,
    pub source: &'a Path,
    pub force: bool,
}

/// Outcome of a freeze invocation. Callers use this to render a summary line
/// and to drive tests.
#[derive(Debug, PartialEq, Eq)]
pub enum FreezeOutcome {
    /// New listing with this tag; frozen file and manifest entry created.
    Created,
    /// Tag already existed and the source content is byte-identical to the
    /// frozen copy on disk; nothing was changed.
    Unchanged,
    /// Tag already existed and the source differs; frozen file and manifest
    /// entry were overwritten (only possible with `--force`).
    Replaced,
}

/// Freeze `opts.source` into `<book_root>/src/listings/<tag>.<ext>` and upsert
/// the corresponding entry in `<book_root>/listings.toml`.
pub fn freeze(opts: FreezeOptions<'_>) -> Result<FreezeOutcome> {
    let source_bytes = fs::read(opts.source)
        .with_context(|| format!("reading source file {}",
opts.source.display()))?;
    let source_sha = hex_sha256(&source_bytes);

    let frozen_rel = frozen_relative_path(opts.tag, opts.source)?;
    let frozen_abs = opts.book_root.join(&frozen_rel);

    let mut manifest = Manifest::load(opts.book_root)?;

    let outcome = match manifest.find(opts.tag) {
        Some(existing) if existing.sha256 == source_sha && frozen_abs.exists()
=> {
            FreezeOutcome::Unchanged
        }
        Some(_existing) if !opts.force => {
            bail!(
                "tag `{}` already frozen with different content; re-run with --
force to overwrite",
                opts.tag
            )
        }
        Some(_existing) => FreezeOutcome::Replaced,
        None => FreezeOutcome::Created,
    };

    if outcome != FreezeOutcome::Unchanged {
```

```
        if let Some(parent) = frozen_abs.parent() {
            fs::create_dir_all(parent).with_context(|| {
                format!("creating frozen-listings directory {}",
parent.display())
            })?;
        }
        fs::write(&frozen_abs, &source_bytes)
            .with_context(|| format!("writing frozen file {}",
frozen_abs.display()))?;

        let source_rel = relativize(opts.source, opts.book_root);
        manifest.upsert(Listing {
            tag: opts.tag.to_string(),
            source: path_to_string(&source_rel)?,
            frozen: path_to_string(&frozen_rel)?,
            sha256: source_sha,
        });
        manifest.save(opts.book_root)?;
    }

    Ok(outcome)
}

fn frozen_relative_path(tag: &str, source: &Path) -> Result<PathBuf> {
    if tag.is_empty() {
        bail!("tag must be non-empty");
    }
    if tag.contains(['/', '\\', '.']) {
        bail!("tag `{tag}` contains disallowed character (/, \\, or .)");
    }
    let ext = source
        .extension()
        .and_then(|s| s.to_str())
        .ok_or_else(|| anyhow!("source {} has no file extension",
source.display()))?;
    Ok(Path::new(LISTINGS_SUBDIR).join(format!("{tag}.{ext}")))
}

fn relativize(path: &Path, base: &Path) -> PathBuf {
    pathdiff(path, base).unwrap_or_else(|| path.to_path_buf())
}

/// Minimal relative-path computation: if `path` is under `base`, strip the
/// prefix; otherwise walk up from `base` with `..` segments.
fn pathdiff(path: &Path, base: &Path) -> Option<PathBuf> {
    let path = path.canonicalize().ok()?;
    let base = base.canonicalize().ok()?;
    let mut path_components: Vec<_> = path.components().collect();
    let mut base_components: Vec<_> = base.components().collect();
    while let (Some(p), Some(b)) = (path_components.first(),
base_components.first()) {
        if p == b {
            path_components.remove(0);
            base_components.remove(0);
        } else {
```

```
            break;
        }
    }
    let mut result = PathBuf::new();
    for _ in &base_components {
        result.push("..");
    }
    for c in &path_components {
        result.push(c.as_os_str());
    }
    Some(result)
}

fn path_to_string(path: &Path) -> Result<String> {
    path.to_str()
        .map(|s| s.replace('\\', "/"))
        .ok_or_else(|| anyhow!("path {} is not valid UTF-8", path.display()))
}

fn hex_sha256(bytes: &[u8]) -> String {
    let mut hasher = Sha256::new();
    hasher.update(bytes);
    let digest = hasher.finalize();
    let mut out = String::with_capacity(digest.len() * 2);
    for byte in digest {
        use std::fmt::Write;
        let _ = write!(out, "{byte:02x}");
    }
    out
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn hex_sha256_matches_known_vector() {
        // Well-known sha256("") per FIPS 180-4.
        assert_eq!(
            hex_sha256(b""),
            "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
        );
    }

    #[test]
    fn frozen_path_composes_tag_and_extension() {
        let p = frozen_relative_path("compose-v1",
Path::new("compose.yaml")).unwrap();
        assert_eq!(p, Path::new("src/listings/compose-v1.yaml"));
    }

    #[test]
    fn frozen_path_rejects_tag_with_slash() {
        assert!(frozen_relative_path("foo/bar", Path::new("x.yaml")).is_err());
    }
```

```
    #[test]
    fn frozen_path_rejects_empty_tag() {
        assert!(frozen_relative_path("", Path::new("x.yaml")).is_err());
    }

    #[test]
    fn frozen_path_rejects_extensionless_source() {
        assert!(frozen_relative_path("tag", Path::new("Makefile")).is_err());
    }
}
```

`FreezeOutcome` carries the Create / Unchanged / Replaced decision up to `main.rs` purely so the CLI can print a different verb for each case. Everything else — the four-way match on `manifest.find(tag)`, the sha256 comparison, the early return on `Unchanged` to avoid spurious disk writes — falls directly out of the acceptance criteria.

`frozen_relative_path` is the guard that rejects tags containing `/`, `\`, or `.` (**AC 6**). Rejecting these early, before any disk writes, matters: a tag like `../escape` would otherwise write outside the listings directory.

`src/manifest.rs` — the persistence layer

```
//! Freeze manifest: the TOML file that records every listing that has been
//! frozen into a book. Lives at `<book_root>/listings.toml`.

use std::fs;
use std::path::{Path, PathBuf};

use anyhow::{Context, Result, anyhow};
use serde::{Deserialize, Serialize};

/// Current manifest schema version. Bumped when the on-disk layout changes in
/// a way that requires a migration.
pub const MANIFEST_VERSION: u32 = 1;

/// Relative path from a book root to the manifest file.
pub const MANIFEST_FILENAME: &str = "listings.toml";

/// Top-level manifest document.
#[derive(Debug, Default, Serialize, Deserialize)]
pub struct Manifest {
    pub version: u32,
    #[serde(default, rename = "listing")]
    pub listings: Vec<Listing>,
}

/// One entry per frozen listing.
#[derive(Debug, Clone, Serialize, Deserialize, PartialEq, Eq)]
pub struct Listing {
    /// Human-readable identifier chosen by the author. Unique within the
    /// manifest.
    pub tag: String,
    /// Original source path, relative to the book root. Informational — shallow
```

```
    /// verify does not re-read this file (deep verify, deferred to a later
    /// release, will).
    pub source: String,
    /// Path to the frozen copy, relative to the book root (e.g.
    /// `src/listings/compose-v1.yaml`).
    pub frozen: String,
    /// Hex-encoded sha256 of the frozen file's byte content. Used by shallow
    /// verify to detect post-freeze tampering.
    pub sha256: String,
}

impl Manifest {
    /// Load the manifest from `<book_root>/listings.toml`. Returns an empty
    /// manifest if the file does not exist.
    pub fn load(book_root: &Path) -> Result<Self> {
        let path = Self::path(book_root);
        if !path.exists() {
            return Ok(Self {
                version: MANIFEST_VERSION,
                listings: Vec::new(),
            });
        }
        let text = fs::read_to_string(&path)
            .with_context(|| format!("reading manifest at {}",
path.display()))?;
        let manifest: Manifest = toml::from_str(&text)
            .with_context(|| format!("parsing manifest at {}",
path.display()))?;
        if manifest.version != MANIFEST_VERSION {
            return Err(anyhow!(
                "manifest at {} has version {}, expected {}",
                path.display(),
                manifest.version,
                MANIFEST_VERSION
            ));
        }
        Ok(manifest)
    }

    /// Write the manifest to `<book_root>/listings.toml`, creating parent
    /// directories as needed.
    pub fn save(&self, book_root: &Path) -> Result<()> {
        let path = Self::path(book_root);
        if let Some(parent) = path.parent() {
            fs::create_dir_all(parent)
                .with_context(|| format!("creating manifest parent {}",
parent.display()))?;
        }
        let text = toml::to_string_pretty(self).context("serializing manifest to
TOML")?;
        fs::write(&path, text)
            .with_context(|| format!("writing manifest to {}",
path.display()))?;
        Ok(())
    }
```

```
    /// Look up a listing by tag.
    pub fn find(&self, tag: &str) -> Option<&Listing> {
        self.listings.iter().find(|l| l.tag == tag)
    }

    /// Insert or replace a listing by tag, keeping the vector in insertion
    /// order (existing entries retain their position).
    pub fn upsert(&mut self, listing: Listing) {
        match self.listings.iter().position(|l| l.tag == listing.tag) {
            Some(idx) => self.listings[idx] = listing,
            None => self.listings.push(listing),
        }
    }

    fn path(book_root: &Path) -> PathBuf {
        book_root.join(MANIFEST_FILENAME)
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use tempfile::TempDir;

    #[test]
    fn load_rejects_unknown_manifest_version() {
        let tmp = TempDir::new().unwrap();
        let manifest_path = tmp.path().join(MANIFEST_FILENAME);
        fs::write(
            &manifest_path,
            "version = 99\n\n[[listing]]\n\
             tag = \"x\"\nsource = \"a\"\nfrozen = \"b\"\nsha256 = \"c\"\n",
        )
        .unwrap();

        let err = Manifest::load(tmp.path()).unwrap_err();
        let msg = format!("{err}");
        assert!(
            msg.contains("version 99") &&
msg.contains(&MANIFEST_VERSION.to_string()),
            "diagnostic should name both the found and the expected version,
got: {msg}",
        );
    }

    #[test]
    fn load_accepts_current_manifest_version() {
        let tmp = TempDir::new().unwrap();
        let manifest_path = tmp.path().join(MANIFEST_FILENAME);
        fs::write(&manifest_path, format!("version =
{MANIFEST_VERSION}\n")).unwrap();

        let m = Manifest::load(tmp.path()).expect("current-version manifest
should load");
```

```
        assert_eq!(m.version, MANIFEST_VERSION);
        assert!(m.listings.is_empty());
    }
}
```

`upsert` preserves insertion order when replacing an existing entry. That's invisible in the CLI today but matters for reading the `listings.toml` diff in code review: a re-freeze of an existing tag should show up as a sha change on one entry, not as a reorder of the whole file.

`src/lib.rs` — public module registrations

```
//! Managed code listings for mdbook.

pub mod freeze;
pub mod manifest;
```

Nothing to see here; `lib.rs` exists only so `src/main.rs` and the integration tests can reach into the crate's modules as `mdbook_listings::freeze::…`. Every future story will add one more `pub mod` line.

What this slice does not solve

- **No drift detection between source and frozen copy.** If any of

the files above is edited but `mdbook-listings freeze` is not re-run, the book keeps showing the old bytes. That's the *feature* of freezing, but it also means nothing in the tool warns you that your latest refactor didn't make it into the book. This gap is closed by the **Verify Sync with Source** story.

- **No way to show evolution within a chapter.** This chapter's

final-state section prints every file in full. When later chapters are built outside-in with many slices, doing the same would make them unreadable. The **Show Diffs Between Slices** story closes that gap.

- **No inline annotations or cross-references.** The five frozen

listings above are bare code blocks with no way to attach prose to a specific line. The **Render Inline Callouts** story addresses this, with YAML first.

- **`--tag` is required.** Running `mdbook-listings freeze src/manifest.rs` (no tag) fails today. Auto-derivation of the

tag from the source filename plus the next available `-v<N>` suffix is a planned ergonomic enhancement and will land as a small follow-up commit — not as a new story, because the behaviour it adds is narrow enough to describe in a single AC amendment to this chapter when it ships.

- **No automatic installation here.** Registering the preprocessor

in `book.toml`, shipping the CSS asset, and adding the additional-css entry are the responsibility of ch. 1 (*Install the Preprocessor*). Freeze itself doesn't need any of that — it's a standalone subcommand that operates on a book directory; the preprocessor only matters once you want diffs or callouts.

- **The `-v1` listings here are superseded by ch. 1's freezes.**

Ch. 1 (*Install the Preprocessor*) shipped *after* this chapter in implementation order and modified `src/lib.rs`, `src/main.rs`, and added `src/install.rs`. The current state of those files is

captured by ch. 1's outside-in narrative — `lib-v2` in slice 2, `main-v2` in slice 6, and `install-v8` in the Refactor sub-section. The original consolidated `tests/integration.rs` was split into per-story files in ch. 1's first slice; what was here as `integration-tests-v1` is now `freeze-tests-v1`. This chapter keeps pointing at the freeze-story end-of-story snapshots so it shows the code as it actually was when freeze shipped, not the post-install state. Until the diff primitive ships, readers reading both chapters in sequence see overlapping full-file listings.

Show Diffs Between Slices

> **This chapter has shipped**
>
> The story shipped across six outside-in slices, a refactor slice, a follow-on red-green-refactor loop (slice 8) that came out of dogfooding the primitive on this very chapter, and a wrap-up chore. Every Acceptance criterion is exercised by at least one test in the suite. Read the chapter top-to-bottom for the methodology view; the **Outside-in narrative** sub-sections embed each frozen tag at the slice that introduced it, so the latest version of each file is in the slice that touched it last (slice 8 for `src/diff.rs`, `src/main.rs`, and `tests/diffs.rs`).

Story

> As a book author, I want to render a unified diff between two frozen listings of the same file in a chapter, so that I can walk the reader through slice-by-slice evolution without repeating the full file contents on every slice.

Acceptance criteria

1. An author can request a diff between two frozen listings (by tag)

inline in a chapter. The directive renders to a fenced diff block at the point of request.
1. Diff bytes are computed from the *frozen* listings on disk, not

from any current source file. Once a chapter is built, later edits to the original sources do not retroactively change the rendered diff.
1. A diff request that names a tag not present in `listings.toml`

(or whose frozen file is missing on disk) fails the build with a diagnostic identifying the missing tag, the chapter source path, and the 1-based line number of the offending directive within that chapter — enough for the author to jump straight to the bad directive without grep.
1. A diff between byte-identical listings renders a clear “no

changes” notice rather than an empty diff block.
1. Adding a `{{#diff}}` directive to a chapter does not change any

other content in the chapter (the preprocessor is a precise in-place splice).
1. The chapter that *documents* the directive can show its own

syntax verbatim by putting the literal `{{#diff …}}` inside an inline code span (`` `…` ``) or a fenced code block (` ``` ` / `~~~`) — the preprocessor skips any directive whose start byte falls inside either. Backslash-escape (`\{{#diff …}}`) is *not* a reliable escape mechanism: mdbook’s built-in `links` preprocessor strips the leading \ from any `{{#…}}` pattern before any custom preprocessor runs, so the \ never reaches our splicer in the real pipeline.
1. An author can opt in to a diff against a live source file via

`live:<path>` in either operand. The path is resolved relative to the chapter’s source markdown directory — the same convention mdbook uses for `{{#include}}` — so a chapter at `book/src/foo.md` can write `live:foo.txt` to reach `book/src/foo.txt`, or `live:../../src/lib.rs` to reach the crate’s source. Doing so defeats the freeze stability guarantee for that diff and is flagged later by the *Verify Sync with Source* story (ch. 5).

The slice — outside-in narrative outline

The story ships as eight slices (six initial outside-in slices, a refactor, and a follow-on red-green that came out of dogfooding the primitive) plus a wrap-up chore:

Slice	What it adds
1	Failing integration test asserting AC 1 against a tempdir fixture book. The test pipes a hand-built `(PreprocessorContext, Book)` envelope to the binary's no-subcommand arm and asserts on a ```` ```diff ```` fence in the round-tripped chapter content. The arm becomes a no-op pass-through that round-trips the book unchanged, so the assertion fails — the test is `#[ignore]`'d to keep the green-build pre-commit chain passing while later slices grow the directive parser, tag resolver, diff renderer, and splicer. ACs 4 and 5 get their own assertions in slice 5.
2	Directive parser as a pure unit. New `src/diff.rs` exposes `parse_directives` returning byte-span-tagged `DiffDirectives`. Unit-tested in isolation; not yet wired into the preprocessor.
3	Tag resolution. `diff::resolve` looks each operand up in `Manifest` (re-using `Manifest::find` from ch. 2) and produces a structured error for missing tags carrying enough context for the splicer to format an AC-3 diagnostic. Unit-tested.
4	Unified diff computation via the `similar` crate. `diff::render` takes the resolved bytes plus labels and produces unified-diff text; identical bytes produce a "no changes" notice rather than an empty block (AC 4). Unit-tested with synthetic byte pairs.
5	Splicer wires slices 2–4 into the no-op preprocessor: every `{{#diff …}}` directive is replaced with a fenced ```` ```diff ```` block, the parser learns to skip directives inside fenced code blocks (initial AC 6 — so chapters can quote literal directive examples), and `cargo run -- install --book-root book` registers `[preprocessor.listings]` in our own `book/book.toml` so the book exercises the diff primitive on every build. Slice 1's integration test goes green; AC 5 gets its own integration test pinning surrounding-content invariance.
6	`live:<path>` operand (initial AC 7). Recognised in either operand position; the resolver reads the live file from disk relative to `book_root`.
7 (refactor)	Remove `parse_escapes`, the escape branch in `splice_chapter`, and the matching tests — dead code in the real mdbook pipeline. Tidy duplication that emerged across slices 2–6.
8	Tighten ACs 6 and 7 in response to dogfooding. Inline code spans (`` `…` ``) join fenced blocks as a directive-skip context (AC 6) — `{{#diff a b}}` in inline backticks no longer crashes the build. `live:<path>` resolution moves from

	book-root-relative to chapter-source-relative (AC 7), matching mdbook's `{{#include}}` convention. Both come from real friction points hit while writing this very chapter.
wrap-up	Update `ROADMAP.md` to mark the diff primitive shipped.

Outside-in narrative

Slice 1 — failing integration test + no-op pass-through

The first slice introduces a CLI-level integration test that pipes a preprocessor envelope to `mdbook-listings`'s no-subcommand arm and asserts on the round-tripped chapter content. The arm itself becomes a no-op pass-through — the smallest possible body that still satisfies the wire format mdbook expects. The test fails because pass-through doesn't add a diff fence, and is `#[ignore]`'d so the green-build chain stays passing while slices 2–4 grow the pieces it needs.

`Cargo.toml` gains two runtime deps: `mdbook-preprocessor` (for the `PreprocessorContext` and `Book` types plus the `parse_input` helper) and `serde_json` (for the round-trip serialisation that `parse_input`'s counterpart writes back to stdout). **What's new in `cargo-toml-v2` compared to `cargo-toml-v1`:** the `mdbook-preprocessor = "0.5"` and `serde_json = "1"` lines added inside `[dependencies]` in alphabetical position. Everything else is unchanged.

```
[package]
name = "mdbook-listings"
version = "0.1.0"
edition = "2024"
rust-version = "1.88"
license = "MIT"
description = "Managed code listings for mdbook: inline callouts, freezing, and
verification"
repository = "https://github.com/padamson/mdbook-listings"
categories = ["command-line-utilities", "text-processing"]
keywords = ["mdbook", "preprocessor", "documentation", "code-listing"]

[dependencies]
anyhow = "1"
clap = { version = "4", features = ["derive"] }
mdbook-preprocessor = "0.5"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
sha2 = "0.11"
toml = "1.1"
toml_edit = "0.22"

[dev-dependencies]
assert_cmd = "2"
predicates = "3"
tempfile = "3"
```

`src/main.rs`'s `preprocess` function used to bail with `not yet implemented`; it now reads the JSON envelope from stdin via `mdbook_preprocessor::parse_input`, discards the `PreprocessorContext` (slice 3 is the first to need it), and writes the book straight back to stdout via `serde_json::to_writer`. Both calls are fully qualified so no new `use` statements are

needed yet. **What's new in `main-v3` compared to `main-v2`:** the body of `preprocess` is replaced with the `parse_input` → `to_writer` round-trip; the doc comment on `preprocess` is unchanged. Everything else — the `clap` derive struct, every other subcommand handler, `supports`, `main`/`run` — is byte-identical.

```
use std::path::PathBuf;
use std::process;

use anyhow::Result;
use clap::{Parser, Subcommand};
use mdbook_listings::freeze::{FreezeOptions, FreezeOutcome, freeze};
use mdbook_listings::install::{InstallOutcome, install};

/// Managed code listings for mdbook: inline callouts, freezing, and
verification.
#[derive(Parser)]
#[command(version, about, long_about = None)]
struct Cli {
    #[command(subcommand)]
    command: Option<Command>,
}

#[derive(Subcommand)]
enum Command {
    /// Check whether a renderer is supported by this preprocessor.
    ///
    /// Invoked by mdbook during the build to decide whether to pipe the book
    /// through this preprocessor for a given renderer. Exits 0 if supported,
    /// 1 otherwise.
    Supports {
        /// Name of the renderer mdbook is asking about (e.g. `html`, `typst-
pdf`).
        renderer: String,
    },

    /// Install preprocessor assets and register mdbook-listings in `book.toml`.
    Install {
        /// Root directory of the book (contains `book.toml`). Defaults to the
        /// current directory.
        #[arg(long)]
        book_root: Option<PathBuf>,
    },

    /// Freeze a source file into the book's listings directory and update
    /// the manifest.
    Freeze {
        /// Human-readable tag used as the frozen filename and as the manifest
        /// entry key. Should be unique within the book.
        #[arg(long)]
        tag: String,

        /// Root directory of the book. Defaults to the current directory.
        #[arg(long)]
        book_root: Option<PathBuf>,
```

```
        /// Overwrite an existing frozen copy with the same tag.
        #[arg(long)]
        force: bool,

        /// Path to the source file to freeze.
        source: PathBuf,
    },

    /// Verify consistency between the manifest, frozen listings, and
`{{#include}}`
    /// references in the book's markdown.
    Verify {
        /// Root directory of the book. Defaults to the current directory.
        #[arg(long)]
        book_root: Option<PathBuf>,
    },
}

fn main() {
    if let Err(err) = run() {
        eprintln!("error: {err:?}");
        process::exit(1);
    }
}

fn run() -> Result<()> {
    let cli = Cli::parse();
    match cli.command {
        None => preprocess(),
        Some(Command::Supports { renderer }) => supports(&renderer),
        Some(Command::Install { book_root }) => {
            let book_root = book_root.unwrap_or_else(|| PathBuf::from("."));
            match install(&book_root)? {
                InstallOutcome::Installed => {
                    println!("installed mdbook-listings into {}",
book_root.display());
                }
                InstallOutcome::Unchanged => {
                    println!(
                        "mdbook-listings already installed in {}; nothing
changed",
                        book_root.display(),
                    );
                }
            }
            Ok(())
        }
        Some(Command::Freeze {
            tag,
            book_root,
            force,
            source,
        }) => {
            let book_root = book_root.unwrap_or_else(|| PathBuf::from("."));
            let outcome = freeze(FreezeOptions {
```

```
                book_root: &book_root,
                tag: &tag,
                source: &source,
                force,
            })?;
            let verb = match outcome {
                FreezeOutcome::Created => "created",
                FreezeOutcome::Unchanged => "unchanged",
                FreezeOutcome::Replaced => "replaced",
            };
            println!("{verb}: {tag}");
            Ok(())
        }
        Some(Command::Verify { book_root: _ }) => {
            anyhow::bail!("`mdbook-listings verify` is not yet implemented")
        }
    }
}

/// Default mode: read an mdbook preprocessor JSON payload from stdin, emit the
/// transformed payload on stdout.
fn preprocess() -> Result<()> {
    let (_ctx, book) = mdbook_preprocessor::parse_input(std::io::stdin())?;
    serde_json::to_writer(std::io::stdout(), &book)?;
    Ok(())
}

/// Answer mdbook's renderer-support probe by exiting 0 (supported) or 1
/// (unsupported). We do not return from this function.
fn supports(renderer: &str) -> ! {
    let supported = matches!(renderer, "html" | "typst-pdf");
    process::exit(if supported { 0 } else { 1 });
}
```

The integration test lives in a new `tests/diffs.rs` (per ch. 0's "one integration-test file per story" rule). The file contains one test plus a `MinimalDiffsBook` helper that materialises a tempdir holding `book.toml`, `book/listings.toml`, and two stub frozen files under `book/src/listings/`. The helper's `envelope_with_chapter` method builds the `(PreprocessorContext, Book)` tuple from public mdbook constructors (`PreprocessorContext::new`, `Chapter::new`, `Book::new_with_items`) and serialises the pair as a two-element JSON array — the exact shape mdbook itself sends a preprocessor.

```
//! Integration tests for the Show Diffs Between Slices story (ch. 3).

use std::fs;
use std::path::{Path, PathBuf};

use mdbook_preprocessor::PreprocessorContext;
use mdbook_preprocessor::book::{Book, BookItem, Chapter};
use mdbook_preprocessor::config::Config;
use tempfile::TempDir;

mod common;
```

```
use common::mdbook_listings;

#[test]
#[ignore = "directive parsing + diff rendering land in slices 2-5"]
fn diff_directive_renders_to_fenced_diff_block() {
    let book = MinimalDiffsBook::new();
    let envelope = book.envelope_with_chapter(
        "Before paragraph.\n\n{{#diff old-tag new-tag}}\n\nAfter paragraph.\n",
    );

    let output = mdbook_listings()
        .write_stdin(envelope)
        .assert()
        .success()
        .get_output()
        .stdout
        .clone();

    let returned: Book = serde_json::from_slice(&output).expect("parse stdout as
Book");
    let content = chapter_content(&returned, "Diff Test");

    assert!(
        content.contains("```diff"),
        "expected the directive to render as a ```diff fenced block; got:
\n{content}",
    );
}

/// The smallest tempdir fixture that backs the diff preprocessor: a `book.toml`
/// that registers `[preprocessor.listings]`, a `book/listings.toml` with two
/// frozen entries the chapters can reference by tag, and the matching frozen
/// files. The tempdir is destroyed when the struct drops.
struct MinimalDiffsBook {
    _tmp: TempDir,
    root: PathBuf,
}

impl MinimalDiffsBook {
    fn new() -> Self {
        let tmp = TempDir::new().expect("tempdir");
        let root = tmp.path().to_path_buf();

        fs::write(
            root.join("book.toml"),
            "[book]\ntitle = \"Test\"\n\n[preprocessor.listings]\ncommand =
\"mdbook-listings\"\n",
        )
        .unwrap();

        let listings_dir = root.join("book").join("src").join("listings");
        fs::create_dir_all(&listings_dir).unwrap();
        fs::write(listings_dir.join("old-tag.txt"), "line one\nline
two\n").unwrap();
        fs::write(listings_dir.join("new-tag.txt"), "line one\nline
```

```
TWO\n").unwrap();

        fs::write(
            root.join("book").join("listings.toml"),
            "version = 1\n\n\
             [[listing]]\n\
             tag = \"old-tag\"\n\
             source = \"../old.txt\"\n\
             frozen = \"src/listings/old-tag.txt\"\n\
             sha256 =
\"0000000000000000000000000000000000000000000000000000000000000000\"\n\n\
             [[listing]]\n\
             tag = \"new-tag\"\n\
             source = \"../new.txt\"\n\
             frozen = \"src/listings/new-tag.txt\"\n\
             sha256 =
\"0000000000000000000000000000000000000000000000000000000000000000\"\n",
        )
        .unwrap();

        Self { _tmp: tmp, root }
    }

    #[allow(dead_code)]
    fn root(&self) -> &Path {
        &self.root
    }

    /// Build the JSON envelope mdbook would send a preprocessor: the tuple
    /// `(PreprocessorContext, Book)` serialised as a two-element JSON array,
    /// with one chapter carrying the supplied markdown.
    fn envelope_with_chapter(&self, chapter_content: &str) -> String {
        let ctx =
            PreprocessorContext::new(self.root.clone(), Config::default(),
"html".to_string());
        let chapter = Chapter::new(
            "Diff Test",
            chapter_content.to_string(),
            "diff-test.md",
            vec![],
        );
        let book = Book::new_with_items(vec![BookItem::Chapter(chapter)]);
        serde_json::to_string(&(&ctx, &book)).expect("serialize envelope")
    }
}

fn chapter_content(book: &Book, chapter_name: &str) -> String {
    book.iter()
        .find_map(|item| match item {
            BookItem::Chapter(ch) if ch.name == chapter_name =>
Some(ch.content.clone()),
            _ => None,
        })
        .unwrap_or_else(|| panic!("chapter `{chapter_name}` missing from
```

```
returned book"))
}
```

`#[ignore]` (with a reason that names the slices that close it out) keeps `cargo nextest run` green while the diff machinery is being built. The test was confirmed to fail at the assertion, not at the `assert().success()` step — the pass-through arm parses the envelope, serialises the book unchanged, and exits zero, so the assertion that the chapter content contains a ```` ```diff ```` fence is what's red.

The `MinimalDiffsBook` fixture is deliberately bigger than the test needs in slice 1 (the stub frozen files are unused while pass-through is the whole pipeline). This pays off in slices 3 and 5 when the resolver and splicer reach for those frozen bytes — the only test change in slice 5 is removing `#[ignore]`, no fixture rewiring.

`MinimalDiffsBook::root` is currently `#[allow(dead_code)]` for the same reason: slice 6's `live:<path>` test will need it to write ad-hoc files into the tempdir. Carrying the accessor here keeps the helper's surface stable across slices.

Slice 2 — directive parser as a pure unit

Slice 2 stands up the first piece slice 5's splicer will need: the parser that turns a chapter's markdown into a list of `{{#diff …}}` directives with byte spans. It's a pure function — no IO, no manifest, no diff library — so its unit tests pin its behaviour completely without touching disk.

A new `src/diff.rs` module declares `DiffDirective { left, right, span }` and the free function `parse_directives(content) → Vec<DiffDirective>`:

```
//! Parses `{{#diff <left> <right>}}` directives out of chapter markdown.
//! The resolver and renderer that consume the parsed [`DiffDirective`]s land
//! in later slices of the *Show Diffs Between Slices* story.

use std::ops::Range;

/// One parsed `{{#diff <left> <right>}}` directive. `span` indexes into the
/// chapter content the parser was handed and covers the directive in full
/// (`{{#diff …}}` inclusive) so the splicer can replace the whole substring
/// in one pass.
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct DiffDirective {
    pub left: String,
    pub right: String,
    pub span: Range<usize>,
}

/// Walks `content` and returns every well-formed `{{#diff a b}}` directive.
/// Directives prefixed with a backslash (`{{#diff …}}`, matching mdbook's
/// `{{#include}}` escape convention) are skipped here; the splicer that
/// lands later strips the leading backslash so the literal directive renders
/// to the reader. Directives with the wrong arity (`{{#diff a}}`,
/// `{{#diff a b c}}`) are silently skipped — the resolver in the next slice
/// surfaces the useful diagnostic, and being over-eager here would fight it.
pub fn parse_directives(content: &str) -> Vec<DiffDirective> {
    const PREFIX: &[u8] = b"{{#diff";
    let bytes = content.as_bytes();
```

```
    let mut out = Vec::new();
    let mut i = 0;
    while i + PREFIX.len() ⩽ bytes.len() {
        if &bytes[i..i + PREFIX.len()] ≠ PREFIX {
            i += 1;
            continue;
        }
        if i > 0 && bytes[i - 1] == b'\\' {
            i += PREFIX.len();
            continue;
        }
        let inner_start = i + PREFIX.len();
        let Some(end_rel) = content[inner_start..].find("}}") else {
            break;
        };
        let directive_end = inner_start + end_rel + 2;
        let tokens: Vec<&str> = content[inner_start..inner_start + end_rel]
            .split_whitespace()
            .collect();
        if tokens.len() == 2 {
            out.push(DiffDirective {
                left: tokens[0].to_string(),
                right: tokens[1].to_string(),
                span: i..directive_end,
            });
        }
        i = directive_end;
    }
    out
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn parse_directives_extracts_well_formed_directive() {
        let s = "before {{#diff old-tag new-tag}} after";
        let got = parse_directives(s);
        assert_eq!(got.len(), 1, "expected one directive; got {got:?}");
        assert_eq!(got[0].left, "old-tag");
        assert_eq!(got[0].right, "new-tag");
        assert_eq!(&s[got[0].span.clone()], "{{#diff old-tag new-tag}}");
    }

    #[test]
    fn parse_directives_handles_multiple_occurrences() {
        let s = "{{#diff a b}} mid {{#diff c d}}";
        let got = parse_directives(s);
        assert_eq!(got.len(), 2);
        assert_eq!(got[0].left, "a");
        assert_eq!(got[1].right, "d");
        assert_eq!(&s[got[0].span.clone()], "{{#diff a b}}");
        assert_eq!(&s[got[1].span.clone()], "{{#diff c d}}");
    }
```

```
    #[test]
    fn parse_directives_skips_escaped_form() {
        let s = "use \{{#diff a b}} verbatim";
        let got = parse_directives(s);
        assert!(
            got.is_empty(),
            "escaped directive should not parse; got {got:?}",
        );
    }

    #[test]
    fn parse_directives_tolerates_extra_whitespace_around_operands() {
        let s = "{{#diff   a    b   }}";
        let got = parse_directives(s);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].left, "a");
        assert_eq!(got[0].right, "b");
    }

    #[test]
    fn parse_directives_skips_malformed_arity() {
        for s in ["{{#diff only-one}}", "{{#diff a b c}}", "{{#diff}}"] {
            let got = parse_directives(s);
            assert!(
                got.is_empty(),
                "malformed directive `{s}` should not parse; got {got:?}",
            );
        }
    }

    #[test]
    fn parse_directives_accepts_arbitrary_operand_strings() {
        let s = "{{#diff live:src/foo.rs new-tag}}";
        let got = parse_directives(s);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].left, "live:src/foo.rs");
        assert_eq!(got[0].right, "new-tag");
    }
}
```

The parser walks `content` byte-wise, looking for `{{#diff`. When it finds one, it checks the byte before for a backslash (the escape AC 6 calls out — kept here as a *skip*, not a strip; the splicer in slice 5 owns the rewrite that drops the leading `\` so the literal directive renders to the reader). It then locates the next `}}`, splits the inner text on whitespace, and only yields a directive when there are exactly two operands. Wrong-arity directives (`{{#diff a}}`, `{{#diff a b c}}`) are silently skipped — surfacing "that's the wrong number of arguments" diagnostics is the resolver's job in slice 3, where the chapter source path and line number are already in scope.

Six unit tests pin the contract: well-formed directives parse and their spans cover the whole `{{#diff …}}` substring; multiple directives in one chapter all parse with correct spans; the escaped form is skipped; whitespace around operands is tolerated; wrong-arity directives are skipped; and arbitrary operand strings (including the future `live:src/foo.rs` shape) are accepted at the parse layer (the resolver decides what they mean).

`src/lib.rs` gains one line — `pub mod diff;` — so `src/main.rs` and the integration tests can reach the new module.

What's new in `lib-v3` compared to `lib-v2`: the `pub mod diff;` line, in alphabetical position. Everything else is unchanged.

```
//! Managed code listings for mdbook.

pub mod diff;
pub mod freeze;
pub mod install;
pub mod manifest;
```

The integration test from slice 1 is still `#[ignore]`'d. The parser is plumbing — slices 3 and 4 add the resolver and renderer that the splicer in slice 5 wires together to make the assertion pass.

Slice 3 — tag resolution + missing-tag diagnostic

Slice 3 turns each parsed `DiffDirective` into the *bytes* a diff renderer can consume: it looks the operand up in the manifest (re-using `Manifest::find` from ch. 2), reads the frozen file from disk, and returns a `ResolvedDiff` carrying both halves' bytes plus labels for the unified-diff headers. When an operand is unknown or its frozen file is missing, the resolver returns a typed `ResolveError` carrying the offending tag name — the splicer in slice 5 wraps that with the chapter source path and 1-based line number derived from the directive's byte span, which together satisfy AC 3.

What's new in `diff-v2` compared to `diff-v1`: the `ResolvedDiff` struct, the `ResolveError` / `ResolveErrorKind` types with manual `Display` and `Error` impls, the `resolve` and `resolve_operand` functions, the `crate::manifest::Manifest` import they need, and four new tests covering the happy path plus the three failure shapes (unknown left tag, unknown right tag, frozen file absent from disk). The tests share a `fixture` helper that materialises a tempdir with two stub frozen files plus an in-memory `Manifest` pointing at them; building the manifest in memory rather than via `Manifest::load` keeps the unit tests independent of the manifest file format. The parser, its tests, and the module's existing imports are unchanged.

```
//! Parses `{{#diff <left> <right>}}` directives out of chapter markdown and
//! resolves each operand tag to the bytes the renderer will diff against.
//! The unified-diff renderer that consumes a [`ResolvedDiff`] lands in slice
//! 4 of the *Show Diffs Between Slices* story.

use std::ops::Range;
use std::path::{Path, PathBuf};

use crate::manifest::Manifest;

/// One parsed `{{#diff <left> <right>}}` directive. `span` indexes into the
/// chapter content the parser was handed and covers the directive in full
/// (`{{#diff …}}` inclusive) so the splicer can replace the whole substring
/// in one pass.
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct DiffDirective {
    pub left: String,
    pub right: String,
```

```
    pub span: Range<usize>,
}

/// Walks `content` and returns every well-formed `{{#diff a b}}` directive.
/// Directives prefixed with a backslash (`{{#diff …}}`, matching mdbook's
/// `{{#include}}` escape convention) are skipped here; the splicer that
/// lands later strips the leading backslash so the literal directive renders
/// to the reader. Directives with the wrong arity (`{{#diff a}}`,
/// `{{#diff a b c}}`) are silently skipped — the resolver in the next slice
/// surfaces the useful diagnostic, and being over-eager here would fight it.
pub fn parse_directives(content: &str) -> Vec<DiffDirective> {
    const PREFIX: &[u8] = b"{{#diff";
    let bytes = content.as_bytes();
    let mut out = Vec::new();
    let mut i = 0;
    while i + PREFIX.len() <= bytes.len() {
        if &bytes[i..i + PREFIX.len()] != PREFIX {
            i += 1;
            continue;
        }
        if i > 0 && bytes[i - 1] == b'\\' {
            i += PREFIX.len();
            continue;
        }
        let inner_start = i + PREFIX.len();
        let Some(end_rel) = content[inner_start..].find("}}") else {
            break;
        };
        let directive_end = inner_start + end_rel + 2;
        let tokens: Vec<&str> = content[inner_start..inner_start + end_rel]
            .split_whitespace()
            .collect();
        if tokens.len() == 2 {
            out.push(DiffDirective {
                left: tokens[0].to_string(),
                right: tokens[1].to_string(),
                span: i..directive_end,
            });
        }
        i = directive_end;
    }
    out
}

/// The bytes plus labels needed to render a unified diff. Labels become the
/// `--- <left_label>` / `+++ <right_label>` headers in the rendered output.
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct ResolvedDiff {
    pub left_label: String,
    pub left_bytes: Vec<u8>,
    pub right_label: String,
    pub right_bytes: Vec<u8>,
}

/// Why an operand could not be turned into bytes. The splicer in slice 5
```

```
/// wraps this with the chapter source path and 1-based line number derived
/// from the directive's span — the location context AC 3 demands.
#[derive(Debug)]
pub struct ResolveError {
    pub tag: String,
    pub kind: ResolveErrorKind,
}

#[derive(Debug)]
pub enum ResolveErrorKind {
    UnknownTag,
    FrozenFileMissing {
        frozen_path: PathBuf,
        source: std::io::Error,
    },
}

impl std::fmt::Display for ResolveError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match &self.kind {
            ResolveErrorKind::UnknownTag => {
                write!(f, "no listing with tag `{}` in manifest", self.tag)
            }
            ResolveErrorKind::FrozenFileMissing {
                frozen_path,
                source,
            } => write!(
                f,
                "manifest tag `{}` references frozen file `{}` which cannot be
read: {}",
                self.tag,
                frozen_path.display(),
                source,
            ),
        }
    }
}

impl std::error::Error for ResolveError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        match &self.kind {
            ResolveErrorKind::UnknownTag => None,
            ResolveErrorKind::FrozenFileMissing { source, .. } => Some(source),
        }
    }
}

/// Look each operand of `directive` up in `manifest`, read the frozen bytes
/// from `<book_root>/<listing.frozen>`, and return them paired with labels
/// the renderer can use in unified-diff headers. Stops at the first failing
/// operand: if the left tag is unknown the right tag is not consulted, so
/// the diagnostic the splicer surfaces names a single missing tag rather
/// than two.
pub fn resolve(
    directive: &DiffDirective,
```

```
    manifest: &Manifest,
    book_root: &Path,
) -> Result<ResolvedDiff, ResolveError> {
    let (left_label, left_bytes) = resolve_operand(&directive.left, manifest,
book_root)?;
    let (right_label, right_bytes) = resolve_operand(&directive.right, manifest,
book_root)?;
    Ok(ResolvedDiff {
        left_label,
        left_bytes,
        right_label,
        right_bytes,
    })
}

fn resolve_operand(
    operand: &str,
    manifest: &Manifest,
    book_root: &Path,
) -> Result<(String, Vec<u8>), ResolveError> {
    let listing = manifest.find(operand).ok_or_else(|| ResolveError {
        tag: operand.to_string(),
        kind: ResolveErrorKind::UnknownTag,
    })?;
    let frozen_path = book_root.join(&listing.frozen);
    let bytes = std::fs::read(&frozen_path).map_err(|source| ResolveError {
        tag: operand.to_string(),
        kind: ResolveErrorKind::FrozenFileMissing {
            frozen_path: frozen_path.clone(),
            source,
        },
    })?;
    Ok((operand.to_string(), bytes))
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn parse_directives_extracts_well_formed_directive() {
        let s = "before {{#diff old-tag new-tag}} after";
        let got = parse_directives(s);
        assert_eq!(got.len(), 1, "expected one directive; got {got:?}");
        assert_eq!(got[0].left, "old-tag");
        assert_eq!(got[0].right, "new-tag");
        assert_eq!(&s[got[0].span.clone()], "{{#diff old-tag new-tag}}");
    }

    #[test]
    fn parse_directives_handles_multiple_occurrences() {
        let s = "{{#diff a b}} mid {{#diff c d}}";
        let got = parse_directives(s);
        assert_eq!(got.len(), 2);
        assert_eq!(got[0].left, "a");
```

```
        assert_eq!(got[1].right, "d");
        assert_eq!(&s[got[0].span.clone()], "{{#diff a b}}");
        assert_eq!(&s[got[1].span.clone()], "{{#diff c d}}");
    }

    #[test]
    fn parse_directives_skips_escaped_form() {
        let s = "use \{{#diff a b}} verbatim";
        let got = parse_directives(s);
        assert!(
            got.is_empty(),
            "escaped directive should not parse; got {got:?}",
        );
    }

    #[test]
    fn parse_directives_tolerates_extra_whitespace_around_operands() {
        let s = "{{#diff   a    b   }}";
        let got = parse_directives(s);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].left, "a");
        assert_eq!(got[0].right, "b");
    }

    #[test]
    fn parse_directives_skips_malformed_arity() {
        for s in ["{{#diff only-one}}", "{{#diff a b c}}", "{{#diff}}"] {
            let got = parse_directives(s);
            assert!(
                got.is_empty(),
                "malformed directive `{s}` should not parse; got {got:?}",
            );
        }
    }

    #[test]
    fn parse_directives_accepts_arbitrary_operand_strings() {
        let s = "{{#diff live:src/foo.rs new-tag}}";
        let got = parse_directives(s);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].left, "live:src/foo.rs");
        assert_eq!(got[0].right, "new-tag");
    }

    use crate::manifest::{MANIFEST_VERSION, Manifest};
    use std::fs;
    use tempfile::TempDir;

    /// Build a tempdir book root with two frozen files and an in-memory
    /// manifest pointing at them. The directive's span is irrelevant to the
    /// resolver — slice 5 uses it to compute line numbers when surfacing
    /// errors — so the helper hands back a span-less directive built from
    /// just the operand tags.
    fn fixture(left_bytes: &[u8], right_bytes: &[u8]) -> (TempDir, Manifest,
DiffDirective) {
```

```
        let tmp = TempDir::new().expect("tempdir");
        let listings_dir = tmp.path().join("src").join("listings");
        fs::create_dir_all(&listings_dir).unwrap();
        fs::write(listings_dir.join("left-tag.txt"), left_bytes).unwrap();
        fs::write(listings_dir.join("right-tag.txt"), right_bytes).unwrap();

        let manifest = Manifest {
            version: MANIFEST_VERSION,
            listings: vec![
                crate::manifest::Listing {
                    tag: "left-tag".into(),
                    source: "../left.txt".into(),
                    frozen: "src/listings/left-tag.txt".into(),
                    sha256: "0".repeat(64),
                },
                crate::manifest::Listing {
                    tag: "right-tag".into(),
                    source: "../right.txt".into(),
                    frozen: "src/listings/right-tag.txt".into(),
                    sha256: "0".repeat(64),
                },
            ],
        };

        let directive = DiffDirective {
            left: "left-tag".into(),
            right: "right-tag".into(),
            span: 0..0,
        };

        (tmp, manifest, directive)
    }

    #[test]
    fn resolve_returns_bytes_and_labels_for_known_tags() {
        let (tmp, manifest, directive) = fixture(b"line one\nline two\n", b"line
one\nline TWO\n");
        let resolved = resolve(&directive, &manifest,
tmp.path()).expect("resolve");
        assert_eq!(resolved.left_label, "left-tag");
        assert_eq!(resolved.right_label, "right-tag");
        assert_eq!(resolved.left_bytes, b"line one\nline two\n");
        assert_eq!(resolved.right_bytes, b"line one\nline TWO\n");
    }

    #[test]
    fn resolve_returns_unknown_tag_error_for_missing_left_operand() {
        let (tmp, manifest, mut directive) = fixture(b"a", b"b");
        directive.left = "nope".into();

        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
        assert_eq!(err.tag, "nope");
        assert!(matches!(err.kind, ResolveErrorKind::UnknownTag));
        let msg = format!("{err}");
```

```
        assert!(
            msg.contains("`nope`"),
            "diagnostic should name the missing tag; got: {msg}",
        );
    }

    #[test]
    fn resolve_returns_unknown_tag_error_for_missing_right_operand() {
        let (tmp, manifest, mut directive) = fixture(b"a", b"b");
        directive.right = "also-nope".into();

        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
        assert_eq!(err.tag, "also-nope");
        assert!(matches!(err.kind, ResolveErrorKind::UnknownTag));
    }

    #[test]
    fn resolve_returns_frozen_file_missing_when_disk_lacks_frozen_copy() {
        let (tmp, manifest, directive) = fixture(b"a", b"b");
        fs::remove_file(tmp.path().join("src/listings/left-tag.txt")).unwrap();

        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
        assert_eq!(err.tag, "left-tag");
        match &err.kind {
            ResolveErrorKind::FrozenFileMissing { frozen_path, .. } => {
                assert!(
                    frozen_path.ends_with("src/listings/left-tag.txt"),
                    "diagnostic should name the absent file; got
{frozen_path:?}",
                );
            }
            other => panic!("expected FrozenFileMissing; got {other:?}"),
        }
    }
}
```

The resolver stops at the first failing operand: if the left tag is unknown, the right tag is not consulted. That keeps slice 5's diagnostic naming a single missing tag rather than two, matching how an author would actually fix the chapter (find the typo, fix the typo, rebuild — the second tag's resolution happens on the rebuild). It also means tests for the right-operand failure path have to use a known left operand, which is what the `resolve_returns_unknown_tag_error_for_missing_right_operand` test does.

`live:<path>` operands (AC 7) currently fall through to the `UnknownTag` arm — `manifest.find("live:src/foo.rs")` returns `None`. Slice 6 inserts a prefix check before the manifest lookup and reads the file directly, leaving this slice's resolver unchanged for the all-frozen happy path.

The integration test from slice 1 is still `#[ignore]`'d. Slice 4 adds the renderer that turns a `ResolvedDiff` into unified-diff text; slice 5 wires parser → resolver → renderer into the preprocessor and removes the `#[ignore]`.

Slice 4 — unified diff computation via `similar`

Slice 4 adds the third and final pure unit slice 5's splicer needs: `render(left, right, left_label, right_label) → String`, which turns two `&str` halves into unified-diff text. The actual diff algorithm is delegated to the `similar` crate (`TextDiff::from_lines(...).unified_diff().header(a, b)`); the function's only original behaviour is the AC-4 short-circuit — identical inputs return a one-line `(no changes between left and right)` notice rather than the empty string `similar` would otherwise emit, which would render as a fence body that looks broken to a reader.

`Cargo.toml` gains `similar = "2"`. **What's new in `cargo-toml-v3` compared to `cargo-toml-v2`:** the single `similar = "2"` line in alphabetical position inside `[dependencies]`. Everything else is unchanged.

```
[package]
name = "mdbook-listings"
version = "0.1.0"
edition = "2024"
rust-version = "1.88"
license = "MIT"
description = "Managed code listings for mdbook: inline callouts, freezing, and
verification"
repository = "https://github.com/padamson/mdbook-listings"
categories = ["command-line-utilities", "text-processing"]
keywords = ["mdbook", "preprocessor", "documentation", "code-listing"]

[dependencies]
anyhow = "1"
clap = { version = "4", features = ["derive"] }
mdbook-preprocessor = "0.5"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
sha2 = "0.11"
similar = "2"
toml = "1.1"
toml_edit = "0.22"

[dev-dependencies]
assert_cmd = "2"
predicates = "3"
tempfile = "3"
```

`src/diff.rs` grows the `render` function plus four unit tests covering the four shapes that matter: differing inputs produce a header-and-hunks unified diff; identical inputs short-circuit to the no-changes notice; two empty inputs do the same; pure additions render with `+` prefixes and no spurious `-` lines. This slice also tightens the doc comments on the parser, resolver, and error types added in slices 2–3 to drop forward references to later slices and acceptance-criteria numbers — the chapter narrative is the right place to talk about story structure, the source code is the right place to talk about behaviour. The behaviour itself is unchanged.

```
//! Parses `{{#diff <left> <right>}}` directives out of chapter markdown,
//! resolves each operand tag to bytes via the manifest, and renders the
//! pair as unified-diff text.
```

```
use std::ops::Range;
use std::path::{Path, PathBuf};

use crate::manifest::Manifest;

/// `span` covers the directive in full (`{{#diff …}}` inclusive) so callers
/// can replace the whole substring in one pass.
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct DiffDirective {
    pub left: String,
    pub right: String,
    pub span: Range<usize>,
}

/// Returns every well-formed `{{#diff a b}}` directive in `content`.
/// Backslash-escaped directives (`{{#diff …}}`, matching mdbook's
/// `{{#include}}` convention) and wrong-arity matches are skipped silently —
/// the caller is in a better position to surface either with chapter-source
/// context.
pub fn parse_directives(content: &str) -> Vec<DiffDirective> {
    const PREFIX: &[u8] = b"{{#diff";
    let bytes = content.as_bytes();
    let mut out = Vec::new();
    let mut i = 0;
    while i + PREFIX.len() <= bytes.len() {
        if &bytes[i..i + PREFIX.len()] != PREFIX {
            i += 1;
            continue;
        }
        if i > 0 && bytes[i - 1] == b'\\' {
            i += PREFIX.len();
            continue;
        }
        let inner_start = i + PREFIX.len();
        let Some(end_rel) = content[inner_start..].find("}}") else {
            break;
        };
        let directive_end = inner_start + end_rel + 2;
        let tokens: Vec<&str> = content[inner_start..inner_start + end_rel]
            .split_whitespace()
            .collect();
        if tokens.len() == 2 {
            out.push(DiffDirective {
                left: tokens[0].to_string(),
                right: tokens[1].to_string(),
                span: i..directive_end,
            });
        }
        i = directive_end;
    }
    out
}

/// Labels become the `--- <left_label>` / `+++ <right_label>` headers in
```

```
/// the rendered output.
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct ResolvedDiff {
    pub left_label: String,
    pub left_bytes: Vec<u8>,
    pub right_label: String,
    pub right_bytes: Vec<u8>,
}

/// `tag` is exposed so callers can compose a chapter-source-located
/// diagnostic that names the offending operand.
#[derive(Debug)]
pub struct ResolveError {
    pub tag: String,
    pub kind: ResolveErrorKind,
}

#[derive(Debug)]
pub enum ResolveErrorKind {
    UnknownTag,
    FrozenFileMissing {
        frozen_path: PathBuf,
        source: std::io::Error,
    },
}

impl std::fmt::Display for ResolveError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match &self.kind {
            ResolveErrorKind::UnknownTag => {
                write!(f, "no listing with tag `{}` in manifest", self.tag)
            }
            ResolveErrorKind::FrozenFileMissing {
                frozen_path,
                source,
            } => write!(
                f,
                "manifest tag `{}` references frozen file `{}` which cannot be
read: {}",
                self.tag,
                frozen_path.display(),
                source,
            ),
        }
    }
}

impl std::error::Error for ResolveError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        match &self.kind {
            ResolveErrorKind::UnknownTag => None,
            ResolveErrorKind::FrozenFileMissing { source, .. } => Some(source),
        }
    }
}
```

```
/// Returns at the first failing operand so callers surface one missing tag
/// at a time — the second tag's resolution can wait for the rebuild after
/// the first fix.
pub fn resolve(
    directive: &DiffDirective,
    manifest: &Manifest,
    book_root: &Path,
) -> Result<ResolvedDiff, ResolveError> {
    let (left_label, left_bytes) = resolve_operand(&directive.left, manifest,
book_root)?;
    let (right_label, right_bytes) = resolve_operand(&directive.right, manifest,
book_root)?;
    Ok(ResolvedDiff {
        left_label,
        left_bytes,
        right_label,
        right_bytes,
    })
}

fn resolve_operand(
    operand: &str,
    manifest: &Manifest,
    book_root: &Path,
) -> Result<(String, Vec<u8>), ResolveError> {
    let listing = manifest.find(operand).ok_or_else(|| ResolveError {
        tag: operand.to_string(),
        kind: ResolveErrorKind::UnknownTag,
    })?;
    let frozen_path = book_root.join(&listing.frozen);
    let bytes = std::fs::read(&frozen_path).map_err(|source| ResolveError {
        tag: operand.to_string(),
        kind: ResolveErrorKind::FrozenFileMissing {
            frozen_path: frozen_path.clone(),
            source,
        },
    })?;
    Ok((operand.to_string(), bytes))
}

/// Identical inputs return a one-line notice rather than the empty string
/// `similar` would otherwise produce — a fence body that's just the header
/// looks broken to a reader.
pub fn render(left: &str, right: &str, left_label: &str, right_label: &str) ->
String {
    if left == right {
        return format!("(no changes between {left_label} and {right_label})\n");
    }
    similar::TextDiff::from_lines(left, right)
        .unified_diff()
        .context_radius(3)
        .header(left_label, right_label)
        .to_string()
}
```

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn parse_directives_extracts_well_formed_directive() {
        let s = "before {{#diff old-tag new-tag}} after";
        let got = parse_directives(s);
        assert_eq!(got.len(), 1, "expected one directive; got {got:?}");
        assert_eq!(got[0].left, "old-tag");
        assert_eq!(got[0].right, "new-tag");
        assert_eq!(&s[got[0].span.clone()], "{{#diff old-tag new-tag}}");
    }

    #[test]
    fn parse_directives_handles_multiple_occurrences() {
        let s = "{{#diff a b}} mid {{#diff c d}}";
        let got = parse_directives(s);
        assert_eq!(got.len(), 2);
        assert_eq!(got[0].left, "a");
        assert_eq!(got[1].right, "d");
        assert_eq!(&s[got[0].span.clone()], "{{#diff a b}}");
        assert_eq!(&s[got[1].span.clone()], "{{#diff c d}}");
    }

    #[test]
    fn parse_directives_skips_escaped_form() {
        let s = "use \{{#diff a b}} verbatim";
        let got = parse_directives(s);
        assert!(
            got.is_empty(),
            "escaped directive should not parse; got {got:?}",
        );
    }

    #[test]
    fn parse_directives_tolerates_extra_whitespace_around_operands() {
        let s = "{{#diff   a    b   }}";
        let got = parse_directives(s);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].left, "a");
        assert_eq!(got[0].right, "b");
    }

    #[test]
    fn parse_directives_skips_malformed_arity() {
        for s in ["{{#diff only-one}}", "{{#diff a b c}}", "{{#diff}}"] {
            let got = parse_directives(s);
            assert!(
                got.is_empty(),
                "malformed directive `{s}` should not parse; got {got:?}",
            );
        }
    }
```

```
    #[test]
    fn parse_directives_accepts_arbitrary_operand_strings() {
        let s = "{{#diff live:src/foo.rs new-tag}}";
        let got = parse_directives(s);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].left, "live:src/foo.rs");
        assert_eq!(got[0].right, "new-tag");
    }

    use crate::manifest::{MANIFEST_VERSION, Manifest};
    use std::fs;
    use tempfile::TempDir;

    /// Build a tempdir book root with two frozen files plus an in-memory
    /// manifest pointing at them. `span` is unused by the resolver, so the
    /// returned directive carries an empty range.
    fn fixture(left_bytes: &[u8], right_bytes: &[u8]) -> (TempDir, Manifest,
DiffDirective) {
        let tmp = TempDir::new().expect("tempdir");
        let listings_dir = tmp.path().join("src").join("listings");
        fs::create_dir_all(&listings_dir).unwrap();
        fs::write(listings_dir.join("left-tag.txt"), left_bytes).unwrap();
        fs::write(listings_dir.join("right-tag.txt"), right_bytes).unwrap();

        let manifest = Manifest {
            version: MANIFEST_VERSION,
            listings: vec![
                crate::manifest::Listing {
                    tag: "left-tag".into(),
                    source: "../left.txt".into(),
                    frozen: "src/listings/left-tag.txt".into(),
                    sha256: "0".repeat(64),
                },
                crate::manifest::Listing {
                    tag: "right-tag".into(),
                    source: "../right.txt".into(),
                    frozen: "src/listings/right-tag.txt".into(),
                    sha256: "0".repeat(64),
                },
            ],
        };

        let directive = DiffDirective {
            left: "left-tag".into(),
            right: "right-tag".into(),
            span: 0..0,
        };

        (tmp, manifest, directive)
    }

    #[test]
    fn resolve_returns_bytes_and_labels_for_known_tags() {
        let (tmp, manifest, directive) = fixture(b"line one\nline two\n", b"line
```

```
one\nline TWO\n");
        let resolved = resolve(&directive, &manifest,
tmp.path()).expect("resolve");
        assert_eq!(resolved.left_label, "left-tag");
        assert_eq!(resolved.right_label, "right-tag");
        assert_eq!(resolved.left_bytes, b"line one\nline two\n");
        assert_eq!(resolved.right_bytes, b"line one\nline TWO\n");
    }

    #[test]
    fn resolve_returns_unknown_tag_error_for_missing_left_operand() {
        let (tmp, manifest, mut directive) = fixture(b"a", b"b");
        directive.left = "nope".into();

        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
        assert_eq!(err.tag, "nope");
        assert!(matches!(err.kind, ResolveErrorKind::UnknownTag));
        let msg = format!("{err}");
        assert!(
            msg.contains("`nope`"),
            "diagnostic should name the missing tag; got: {msg}",
        );
    }

    #[test]
    fn resolve_returns_unknown_tag_error_for_missing_right_operand() {
        let (tmp, manifest, mut directive) = fixture(b"a", b"b");
        directive.right = "also-nope".into();

        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
        assert_eq!(err.tag, "also-nope");
        assert!(matches!(err.kind, ResolveErrorKind::UnknownTag));
    }

    #[test]
    fn resolve_returns_frozen_file_missing_when_disk_lacks_frozen_copy() {
        let (tmp, manifest, directive) = fixture(b"a", b"b");
        fs::remove_file(tmp.path().join("src/listings/left-tag.txt")).unwrap();

        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
        assert_eq!(err.tag, "left-tag");
        match &err.kind {
            ResolveErrorKind::FrozenFileMissing { frozen_path, .. } => {
                assert!(
                    frozen_path.ends_with("src/listings/left-tag.txt"),
                    "diagnostic should name the absent file; got
{frozen_path:?}",
                );
            }
            other => panic!("expected FrozenFileMissing; got {other:?}"),
        }
    }
```

```
    #[test]
    fn render_produces_unified_diff_with_headers_for_differing_inputs() {
        let out = render("line one\nline two\n", "line one\nline TWO\n", "old",
"new");
        assert!(out.contains("--- old"), "expected --- header; got:\n{out}");
        assert!(out.contains("+++ new"), "expected +++ header; got:\n{out}");
        assert!(
            out.contains("-line two"),
            "expected removed line; got:\n{out}"
        );
        assert!(
            out.contains("+line TWO"),
            "expected added line; got:\n{out}"
        );
    }

    #[test]
    fn render_returns_no_changes_notice_for_identical_inputs() {
        let out = render("same\nbytes\n", "same\nbytes\n", "old", "new");
        assert_eq!(out, "(no changes between old and new)\n");
    }

    #[test]
    fn render_returns_no_changes_notice_for_two_empty_inputs() {
        let out = render("", "", "old", "new");
        assert_eq!(out, "(no changes between old and new)\n");
    }

    #[test]
    fn render_marks_pure_additions_with_plus_prefix() {
        let out = render("a\n", "a\nb\n", "old", "new");
        assert!(out.contains("+b"), "expected added line `b`; got:\n{out}");
        assert!(
            !out.lines().any(|l| l.starts_with('-')
                && !l.starts_with("---")
                && l.trim_start_matches('-').contains("a")),
            "no removal expected on a pure addition; got:\n{out}",
        );
    }
}
```

The integration test from slice 1 is still `#[ignore]`'d. All three pure-unit pieces (parse, resolve, render) now exist in `diff.rs`; slice 5 wires them into `preprocess` and removes the `#[ignore]`.

Slice 5 — splicer + book registration + slice-1 test goes green

Slice 5 wires the three pure-unit pieces from slices 2–4 into the preprocessor and registers it in our own book so this very chapter starts rendering with diffs from this commit forward. The sub-section's three listings are the first in the book to be embedded as `{{#diff …}}` rather than full file contents.

`src/diff.rs` grows three things:

- `parse_escapes` — byte positions of `\` characters that

immediately precede an unescaped `{{#diff` substring; the splicer drops each one without touching the directive that follows.

- `SpliceError` — pairs the `ResolveError` from slice 3 with the

chapter source path and 1-based line number, so a missing-tag diagnostic reads `src/ch99-foo.md:5: no listing with tag \missing-tag`` rather than just naming the tag.

- `splice_chapter` — gathers directive and escape edits in one

pass, sorts by start offset, and stitches the output by copying the gaps verbatim. Edits anchor to byte spans of the *original* chapter text, so the splicer never has to think about offset shifts as it rewrites. The parser also gains code-fence awareness. Without it, registering the preprocessor in our own `book.toml` would break the build the moment a chapter quoted a frozen test fixture: the included `.rs` file's literal `{{#diff …}}` strings (with real-looking tag operands) would be parsed as real directives, and the resolver would fail to find those tags in our manifest. The parser now tracks ```` ```/~~~ ```` fences line-by-line and skips any directive whose start byte falls inside an open fence — the same rule that lets this very narrative quote `{{#diff …}}` syntax in fenced examples without the splicer eating them.

```
--- diff-v3
+++ diff-v4
@@ -17,27 +17,21 @@
 }

 /// Returns every well-formed `{{#diff a b}}` directive in `content`.
-/// Backslash-escaped directives (`{{#diff …}}`, matching mdbook's
-/// `{{#include}}` convention) and wrong-arity matches are skipped silently —
-/// the caller is in a better position to surface either with chapter-source
-/// context.
+/// Backslash-escaped directives, wrong-arity matches, and any directive
+/// whose start byte falls inside a fenced code block are skipped — the
+/// fence rule lets a chapter quote literal directive examples (e.g. a
+/// frozen test fixture) without the preprocessor consuming them.
 pub fn parse_directives(content: &str) -> Vec<DiffDirective> {
     const PREFIX: &[u8] = b"{{#diff";
     let bytes = content.as_bytes();
     let mut out = Vec::new();
-    let mut i = 0;
-    while i + PREFIX.len() <= bytes.len() {
-        if &bytes[i..i + PREFIX.len()] != PREFIX {
-            i += 1;
-            continue;
-        }
+    for_each_directive_position(content, |i| {
         if i > 0 && bytes[i - 1] == b'\\' {
-            i += PREFIX.len();
-            continue;
+            return PREFIX.len();
         }
         let inner_start = i + PREFIX.len();
         let Some(end_rel) = content[inner_start..].find("}}") else {
-            break;
+            return content.len() - i;
         };
         let directive_end = inner_start + end_rel + 2;
```

```
          let tokens: Vec<&str> = content[inner_start..inner_start + end_rel]
@@ -50,8 +44,8 @@
                  span: i..directive_end,
              });
          }
-         i = directive_end;
-     }
+         directive_end - i
+     });
      out
  }

@@ -163,6 +157,141 @@
          .to_string()
  }

+/// Byte positions of `\` characters that immediately precede a `{{#diff`
+/// substring outside fenced code blocks. The splicer drops these so the
+/// literal directive renders to the reader.
+pub fn parse_escapes(content: &str) -> Vec<usize> {
+    const PREFIX: &[u8] = b"{{#diff";
+    let bytes = content.as_bytes();
+    let mut out = Vec::new();
+    for_each_directive_position(content, |i| {
+        if i > 0 && bytes[i - 1] == b'\\' {
+            out.push(i - 1);
+        }
+        PREFIX.len()
+    });
+    out
+}
+
+/// Walks `content` byte-wise, skipping fenced code blocks, and invokes
+/// `visit(i)` at every byte offset `i` where `{{#diff` starts. The closure
+/// returns how many bytes to advance past the match — letting callers
+/// consume the whole directive (or just the prefix) without re-scanning.
+fn for_each_directive_position<F>(content: &str, mut visit: F)
+where
+    F: FnMut(usize) -> usize,
+{
+    const PREFIX: &[u8] = b"{{#diff";
+    let bytes = content.as_bytes();
+    let mut in_fence = false;
+    let mut line_start = 0;
+    while line_start < bytes.len() {
+        let line_end = match content[line_start..].find('\n') {
+            Some(off) => line_start + off,
+            None => bytes.len(),
+        };
+        if line_is_code_fence(&bytes[line_start..line_end]) {
+            in_fence = !in_fence;
+        } else if !in_fence {
+            let mut i = line_start;
+            while i + PREFIX.len() <= line_end {
+                if &bytes[i..i + PREFIX.len()] == PREFIX {
```

```
+                    let advance = visit(i);
+                    i += advance.max(1);
+                } else {
+                    i += 1;
+                }
+            }
+        }
+        line_start = line_end + 1;
+    }
+}
+
+fn line_is_code_fence(line: &[u8]) -> bool {
+    let leading_spaces = line.iter().take_while(|&&b| b == b' ').count();
+    if leading_spaces > 3 {
+        return false;
+    }
+    let rest = &line[leading_spaces..];
+    rest.starts_with(b"```") || rest.starts_with(b"~~~")
+}
+
+/// Failure shape carrying enough chapter context to point an author straight
+/// at the offending directive.
+#[derive(Debug)]
+pub struct SpliceError {
+    pub chapter_path: Option<PathBuf>,
+    pub line: usize,
+    pub source: ResolveError,
+}
+
+impl std::fmt::Display for SpliceError {
+    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
+        match &self.chapter_path {
+            Some(p) => write!(f, "{}:{}: {}", p.display(), self.line,
self.source),
+            None => write!(f, "<unknown chapter>:{}: {}", self.line,
self.source),
+        }
+    }
+}
+
+impl std::error::Error for SpliceError {
+    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
+        Some(&self.source)
+    }
+}
+
+/// Replace every `{{#diff …}}` directive in `content` with a fenced ` ```diff
`
+/// block of unified-diff text and strip the leading `\` from any
+/// `{{#diff …}}` escape so the literal directive renders to the reader.
+/// Bytes outside those spans are copied through unchanged.
+pub fn splice_chapter(
+    content: &str,
+    manifest: &Manifest,
+    book_root: &Path,
```

```
+    chapter_path: Option<&Path>,
+) -> Result<String, SpliceError> {
+    let directives = parse_directives(content);
+    let escapes = parse_escapes(content);
+
+    let mut edits: Vec<(usize, usize, String)> =
+        Vec::with_capacity(directives.len() + escapes.len());
+
+    for d in directives {
+        let resolved = resolve(&d, manifest, book_root).map_err(|source|
SpliceError {
+            chapter_path: chapter_path.map(Path::to_path_buf),
+            line: line_number(content, d.span.start),
+            source,
+        })?;
+        let left = String::from_utf8_lossy(&resolved.left_bytes);
+        let right = String::from_utf8_lossy(&resolved.right_bytes);
+        let body = render(&left, &right, &resolved.left_label,
&resolved.right_label);
+        edits.push((d.span.start, d.span.end, format!("```diff\n{body}```")));
+    }
+    for pos in escapes {
+        edits.push((pos, pos + 1, String::new()));
+    }
+
+    edits.sort_by_key(|(start, _, _)| *start);
+
+    let mut out = String::with_capacity(content.len());
+    let mut cursor = 0;
+    for (start, end, replacement) in edits {
+        out.push_str(&content[cursor..start]);
+        out.push_str(&replacement);
+        cursor = end;
+    }
+    out.push_str(&content[cursor..]);
+    Ok(out)
+}
+
+fn line_number(content: &str, byte_offset: usize) -> usize {
+    content[..byte_offset]
+        .bytes()
+        .filter(|&b| b == b'\n')
+        .count()
+        + 1
+}
+
 #[cfg(test)]
 mod tests {
     use super::*;
@@ -227,6 +356,28 @@
         assert_eq!(got[0].right, "new-tag");
     }

+    #[test]
+    fn parse_directives_skips_directives_inside_fenced_code_blocks() {
```

```
+        let s = "outside {{#diff a b}}\n\n```rust\nlet s = \"{{#diff inner-a
inner-b}}\";\n```\n\nmore {{#diff c d}}\n";
+        let got = parse_directives(s);
+        assert_eq!(got.len(), 2, "fenced one should be skipped; got {got:?}");
+        assert_eq!(got[0].left, "a");
+        assert_eq!(got[1].left, "c");
+    }
+
+    #[test]
+    fn parse_directives_handles_tilde_fences() {
+        let s = "~~~\n{{#diff a b}}\n~~~\n";
+        assert!(parse_directives(s).is_empty());
+    }
+
+    #[test]
+    fn parse_escapes_skips_inside_fenced_code_blocks() {
+        let s = "outside \{{#diff a b}}\n\n```\nlet s = \"\\{{#diff x y}}\";
\n```\n";
+        let escapes = parse_escapes(s);
+        assert_eq!(escapes.len(), 1, "fenced escape should be skipped");
+    }
+
     use crate::manifest::{MANIFEST_VERSION, Manifest};
     use std::fs;
     use tempfile::TempDir;
@@ -359,4 +510,70 @@
             "no removal expected on a pure addition; got:\n{out}",
         );
     }
+
+    #[test]
+    fn parse_escapes_returns_positions_of_backslashes_before_diff_directives()
{
+        let s = "use \{{#diff a b}} verbatim and \\{{#diff c d}} again";
+        let escapes = parse_escapes(s);
+        assert_eq!(escapes.len(), 2);
+        assert_eq!(&s[escapes[0]..=escapes[0]], "\\");
+        assert_eq!(&s[escapes[1]..=escapes[1]], "\\");
+    }
+
+    #[test]
+    fn parse_escapes_ignores_unescaped_directives() {
+        let s = "{{#diff a b}}";
+        assert!(parse_escapes(s).is_empty());
+    }
+
+    #[test]
+    fn
splice_chapter_replaces_directive_with_diff_fence_and_preserves_surroundings() {
+        let (tmp, manifest, _) = fixture(b"line one\nline two\n", b"line
one\nline TWO\n");
+        let chapter_path = Path::new("ch99.md");
+        let content = "Before paragraph.\n\n{{#diff left-tag right-
tag}}\n\nAfter paragraph.\n";
+        let out = splice_chapter(content, &manifest, tmp.path(),
```

```
Some(chapter_path)).unwrap();
+
+        assert!(out.starts_with("Before paragraph.\n"), "got:\n{out}");
+        assert!(out.ends_with("After paragraph.\n"), "got:\n{out}");
+        assert!(
+            out.contains("```diff\n"),
+            "expected diff fence; got:\n{out}"
+        );
+        assert!(
+            out.contains("--- left-tag"),
+            "expected left header; got:\n{out}"
+        );
+        assert!(
+            out.contains("+++ right-tag"),
+            "expected right header; got:\n{out}"
+        );
+        assert!(
+            !out.contains("{{#diff"),
+            "directive should be consumed; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_strips_leading_backslash_from_escaped_directives() {
+        let (tmp, manifest, _) = fixture(b"a", b"b");
+        let content = "Use \{{#diff a b}} to render a diff.\n";
+        let out = splice_chapter(content, &manifest, tmp.path(),
None).unwrap();
+        assert_eq!(out, "Use {{#diff a b}} to render a diff.\n");
+    }
+
+    #[test]
+    fn
splice_chapter_short_circuits_with_chapter_path_and_line_for_unknown_tag() {
+        let (tmp, manifest, _) = fixture(b"a", b"b");
+        let chapter_path = Path::new("src/ch99-foo.md");
+        let content = "intro\n\nmore\n\n{{#diff missing-tag right-tag}}\n";
+        let err =
+            splice_chapter(content, &manifest, tmp.path(),
Some(chapter_path)).expect_err("err");
+        assert_eq!(err.line, 5, "directive sits on line 5; got: {err}");
+        assert_eq!(err.chapter_path.as_deref(), Some(chapter_path));
+        let msg = format!("{err}");
+        assert!(
+            msg.contains("src/ch99-foo.md:5") && msg.contains("`missing-tag`"),
+            "diagnostic should name file:line and tag; got: {msg}",
+        );
+    }
 }
```

`src/main.rs`'s `preprocess` function goes from a no-op pass-through to the actual transformation: load the manifest from `<ctx.root>/listings.toml`, walk every `BookItem::Chapter` via `book.for_each_mut`, hand the chapter content to `splice_chapter`, and write the mutated book

back to stdout. `for_each_mut` doesn't let the closure return errors, so the splicer's failures are captured into a local `Option<anyhow::Error>` checked after the walk.

```
--- main-v3
+++ main-v4
@@ -1,10 +1,13 @@
 use std::path::PathBuf;
 use std::process;

-use anyhow::Result;
+use anyhow::{Context, Result};
 use clap::{Parser, Subcommand};
+use mdbook_listings::diff::splice_chapter;
 use mdbook_listings::freeze::{FreezeOptions, FreezeOutcome, freeze};
 use mdbook_listings::install::{InstallOutcome, install};
+use mdbook_listings::manifest::Manifest;
+use mdbook_preprocessor::book::BookItem;

 /// Managed code listings for mdbook: inline callouts, freezing, and
verification.
 #[derive(Parser)]
@@ -117,12 +120,38 @@
     }
 }

-/// Default mode: read an mdbook preprocessor JSON payload from stdin, emit the
-/// transformed payload on stdout.
+/// Default mode: read an mdbook preprocessor JSON payload from stdin, splice
+/// rendered diffs into every `{{#diff …}}` directive, emit the transformed
+/// payload on stdout.
 fn preprocess() -> Result<()> {
-    let (_ctx, book) = mdbook_preprocessor::parse_input(std::io::stdin())?;
-    serde_json::to_writer(std::io::stdout(), &book)?;
-    Ok(())
+    let (ctx, mut book) = mdbook_preprocessor::parse_input(std::io::stdin())?;
+    let manifest = Manifest::load(&ctx.root)?;
+
+    let mut splice_err: Option<anyhow::Error> = None;
+    book.for_each_mut(|item| {
+        if splice_err.is_some() {
+            return;
+        }
+        if let BookItem::Chapter(chapter) = item {
+            match splice_chapter(
+                &chapter.content,
+                &manifest,
+                &ctx.root,
+                chapter.source_path.as_deref(),
+            ) {
+                Ok(new_content) => chapter.content = new_content,
+                Err(e) => {
+                    splice_err =
+                        Some(anyhow::Error::new(e).context("rendering {{#diff}}
directive failed"));
+                }
```

```
+            }
+        }
+    });
+    if let Some(e) = splice_err {
+        return Err(e);
+    }
+
+    serde_json::to_writer(std::io::stdout(), &book).context("writing
transformed book to stdout")
 }

 /// Answer mdbook's renderer-support probe by exiting 0 (supported) or 1
```

`tests/diffs.rs` drops the `#[ignore]` on the slice-1 acceptance test (the splicer makes it pass) and gains two more integration tests pinning the surrounding-content invariance and the backslash-escape behaviour at the binary boundary. The fixture is rebuilt to mirror a real mdbook book root: `listings.toml` at the tempdir top, frozen files under `src/listings/`, matching what `Manifest::load(&ctx.root)` actually reads. The slice-1 fixture put those under a redundant `book/` subdirectory, which worked while the preprocessor was a pass-through but doesn't now.

```
--- diffs-tests-v1
+++ diffs-tests-v2
@@ -1,7 +1,7 @@
 //! Integration tests for the Show Diffs Between Slices story (ch. 3).

 use std::fs;
-use std::path::{Path, PathBuf};
+use std::path::PathBuf;

 use mdbook_preprocessor::PreprocessorContext;
 use mdbook_preprocessor::book::{Book, BookItem, Chapter};
@@ -12,13 +12,71 @@
 use common::mdbook_listings;

 #[test]
-#[ignore = "directive parsing + diff rendering land in slices 2-5"]
 fn diff_directive_renders_to_fenced_diff_block() {
     let book = MinimalDiffsBook::new();
     let envelope = book.envelope_with_chapter(
         "Before paragraph.\n\n{{#diff old-tag new-tag}}\n\nAfter paragraph.\n",
     );

+    let returned = run_preprocessor(envelope);
+    let content = chapter_content(&returned, "Diff Test");
+
+    assert!(
+        content.contains("```diff"),
+        "expected the directive to render as a ```diff fenced block; got:
\n{content}",
+    );
+    assert!(
+        content.contains("--- old-tag") && content.contains("+++ new-tag"),
```

```
+        "expected unified-diff headers naming the operands; got:\n{content}",
+    );
+    assert!(
+        content.contains("-line two") && content.contains("+line TWO"),
+        "expected the +/- lines from the frozen pair; got:\n{content}",
+    );
+}
+
+#[test]
+fn diff_directive_does_not_disturb_surrounding_chapter_content() {
+    let book = MinimalDiffsBook::new();
+    let envelope = book.envelope_with_chapter(
+        "Before paragraph.\n\n{{#diff old-tag new-tag}}\n\nAfter paragraph.\n",
+    );
+
+    let returned = run_preprocessor(envelope);
+    let content = chapter_content(&returned, "Diff Test");
+
+    assert!(
+        content.starts_with("Before paragraph.\n"),
+        "leading text should survive verbatim; got:\n{content}",
+    );
+    assert!(
+        content.ends_with("After paragraph.\n"),
+        "trailing text should survive verbatim; got:\n{content}",
+    );
+    assert!(
+        !content.contains("{{#diff"),
+        "directive should be consumed; got:\n{content}",
+    );
+}
+
+#[test]
+fn escaped_diff_directive_is_left_literal_minus_the_backslash() {
+    let book = MinimalDiffsBook::new();
+    let envelope =
+        book.envelope_with_chapter("Use \{{#diff old-tag new-tag}} verbatim in
prose.\n");
+
+    let returned = run_preprocessor(envelope);
+    let content = chapter_content(&returned, "Diff Test");
+
+    assert_eq!(
+        content,
+        "Use {{#diff old-tag new-tag}} verbatim in prose.\n"
+    );
+}
+
+/// Pipes the envelope through the preprocessor binary and returns the
+/// transformed `Book` parsed from stdout.
+fn run_preprocessor(envelope: String) -> Book {
     let output = mdbook_listings()
         .write_stdin(envelope)
         .assert()
@@ -26,20 +84,12 @@
```

```
         .get_output()
         .stdout
         .clone();
-
-    let returned: Book = serde_json::from_slice(&output).expect("parse stdout
as Book");
-    let content = chapter_content(&returned, "Diff Test");
-
-    assert!(
-        content.contains("```diff"),
-        "expected the directive to render as a ```diff fenced block; got:
\n{content}",
-    );
+    serde_json::from_slice(&output).expect("parse stdout as Book")
 }

-/// The smallest tempdir fixture that backs the diff preprocessor: a
`book.toml`
-/// that registers `[preprocessor.listings]`, a `book/listings.toml` with two
-/// frozen entries the chapters can reference by tag, and the matching frozen
-/// files. The tempdir is destroyed when the struct drops.
+/// Tempdir laid out as a real mdbook book root: `listings.toml` at the top
+/// plus the two frozen files under `src/listings/` that the integration
+/// chapters reference by tag.
 struct MinimalDiffsBook {
     _tmp: TempDir,
     root: PathBuf,
@@ -50,19 +100,13 @@
         let tmp = TempDir::new().expect("tempdir");
         let root = tmp.path().to_path_buf();

-        fs::write(
-            root.join("book.toml"),
-            "[book]\ntitle = \"Test\"\n\n[preprocessor.listings]\ncommand =
\"mdbook-listings\"\n",
-        )
-        .unwrap();
-
-        let listings_dir = root.join("book").join("src").join("listings");
+        let listings_dir = root.join("src").join("listings");
         fs::create_dir_all(&listings_dir).unwrap();
         fs::write(listings_dir.join("old-tag.txt"), "line one\nline
two\n").unwrap();
         fs::write(listings_dir.join("new-tag.txt"), "line one\nline
TWO\n").unwrap();

         fs::write(
-            root.join("book").join("listings.toml"),
+            root.join("listings.toml"),
             "version = 1\n\n\
              [[listing]]\n\
              tag = \"old-tag\"\n\
@@ -80,14 +124,8 @@
         Self { _tmp: tmp, root }
     }
```

```
-    #[allow(dead_code)]
-    fn root(&self) -> &Path {
-        &self.root
-    }
-
-    /// Build the JSON envelope mdbook would send a preprocessor: the tuple
-    /// `(PreprocessorContext, Book)` serialised as a two-element JSON array,
-    /// with one chapter carrying the supplied markdown.
+    /// Build the `[PreprocessorContext, Book]` JSON tuple mdbook would send,
+    /// with one chapter carrying `chapter_content`.
     fn envelope_with_chapter(&self, chapter_content: &str) -> String {
         let ctx =
             PreprocessorContext::new(self.root.clone(), Config::default(),
"html".to_string());
```

`book/book.toml` gains `[preprocessor.listings]` (with `before = ["admonish"]` because admonish is registered too) and `[output.html].additional-css` picks up `mdbook-listings.css`. The edit was made by running `cargo run -- install --book-root book` — the install handler from ch. 1 is idempotent, so re-running it in future builds is harmless.

The integration suite is fully green: 53 tests pass, 0 skipped. The diff primitive is end-to-end functional and the book itself exercises it.

The `parse_escapes` helper, the escape-handling branch in `splice_chapter`, and the `escaped_diff_directive_is_left_literal_minus_the_backslash` integration test are dead code in the real pipeline (mdbook's `links` preprocessor strips the leading `\` before our binary ever runs — see the AC 6 note above). They earn their keep only when our binary is driven directly via stdin, which isn't a supported use case. The refactor slice removes them and re-freezes the affected files; until then they document the fact-of-life by their visible presence.

Slice 6 — `live:<path>` operand

Slice 6 closes out the initial AC 7. `resolve_operand` now recognises the `live:` prefix and reads the named file from disk (slice 6 resolved against `book_root`; slice 8 changed this to the chapter's source directory, matching `{{#include}}` semantics). The operand's full text (including the `live:` prefix) becomes the unified-diff header label, so a reader can tell at a glance which side is frozen and which side is live.

A new `ResolveErrorKind::LiveFileMissing` variant carries the absolute path that failed to read so the splicer's chapter-located diagnostic stays specific. Two unit tests cover the happy path and the missing-file error; one new integration test in `tests/diffs.rs` drives a `{{#diff …}}` whose right operand is `live:compose-live.yaml` end-to-end through the binary and asserts on the `+++ live:…` header and the `+/-` lines reflecting the live bytes. The `MinimalDiffsBook` fixture grows a `write_live_file` helper for the same.

```
--- diff-v4
+++ diff-v5
@@ -74,6 +74,10 @@
         frozen_path: PathBuf,
         source: std::io::Error,
     },
+    LiveFileMissing {
```

```
+        live_path: PathBuf,
+        source: std::io::Error,
+    },
 }

 impl std::fmt::Display for ResolveError {
@@ -92,6 +96,13 @@
                 frozen_path.display(),
                 source,
             ),
+            ResolveErrorKind::LiveFileMissing { live_path, source } => write!(
+                f,
+                "live operand `{}` cannot be read at `{}`: {}",
+                self.tag,
+                live_path.display(),
+                source,
+            ),
         }
     }
 }
@@ -101,6 +112,7 @@
         match &self.kind {
             ResolveErrorKind::UnknownTag => None,
             ResolveErrorKind::FrozenFileMissing { source, .. } => Some(source),
+            ResolveErrorKind::LiveFileMissing { source, .. } => Some(source),
         }
     }
 }
@@ -128,6 +140,17 @@
     manifest: &Manifest,
     book_root: &Path,
 ) -> Result<(String, Vec<u8>), ResolveError> {
+    if let Some(rel_path) = operand.strip_prefix("live:") {
+        let live_path = book_root.join(rel_path);
+        let bytes = std::fs::read(&live_path).map_err(|source| ResolveError {
+            tag: operand.to_string(),
+            kind: ResolveErrorKind::LiveFileMissing {
+                live_path: live_path.clone(),
+                source,
+            },
+        })?;
+        return Ok((operand.to_string(), bytes));
+    }
     let listing = manifest.find(operand).ok_or_else(|| ResolveError {
         tag: operand.to_string(),
         kind: ResolveErrorKind::UnknownTag,
@@ -455,6 +478,35 @@
     }

     #[test]
+    fn resolve_returns_bytes_for_live_operand() {
+        let (tmp, manifest, mut directive) = fixture(b"a", b"b");
+        fs::write(tmp.path().join("live-source.txt"), "live one\nlive
two\n").unwrap();
+        directive.left = "live:live-source.txt".into();
```

```
+
+        let resolved = resolve(&directive, &manifest,
tmp.path()).expect("resolve");
+        assert_eq!(resolved.left_label, "live:live-source.txt");
+        assert_eq!(resolved.left_bytes, b"live one\nlive two\n");
+    }
+
+    #[test]
+    fn resolve_returns_live_file_missing_when_disk_lacks_live_path() {
+        let (tmp, manifest, mut directive) = fixture(b"a", b"b");
+        directive.left = "live:nope.txt".into();
+
+        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
+        assert_eq!(err.tag, "live:nope.txt");
+        match &err.kind {
+            ResolveErrorKind::LiveFileMissing { live_path, .. } => {
+                assert!(
+                    live_path.ends_with("nope.txt"),
+                    "diagnostic should name the absent file; got
{live_path:?}",
+                );
+            }
+            other => panic!("expected LiveFileMissing; got {other:?}"),
+        }
+    }
+
+    #[test]
     fn resolve_returns_frozen_file_missing_when_disk_lacks_frozen_copy() {
         let (tmp, manifest, directive) = fixture(b"a", b"b");
         fs::remove_file(tmp.path().join("src/listings/left-tag.txt")).unwrap();
```

```
--- diffs-tests-v2
+++ diffs-tests-v3
@@ -60,6 +60,28 @@
 }

 #[test]
+fn live_path_operand_diffs_against_disk_relative_to_book_root() {
+    let book = MinimalDiffsBook::new();
+    book.write_live_file("compose-live.yaml", b"line one\nline LIVE\n");
+
+    let envelope = book.envelope_with_chapter(
+        "Diffing live source.\n\n{{#diff old-tag live:compose-live.yaml}}\n",
+    );
+
+    let returned = run_preprocessor(envelope);
+    let content = chapter_content(&returned, "Diff Test");
+
+    assert!(
+        content.contains("--- old-tag") && content.contains("+++ live:compose-
live.yaml"),
+        "expected headers naming the frozen tag and the live operand; got:
\n{content}",
```

```
+    );
+    assert!(
+        content.contains("-line two") && content.contains("+line LIVE"),
+        "expected +/- lines reflecting the live source; got:\n{content}",
+    );
+}
+
+#[test]
 fn escaped_diff_directive_is_left_literal_minus_the_backslash() {
     let book = MinimalDiffsBook::new();
     let envelope =
@@ -124,6 +146,16 @@
         Self { _tmp: tmp, root }
     }

+    /// Write a file at `rel` (relative to the book root) so a chapter can
+    /// reference it via a `live:<rel>` operand.
+    fn write_live_file(&self, rel: &str, bytes: &[u8]) {
+        let abs = self.root.join(rel);
+        if let Some(parent) = abs.parent() {
+            fs::create_dir_all(parent).unwrap();
+        }
+        fs::write(&abs, bytes).unwrap();
+    }
+
     /// Build the `[PreprocessorContext, Book]` JSON tuple mdbook would send,
     /// with one chapter carrying `chapter_content`.
     fn envelope_with_chapter(&self, chapter_content: &str) -> String {
```

To dogfood it, here is the chapter rendering a `live:` diff between the `diff-v5` tag (frozen above) and the live `src/diff.rs` on disk at build time. The path is relative to the chapter's own source directory (`book/src/`, post-slice-8), so `../../src/diff.rs` walks up two levels to the repo root and back into the crate's `src/`:

```
--- diff-v5
+++ live:../../src/diff.rs
@@ -9,13 +9,103 @@

 /// `span` covers the directive in full (`{{#diff …}}` inclusive) so callers
 /// can replace the whole substring in one pass.
+///
+/// `left_range` and `right_range` are present when the directive carries
+/// optional 3rd and 4th `START:END` arguments — `{{#diff a b 1:50 1:60}}`
+/// renders only those slices of each operand. Empty endpoints mean "to
+/// start" or "to end". Two ranges (one per operand) because line numbers
+/// shift between versions.
 #[derive(Debug, Clone, PartialEq, Eq)]
 pub struct DiffDirective {
     pub left: String,
     pub right: String,
+    pub left_range: Option<LineRange>,
+    pub right_range: Option<LineRange>,
     pub span: Range<usize>,
```

```
 }

+/// 1-based inclusive line range. `None` endpoints mean "to start"
+/// (`start`) or "to end" (`end`). Out-of-range endpoints are clamped to
+/// the file's actual line count silently.
+#[derive(Debug, Clone, Copy, PartialEq, Eq)]
+pub struct LineRange {
+    pub start: Option<usize>,
+    pub end: Option<usize>,
+}
+
+impl LineRange {
+    /// Render as the `START:END` form used in directives and anchors.
+    /// Empty endpoints render as the empty string.
+    pub fn render(&self) -> String {
+        let s = self.start.map(|n| n.to_string()).unwrap_or_default();
+        let e = self.end.map(|n| n.to_string()).unwrap_or_default();
+        format!("{s}:{e}")
+    }
+
+    /// Slice `text` to the range's line span, 1-based inclusive. Returns
+    /// the substring that includes lines `[start..=end]` (clamped). The
+    /// returned substring preserves trailing newlines.
+    pub fn slice<'a>(&self, text: &'a str) -> &'a str {
+        let total = text.lines().count();
+        let start_1 = self.start.unwrap_or(1).max(1);
+        let end_1 = self.end.unwrap_or(total).min(total);
+        if start_1 > end_1 || total == 0 {
+            return "";
+        }
+        // Find byte offsets for line `start_1` and `end_1 + 1` (or EOF).
+        let mut byte_start = 0usize;
+        let mut current_line = 1usize;
+        let bytes = text.as_bytes();
+        while current_line < start_1 && byte_start < bytes.len() {
+            match text[byte_start..].find('\n') {
+                Some(off) => byte_start += off + 1,
+                None => return "",
+            }
+            current_line += 1;
+        }
+        let mut byte_end = byte_start;
+        let mut line = current_line;
+        while line <= end_1 && byte_end < bytes.len() {
+            match text[byte_end..].find('\n') {
+                Some(off) => byte_end += off + 1,
+                None => {
+                    byte_end = bytes.len();
+                    break;
+                }
+            }
+            line += 1;
+        }
+        &text[byte_start..byte_end]
+    }
```

```
+}
+
+/// Parses one `START:END` token. Returns `None` when the form is malformed
+/// (the directive is then skipped just like a wrong-arity `{{#diff}}`).
+/// Empty endpoints are allowed; `:` (both empty) means whole file. Numeric
+/// endpoints must be positive (zero rejected).
+pub fn parse_line_range(tok: &str) -> Option<LineRange> {
+    let (s, e) = tok.split_once(':')?;
+    let start = if s.is_empty() {
+        None
+    } else {
+        let n: usize = s.parse().ok()?;
+        if n == 0 {
+            return None;
+        }
+        Some(n)
+    };
+    let end = if e.is_empty() {
+        None
+    } else {
+        let n: usize = e.parse().ok()?;
+        if n == 0 {
+            return None;
+        }
+        Some(n)
+    };
+    Some(LineRange { start, end })
+}
+
 /// Returns every well-formed `{{#diff a b}}` directive in `content`.
 /// Backslash-escaped directives, wrong-arity matches, and any directive
 /// whose start byte falls inside a fenced code block are skipped — the
@@ -25,27 +115,66 @@
     const PREFIX: &[u8] = b"{{#diff";
     let bytes = content.as_bytes();
     let mut out = Vec::new();
-    for_each_directive_position(content, |i| {
-        if i > 0 && bytes[i - 1] == b'\\' {
-            return PREFIX.len();
-        }
-        let inner_start = i + PREFIX.len();
-        let Some(end_rel) = content[inner_start..].find("}}") else {
-            return content.len() - i;
+    let mut in_fence = false;
+    let mut line_start = 0;
+    while line_start < bytes.len() {
+        let line_end = match content[line_start..].find('\n') {
+            Some(off) => line_start + off,
+            None => bytes.len(),
         };
-        let directive_end = inner_start + end_rel + 2;
-        let tokens: Vec<&str> = content[inner_start..inner_start + end_rel]
-            .split_whitespace()
-            .collect();
-        if tokens.len() == 2 {
```

```
-            out.push(DiffDirective {
-                left: tokens[0].to_string(),
-                right: tokens[1].to_string(),
-                span: i..directive_end,
-            });
+        if line_is_code_fence(&bytes[line_start..line_end]) {
+            in_fence = !in_fence;
+        } else if !in_fence {
+            let mut i = line_start;
+            while i + PREFIX.len() ⩽ line_end {
+                if &bytes[i..i + PREFIX.len()] ≠ PREFIX {
+                    i += 1;
+                    continue;
+                }
+                if i > 0 && bytes[i - 1] == b'\\' {
+                    i += PREFIX.len();
+                    continue;
+                }
+                let backticks_before = bytes[line_start..i].iter().filter(|&&b|
b == b'`').count();
+                if backticks_before % 2 == 1 {
+                    i += PREFIX.len();
+                    continue;
+                }
+                let inner_start = i + PREFIX.len();
+                let Some(end_rel) = content[inner_start..].find("}}") else {
+                    break;
+                };
+                let directive_end = inner_start + end_rel + 2;
+                let tokens: Vec<&str> = content[inner_start..inner_start +
end_rel]
+                    .split_whitespace()
+                    .collect();
+                let parsed = match tokens.as_slice() {
+                    [l, r] ⇒ Some((l.to_string(), r.to_string(), None, None)),
+                    [l, r, lr, rr] ⇒ match (parse_line_range(lr),
parse_line_range(rr)) {
+                        (Some(left_range), Some(right_range)) ⇒ Some((
+                            l.to_string(),
+                            r.to_string(),
+                            Some(left_range),
+                            Some(right_range),
+                        )),
+                        _ ⇒ None,
+                    },
+                    _ ⇒ None,
+                };
+                if let Some((left, right, left_range, right_range)) = parsed {
+                    out.push(DiffDirective {
+                        left,
+                        right,
+                        left_range,
+                        right_range,
+                        span: i..directive_end,
+                    });
```

```
+                }
+                i = directive_end;
+            }
         }
-        directive_end - i
-    });
+        line_start = line_end + 1;
+    }
     out
 }

@@ -119,14 +248,20 @@

 /// Returns at the first failing operand so callers surface one missing tag
 /// at a time — the second tag's resolution can wait for the rebuild after
-/// the first fix.
+/// the first fix. `live_base` is the absolute directory `live:<rel_path>`
+/// operands resolve against; the splicer passes the chapter's source
+/// directory so authors can reference siblings the same way they would for
+/// `{{#include}}`.
 pub fn resolve(
     directive: &DiffDirective,
     manifest: &Manifest,
     book_root: &Path,
+    live_base: &Path,
 ) -> Result<ResolvedDiff, ResolveError> {
-    let (left_label, left_bytes) = resolve_operand(&directive.left, manifest,
book_root)?;
-    let (right_label, right_bytes) = resolve_operand(&directive.right,
manifest, book_root)?;
+    let (left_label, left_bytes) =
+        resolve_operand(&directive.left, manifest, book_root, live_base)?;
+    let (right_label, right_bytes) =
+        resolve_operand(&directive.right, manifest, book_root, live_base)?;
     Ok(ResolvedDiff {
         left_label,
         left_bytes,
@@ -139,9 +274,10 @@
     operand: &str,
     manifest: &Manifest,
     book_root: &Path,
+    live_base: &Path,
 ) -> Result<(String, Vec<u8>), ResolveError> {
     if let Some(rel_path) = operand.strip_prefix("live:") {
-        let live_path = book_root.join(rel_path);
+        let live_path = live_base.join(rel_path);
         let bytes = std::fs::read(&live_path).map_err(|source| ResolveError {
             tag: operand.to_string(),
             kind: ResolveErrorKind::LiveFileMissing {
@@ -180,56 +316,74 @@
         .to_string()
 }

-/// Byte positions of `\` characters that immediately precede a `{{#diff`
-/// substring outside fenced code blocks. The splicer drops these so the
```

```
-/// literal directive renders to the reader.
-pub fn parse_escapes(content: &str) -> Vec<usize> {
-    const PREFIX: &[u8] = b"{{#diff";
-    let bytes = content.as_bytes();
-    let mut out = Vec::new();
-    for_each_directive_position(content, |i| {
-        if i > 0 && bytes[i - 1] == b'\\' {
-            out.push(i - 1);
+/// Shift the line numbers in every `@@ -A,B +C,D @@` hunk header by
+/// `left_offset` and `right_offset` respectively. Used when a sliced
+/// `{{#diff a b LR LR}}` directive feeds only a fragment of each
+/// listing to `similar` — without the shift the rendered hunk headers
+/// would be relative to the slice (`@@ -3,18 +3,28 @@`) rather than the
+/// absolute line numbers in the original files (`@@ -58,18 +148,28 @@`),
+/// and readers would have no way to map a `+` line in the rendered diff
+/// back to its position in the parent listing.
+///
+/// Hunk headers are the only diff syntax that carries line numbers, so
+/// every other line passes through verbatim. Lines that look like `@@`
+/// headers but aren't well-formed are left alone.
+pub fn shift_hunk_headers(diff_text: &str, left_offset: usize, right_offset:
usize) -> String {
+    if left_offset == 0 && right_offset == 0 {
+        return diff_text.to_string();
+    }
+    let mut out = String::with_capacity(diff_text.len());
+    for line in diff_text.split_inclusive('\n') {
+        let trailing_newline = line.ends_with('\n');
+        let body = line.strip_suffix('\n').unwrap_or(line);
+        if let Some(shifted) = shift_one_hunk_header(body, left_offset,
right_offset) {
+            out.push_str(&shifted);
+            if trailing_newline {
+                out.push('\n');
+            }
+        } else {
+            out.push_str(line);
         }
-        PREFIX.len()
-    });
+    }
     out
 }

-/// Walks `content` byte-wise, skipping fenced code blocks, and invokes
-/// `visit(i)` at every byte offset `i` where `{{#diff` starts. The closure
-/// returns how many bytes to advance past the match — letting callers
-/// consume the whole directive (or just the prefix) without re-scanning.
-fn for_each_directive_position<F>(content: &str, mut visit: F)
-where
-    F: FnMut(usize) -> usize,
-{
-    const PREFIX: &[u8] = b"{{#diff";
-    let bytes = content.as_bytes();
-    let mut in_fence = false;
```

```
-    let mut line_start = 0;
-    while line_start < bytes.len() {
-        let line_end = match content[line_start..].find('\n') {
-            Some(off) => line_start + off,
-            None => bytes.len(),
-        };
-        if line_is_code_fence(&bytes[line_start..line_end]) {
-            in_fence = !in_fence;
-        } else if !in_fence {
-            let mut i = line_start;
-            while i + PREFIX.len() <= line_end {
-                if &bytes[i..i + PREFIX.len()] == PREFIX {
-                    let advance = visit(i);
-                    i += advance.max(1);
-                } else {
-                    i += 1;
-                }
-            }
-        }
-        line_start = line_end + 1;
+/// Returns `Some(shifted_line)` when `body` is a well-formed unified-diff
+/// `@@ -A[,B] +C[,D] @@[ context]` hunk header, with the line numbers
+/// shifted by the offsets. Returns `None` otherwise so the caller passes
+/// the line through verbatim.
+fn shift_one_hunk_header(body: &str, left_offset: usize, right_offset: usize) -
> Option<String> {
+    let rest = body.strip_prefix("@@ ")?;
+    // `rest` looks like "-A[,B] +C[,D] @@[ context]" — split off the
+    // closing "@@" and the optional context that follows it.
+    let (ranges, suffix) = rest.split_once(" @@")?;
+    let parts: Vec<&str> = ranges.split_whitespace().collect();
+    if parts.len() != 2 {
+        return None;
     }
+    let left = parts[0].strip_prefix('-')?;
+    let right = parts[1].strip_prefix('+')?;
+    let (l_start, l_count) = parse_hunk_range(left)?;
+    let (r_start, r_count) = parse_hunk_range(right)?;
+    Some(format!(
+        "@@ -{},{} +{},{} @@{}",
+        l_start + left_offset,
+        l_count,
+        r_start + right_offset,
+        r_count,
+        suffix,
+    ))
 }

+fn parse_hunk_range(s: &str) -> Option<(usize, usize)> {
+    if let Some((a, b)) = s.split_once(',') {
+        Some((a.parse().ok()?, b.parse().ok()?))
+    } else {
+        let n: usize = s.parse().ok()?;
+        Some((n, 1))
+    }
```

```
+}
+
 fn line_is_code_fence(line: &[u8]) -> bool {
     let leading_spaces = line.iter().take_while(|&&b| b == b' ').count();
     if leading_spaces > 3 {
@@ -264,44 +418,78 @@
 }

 /// Replace every `{{#diff …}}` directive in `content` with a fenced ` ```diff
`
-/// block of unified-diff text and strip the leading `\` from any
-/// `{{#diff …}}` escape so the literal directive renders to the reader.
-/// Bytes outside those spans are copied through unchanged.
+/// block of unified-diff text. Bytes outside those spans are copied through
+/// unchanged. `chapter_dir` is the absolute directory the chapter's source
+/// markdown lives in; `live:<rel>` operands resolve against it the same way
+/// mdbook's `{{#include <rel>}}` does.
 pub fn splice_chapter(
     content: &str,
     manifest: &Manifest,
     book_root: &Path,
     chapter_path: Option<&Path>,
+    chapter_dir: &Path,
 ) -> Result<String, SpliceError> {
-    let directives = parse_directives(content);
-    let escapes = parse_escapes(content);
-
-    let mut edits: Vec<(usize, usize, String)> =
-        Vec::with_capacity(directives.len() + escapes.len());
-
-    for d in directives {
-        let resolved = resolve(&d, manifest, book_root).map_err(|source|
SpliceError {
-            chapter_path: chapter_path.map(Path::to_path_buf),
-            line: line_number(content, d.span.start),
-            source,
-        })?;
-        let left = String::from_utf8_lossy(&resolved.left_bytes);
-        let right = String::from_utf8_lossy(&resolved.right_bytes);
-        let body = render(&left, &right, &resolved.left_label,
&resolved.right_label);
-        edits.push((d.span.start, d.span.end, format!("```diff\n{body}```")));
-    }
-    for pos in escapes {
-        edits.push((pos, pos + 1, String::new()));
-    }
-
-    edits.sort_by_key(|(start, _, _)| *start);
-
     let mut out = String::with_capacity(content.len());
     let mut cursor = 0;
-    for (start, end, replacement) in edits {
-        out.push_str(&content[cursor..start]);
-        out.push_str(&replacement);
-        cursor = end;
```

```
+    for d in parse_directives(content) {
+        let resolved =
+            resolve(&d, manifest, book_root, chapter_dir).map_err(|source|
SpliceError {
+                chapter_path: chapter_path.map(Path::to_path_buf),
+                line: line_number(content, d.span.start),
+                source,
+            })?;
+        let left_full = String::from_utf8_lossy(&resolved.left_bytes);
+        let right_full = String::from_utf8_lossy(&resolved.right_bytes);
+        let left_sliced: &str = match &d.left_range {
+            Some(r) => r.slice(&left_full),
+            None => &left_full,
+        };
+        let right_sliced: &str = match &d.right_range {
+            Some(r) => r.slice(&right_full),
+            None => &right_full,
+        };
+        let body = render(
+            left_sliced,
+            right_sliced,
+            &resolved.left_label,
+            &resolved.right_label,
+        );
+        // When a range is set, similar's hunk headers are relative to the
+        // slice (line 1 of the slice = line N of the original). Shift them
+        // back to absolute line numbers so readers can map a +/- line in
+        // the rendered diff to its real position in the parent listing.
+        let left_offset = d
+            .left_range
+            .and_then(|r| r.start)
+            .map(|n| n - 1)
+            .unwrap_or(0);
+        let right_offset = d
+            .right_range
+            .and_then(|r| r.start)
+            .map(|n| n - 1)
+            .unwrap_or(0);
+        let body = shift_hunk_headers(&body, left_offset, right_offset);
+        out.push_str(&content[cursor..d.span.start]);
+        out.push_str("```diff\n");
+        out.push_str(&body);
+        out.push_str("```\n");
+        // CALLOUT: diff-anchor-dual Locator anchor for the capture-screenshots
tool. Both operands are emitted as separate data-attributes so the tool can
locate a diff block by its (LEFT, RIGHT) pair — unique even when multiple diffs
share the same RIGHT tag, and unambiguous against the include splicer's `data-
listing-tag` anchors.
+        let mut anchor = format!(
+            "<div data-listing-diff-left=\"{}\" data-listing-diff-
right=\"{}\"",
+            d.left, d.right,
+        );
+        if let Some(r) = &d.left_range {
+            anchor.push_str(&format!(" data-listing-diff-left-range=\"{}\"",
```

```
r.render()));
+        }
+        if let Some(r) = &d.right_range {
+            anchor.push_str(&format!(
+                " data-listing-diff-right-range=\"{}\"",
+                r.render()
+            ));
+        }
+        anchor.push_str(" aria-hidden=\"true\"></div>");
+        out.push_str(&anchor);
+        cursor = d.span.end;
     }
     out.push_str(&content[cursor..]);
     Ok(out)
@@ -360,6 +548,226 @@
     }

     #[test]
+    fn parse_directives_accepts_optional_line_ranges() {
+        let s = "{{#diff a b 1:50 1:60}}";
+        let got = parse_directives(s);
+        assert_eq!(got.len(), 1, "got {got:?}");
+        assert_eq!(got[0].left, "a");
+        assert_eq!(got[0].right, "b");
+        assert_eq!(
+            got[0].left_range,
+            Some(LineRange {
+                start: Some(1),
+                end: Some(50)
+            })
+        );
+        assert_eq!(
+            got[0].right_range,
+            Some(LineRange {
+                start: Some(1),
+                end: Some(60)
+            })
+        );
+    }
+
+    #[test]
+    fn parse_directives_accepts_open_endpoints_in_ranges() {
+        let cases = [
+            (
+                "{{#diff a b 200: 220:}}",
+                LineRange {
+                    start: Some(200),
+                    end: None,
+                },
+                LineRange {
+                    start: Some(220),
+                    end: None,
+                },
+            ),
+            (
```

```
+                "{{#diff a b :100 :100}}",
+                LineRange {
+                    start: None,
+                    end: Some(100),
+                },
+                LineRange {
+                    start: None,
+                    end: Some(100),
+                },
+            ),
+            (
+                "{{#diff a b : :}}",
+                LineRange {
+                    start: None,
+                    end: None,
+                },
+                LineRange {
+                    start: None,
+                    end: None,
+                },
+            ),
+        ];
+        for (s, expected_l, expected_r) in cases {
+            let got = parse_directives(s);
+            assert_eq!(got.len(), 1, "input `{s}` → {got:?}");
+            assert_eq!(got[0].left_range, Some(expected_l), "input `{s}`");
+            assert_eq!(got[0].right_range, Some(expected_r), "input `{s}`");
+        }
+    }
+
+    #[test]
+    fn parse_directives_rejects_malformed_or_negative_range() {
+        for s in [
+            "{{#diff a b 1 1}}",       // no colon — not a range
+            "{{#diff a b 0:5 1:5}}",   // zero start rejected
+            "{{#diff a b 1:0 1:5}}",   // zero end rejected
+            "{{#diff a b 1:abc 1:5}}", // non-numeric
+            "{{#diff a b -1:5 1:5}}",  // negative
+            "{{#diff a b 1:5 1:5 x}}", // 5 args
+        ] {
+            let got = parse_directives(s);
+            assert!(
+                got.is_empty(),
+                "malformed range directive `{s}` should not parse; got
{got:?}",
+            );
+        }
+    }
+
+    #[test]
+    fn
line_range_slice_returns_inclusive_lines_with_trailing_newlines_preserved() {
+        let text = "alpha\nbeta\ngamma\ndelta\nepsilon\n";
+        let r = LineRange {
+            start: Some(2),
```

```
+            end: Some(4),
+        };
+        assert_eq!(r.slice(text), "beta\ngamma\ndelta\n");
+    }
+
+    #[test]
+    fn line_range_slice_handles_open_endpoints() {
+        let text = "1\n2\n3\n4\n5\n";
+        assert_eq!(
+            LineRange {
+                start: None,
+                end: Some(2)
+            }
+            .slice(text),
+            "1\n2\n",
+        );
+        assert_eq!(
+            LineRange {
+                start: Some(4),
+                end: None
+            }
+            .slice(text),
+            "4\n5\n",
+        );
+        assert_eq!(
+            LineRange {
+                start: None,
+                end: None
+            }
+            .slice(text),
+            "1\n2\n3\n4\n5\n",
+        );
+    }
+
+    #[test]
+    fn line_range_slice_clamps_out_of_range_endpoints() {
+        let text = "1\n2\n3\n";
+        assert_eq!(
+            LineRange {
+                start: Some(2),
+                end: Some(999)
+            }
+            .slice(text),
+            "2\n3\n",
+            "end > line count clamps to end of file",
+        );
+        assert_eq!(
+            LineRange {
+                start: Some(999),
+                end: Some(1000)
+            }
+            .slice(text),
+            "",
+            "fully out-of-range yields empty slice",
+        );
```

```
+    }
+
+    #[test]
+    fn shift_hunk_headers_rewrites_left_and_right_starts_by_offsets() {
+        let diff = "--- a\n+++ b\n@@ -3,18 +3,28 @@\n context\n+added\n";
+        let shifted = shift_hunk_headers(diff, 55, 145);
+        assert!(
+            shifted.contains("@@ -58,18 +148,28 @@"),
+            "expected shifted hunk header; got:\n{shifted}",
+        );
+        assert!(
+            shifted.contains("--- a\n+++ b\n"),
+            "non-hunk lines must pass through unchanged; got:\n{shifted}",
+        );
+        assert!(
+            shifted.contains(" context\n+added\n"),
+            "body lines must pass through unchanged; got:\n{shifted}",
+        );
+    }
+
+    #[test]
+    fn shift_hunk_headers_handles_multiple_hunks_in_one_diff() {
+        let diff = "--- a\n+++ b\n@@ -1,3 +1,3 @@\n line1\n@@ -10,2 +10,2 @@\n
line10\n";
+        let shifted = shift_hunk_headers(diff, 100, 200);
+        assert!(
+            shifted.contains("@@ -101,3 +201,3 @@"),
+            "first hunk shifted; got:\n{shifted}",
+        );
+        assert!(
+            shifted.contains("@@ -110,2 +210,2 @@"),
+            "second hunk shifted; got:\n{shifted}",
+        );
+    }
+
+    #[test]
+    fn shift_hunk_headers_passes_zero_offsets_through_unchanged() {
+        let diff = "--- a\n+++ b\n@@ -3,18 +3,28 @@\n line\n";
+        assert_eq!(shift_hunk_headers(diff, 0, 0), diff);
+    }
+
+    #[test]
+    fn shift_hunk_headers_handles_short_form_with_no_count() {
+        // `@@ -A +C @@` (no `,B`/`,D`) is a valid unified-diff form when
+        // the hunk is exactly one line on each side. The implementation
+        // expands it to the explicit `,1` form when shifting.
+        let diff = "@@ -3 +3 @@ context\n";
+        let shifted = shift_hunk_headers(diff, 10, 20);
+        assert!(
+            shifted.contains("@@ -13,1 +23,1 @@ context"),
+            "short form expanded with shift; got:\n{shifted}",
+        );
+    }
+
+    #[test]
```

```
+    fn shift_hunk_headers_leaves_malformed_at_at_lines_alone() {
+        // `@@ ` followed by something that isn't a well-formed range
+        // pair should not be rewritten.
+        let diff = "@@ not a real header\nbody\n";
+        assert_eq!(shift_hunk_headers(diff, 5, 5), diff);
+    }
+
+    #[test]
+    fn line_range_slice_handles_single_line_range() {
+        let text = "alpha\nbeta\ngamma\n";
+        assert_eq!(
+            LineRange {
+                start: Some(2),
+                end: Some(2)
+            }
+            .slice(text),
+            "beta\n",
+        );
+    }
+
+    #[test]
     fn parse_directives_skips_malformed_arity() {
         for s in ["{{#diff only-one}}", "{{#diff a b c}}", "{{#diff}}"] {
             let got = parse_directives(s);
@@ -395,12 +803,27 @@
     }

     #[test]
-    fn parse_escapes_skips_inside_fenced_code_blocks() {
-        let s = "outside \{{#diff a b}}\n\n```\nlet s = \"\\{{#diff x y}}\";
\n```\n";
-        let escapes = parse_escapes(s);
-        assert_eq!(escapes.len(), 1, "fenced escape should be skipped");
+    fn parse_directives_skips_inside_inline_code_spans() {
+        let s = "Use `{{#diff a b}}` in prose.\n";
+        assert!(
+            parse_directives(s).is_empty(),
+            "directive inside inline backticks should be skipped",
+        );
     }

+    #[test]
+    fn parse_directives_picks_up_directive_after_a_closed_inline_code_span() {
+        let s = "the syntax is `{{#diff a b}}` and {{#diff c d}}\n";
+        let got = parse_directives(s);
+        assert_eq!(
+            got.len(),
+            1,
+            "only the bare directive should parse; got {got:?}"
+        );
+        assert_eq!(got[0].left, "c");
+        assert_eq!(got[0].right, "d");
+    }
+
     use crate::manifest::{MANIFEST_VERSION, Manifest};
```

```
     use std::fs;
     use tempfile::TempDir;
@@ -436,6 +859,8 @@
         let directive = DiffDirective {
             left: "left-tag".into(),
             right: "right-tag".into(),
+            left_range: None,
+            right_range: None,
             span: 0..0,
         };

@@ -445,7 +870,7 @@
     #[test]
     fn resolve_returns_bytes_and_labels_for_known_tags() {
         let (tmp, manifest, directive) = fixture(b"line one\nline two\n",
b"line one\nline TWO\n");
-        let resolved = resolve(&directive, &manifest,
tmp.path()).expect("resolve");
+        let resolved = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect("resolve");
         assert_eq!(resolved.left_label, "left-tag");
         assert_eq!(resolved.right_label, "right-tag");
         assert_eq!(resolved.left_bytes, b"line one\nline two\n");
@@ -457,7 +882,7 @@
         let (tmp, manifest, mut directive) = fixture(b"a", b"b");
         directive.left = "nope".into();

-        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
+        let err = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect_err("should fail");
         assert_eq!(err.tag, "nope");
         assert!(matches!(err.kind, ResolveErrorKind::UnknownTag));
         let msg = format!("{err}");
@@ -472,7 +897,7 @@
         let (tmp, manifest, mut directive) = fixture(b"a", b"b");
         directive.right = "also-nope".into();

-        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
+        let err = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect_err("should fail");
         assert_eq!(err.tag, "also-nope");
         assert!(matches!(err.kind, ResolveErrorKind::UnknownTag));
     }
@@ -483,17 +908,29 @@
         fs::write(tmp.path().join("live-source.txt"), "live one\nlive
two\n").unwrap();
         directive.left = "live:live-source.txt".into();

-        let resolved = resolve(&directive, &manifest,
tmp.path()).expect("resolve");
+        let resolved = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect("resolve");
         assert_eq!(resolved.left_label, "live:live-source.txt");
```

```
          assert_eq!(resolved.left_bytes, b"live one\nlive two\n");
      }

      #[test]
+     fn resolve_resolves_live_operand_against_live_base_not_book_root() {
+         let (tmp, manifest, mut directive) = fixture(b"a", b"b");
+         let chapter_dir = tmp.path().join("src").join("chapters");
+         fs::create_dir_all(&chapter_dir).unwrap();
+         fs::write(chapter_dir.join("sibling.txt"), "from chapter
dir\n").unwrap();
+         directive.left = "live:sibling.txt".into();
+
+         let resolved = resolve(&directive, &manifest, tmp.path(),
&chapter_dir).expect("resolve");
+         assert_eq!(resolved.left_bytes, b"from chapter dir\n");
+     }
+
+     #[test]
      fn resolve_returns_live_file_missing_when_disk_lacks_live_path() {
          let (tmp, manifest, mut directive) = fixture(b"a", b"b");
          directive.left = "live:nope.txt".into();

-         let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
+         let err = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect_err("should fail");
          assert_eq!(err.tag, "live:nope.txt");
          match &err.kind {
              ResolveErrorKind::LiveFileMissing { live_path, .. } => {
@@ -511,7 +948,7 @@
          let (tmp, manifest, directive) = fixture(b"a", b"b");
          fs::remove_file(tmp.path().join("src/listings/left-tag.txt")).unwrap();

-         let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
+         let err = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect_err("should fail");
          assert_eq!(err.tag, "left-tag");
          match &err.kind {
              ResolveErrorKind::FrozenFileMissing { frozen_path, .. } => {
@@ -564,26 +1001,18 @@
      }

      #[test]
-     fn parse_escapes_returns_positions_of_backslashes_before_diff_directives()
{
-         let s = "use \{{#diff a b}} verbatim and \\{{#diff c d}} again";
-         let escapes = parse_escapes(s);
-         assert_eq!(escapes.len(), 2);
-         assert_eq!(&s[escapes[0]..=escapes[0]], "\\");
-         assert_eq!(&s[escapes[1]..=escapes[1]], "\\");
-     }
-
-     #[test]
-     fn parse_escapes_ignores_unescaped_directives() {
```

```
-        let s = "{{#diff a b}}";
-        assert!(parse_escapes(s).is_empty());
-    }
-
-    #[test]
     fn
splice_chapter_replaces_directive_with_diff_fence_and_preserves_surroundings() {
         let (tmp, manifest, _) = fixture(b"line one\nline two\n", b"line
one\nline TWO\n");
         let chapter_path = Path::new("ch99.md");
         let content = "Before paragraph.\n\n{{#diff left-tag right-
tag}}\n\nAfter paragraph.\n";
-        let out = splice_chapter(content, &manifest, tmp.path(),
Some(chapter_path)).unwrap();
+        let out = splice_chapter(
+            content,
+            &manifest,
+            tmp.path(),
+            Some(chapter_path),
+            tmp.path(),
+        )
+        .unwrap();

         assert!(out.starts_with("Before paragraph.\n"), "got:\n{out}");
         assert!(out.ends_with("After paragraph.\n"), "got:\n{out}");
@@ -603,23 +1032,190 @@
             !out.contains("{{#diff"),
             "directive should be consumed; got:\n{out}",
         );
+        assert!(
+            out.contains("data-listing-diff-left=\"left-tag\""),
+            "expected diff-left anchor attribute; got:\n{out}",
+        );
+        assert!(
+            out.contains("data-listing-diff-right=\"right-tag\""),
+            "expected diff-right anchor attribute; got:\n{out}",
+        );
     }

     #[test]
-    fn splice_chapter_strips_leading_backslash_from_escaped_directives() {
-        let (tmp, manifest, _) = fixture(b"a", b"b");
-        let content = "Use \{{#diff a b}} to render a diff.\n";
-        let out = splice_chapter(content, &manifest, tmp.path(),
None).unwrap();
-        assert_eq!(out, "Use {{#diff a b}} to render a diff.\n");
+    fn splice_chapter_slices_both_listings_to_ranges_and_diffs_slices_only() {
+        // Two 5-line files; differ on lines 2 and 4. With ranges 1:2 / 1:2
+        // the diff sees only lines 1-2 of each — the line-2 difference
+        // shows up but the line-4 one doesn't.
+        let (tmp, manifest, _) = fixture(
+            b"line1\nold-2\nline3\nold-4\nline5\n",
+            b"line1\nnew-2\nline3\nnew-4\nline5\n",
+        );
+        let content = "{{#diff left-tag right-tag 1:2 1:2}}\n";
```

```
+        let out = splice_chapter(content, &manifest, tmp.path(), None,
tmp.path()).unwrap();
+        assert!(
+            out.contains("-old-2"),
+            "expected line-2 removal in slice; got:\n{out}",
+        );
+        assert!(
+            out.contains("+new-2"),
+            "expected line-2 addition in slice; got:\n{out}",
+        );
+        assert!(
+            !out.contains("old-4") && !out.contains("new-4"),
+            "lines past the range should not appear in the rendered diff; got:
\n{out}",
+        );
     }

     #[test]
+    fn
splice_chapter_emits_absolute_line_numbers_in_hunk_headers_for_sliced_diff() {
+        // Both files differ on absolute line 60. With ranges 55:65 / 55:65
+        // the slice contains line 60 at slice-relative position 6. Pre-fix
+        // the hunk header read `@@ -... +... @@` keyed to slice positions;
+        // post-fix it must read `@@ -...60... +...60... @@` so a reader
+        // can map the diff's `+`/`-` lines back to absolute line numbers
+        // in the parent file.
+        let mut left = String::new();
+        let mut right = String::new();
+        for i in 1..=70 {
+            left.push_str(&format!("line{i}\n"));
+            if i == 60 {
+                right.push_str("line60-CHANGED\n");
+            } else {
+                right.push_str(&format!("line{i}\n"));
+            }
+        }
+        let (tmp, manifest, _) = fixture(left.as_bytes(), right.as_bytes());
+        let content = "{{#diff left-tag right-tag 55:65 55:65}}\n";
+        let out = splice_chapter(content, &manifest, tmp.path(), None,
tmp.path()).unwrap();
+        // The hunk header should reference absolute line numbers in the
+        // 55-65 window (likely `@@ -57,9 +57,9 @@` since the diff context
+        // around line 60 covers 57-63 absolutely).
+        let hunk_line = out
+            .lines()
+            .find(|l| l.starts_with("@@ "))
+            .unwrap_or_else(|| panic!("expected a hunk header in:\n{out}"));
+        // The starting line numbers must fall within the slice window
+        // [55, 65] — pre-fix they were < 55 (slice-relative).
+        let parts: Vec<&str> = hunk_line.split_whitespace().collect();
+        let left_start: usize = parts[1]
+            .trim_start_matches('-')
+            .split(',')
+            .next()
+            .unwrap()
```

```
+            .parse()
+            .unwrap();
+        let right_start: usize = parts[2]
+            .trim_start_matches('+')
+            .split(',')
+            .next()
+            .unwrap()
+            .parse()
+            .unwrap();
+        assert!(
+            (55..=65).contains(&left_start),
+            "left hunk start must be inside the [55,65] absolute window; got
`{hunk_line}` (left_start={left_start})",
+        );
+        assert!(
+            (55..=65).contains(&right_start),
+            "right hunk start must be inside the [55,65] absolute window; got
`{hunk_line}` (right_start={right_start})",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_emits_range_data_attributes_when_ranges_present() {
+        let (tmp, manifest, _) = fixture(b"a\nb\nc\nd\n", b"a\nB\nc\nD\n");
+        let content = "{{#diff left-tag right-tag 1:2 1:3}}\n";
+        let out = splice_chapter(content, &manifest, tmp.path(), None,
tmp.path()).unwrap();
+        assert!(
+            out.contains(r#"data-listing-diff-left-range="1:2""#),
+            "expected left-range data attribute; got:\n{out}",
+        );
+        assert!(
+            out.contains(r#"data-listing-diff-right-range="1:3""#),
+            "expected right-range data attribute; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn
splice_chapter_preserves_callout_markers_inside_sliced_diff_for_callout_splicer_downstream()
+    {
+        // The chapter pipeline runs splice_diffs THEN splice_callouts. A
+        // CALLOUT marker that lives inside the slice window must survive
+        // the sliced diff render so the downstream callout splicer can
+        // find it and emit a badge. This test asserts the survival; the
+        // end-to-end badge rendering is covered by the e2e suite.
+        let mut left = String::new();
+        let mut right = String::new();
+        for i in 1..=30 {
+            left.push_str(&format!("// row {i}\n"));
+            if i == 15 {
+                right.push_str("// CALLOUT: sliced-marker Verifies callouts
inside a sliced diff range survive.\n");
+            } else {
+                right.push_str(&format!("// row {i}\n"));
```

```
+            }
+        }
+        let (tmp, manifest, _) = fixture(left.as_bytes(), right.as_bytes());
+        let content = "{{#diff left-tag right-tag 10:20 10:20}}\n";
+        let out = splice_chapter(content, &manifest, tmp.path(), None,
tmp.path()).unwrap();
+        assert!(
+            out.contains("CALLOUT: sliced-marker"),
+            "callout marker on line 15 (inside the 10:20 slice) must survive
into the rendered diff body so the callout splicer can pick it up; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_drops_callout_marker_outside_sliced_range() {
+        // A CALLOUT marker outside the slice window must NOT appear in
+        // the rendered diff — neither in the diff body nor in any future
+        // badge — because the slice never reached it.
+        let mut left = String::new();
+        let mut right = String::new();
+        for i in 1..=30 {
+            left.push_str(&format!("// row {i}\n"));
+            if i == 25 {
+                right.push_str("// CALLOUT: outside-slice This must not
survive.\n");
+            } else {
+                right.push_str(&format!("// row {i}\n"));
+            }
+        }
+        let (tmp, manifest, _) = fixture(left.as_bytes(), right.as_bytes());
+        let content = "{{#diff left-tag right-tag 1:10 1:10}}\n";
+        let out = splice_chapter(content, &manifest, tmp.path(), None,
tmp.path()).unwrap();
+        assert!(
+            !out.contains("outside-slice"),
+            "marker outside the 1:10 slice must not appear; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_omits_range_data_attributes_when_no_ranges() {
+        let (tmp, manifest, _) = fixture(b"a\nb\n", b"a\nB\n");
+        let content = "{{#diff left-tag right-tag}}\n";
+        let out = splice_chapter(content, &manifest, tmp.path(), None,
tmp.path()).unwrap();
+        assert!(
+            !out.contains("data-listing-diff-left-range"),
+            "no left-range attr expected without ranges; got:\n{out}",
+        );
+        assert!(
+            !out.contains("data-listing-diff-right-range"),
+            "no right-range attr expected without ranges; got:\n{out}",
+        );
+    }
+
```

```
+    #[test]
     fn
splice_chapter_short_circuits_with_chapter_path_and_line_for_unknown_tag() {
         let (tmp, manifest, _) = fixture(b"a", b"b");
         let chapter_path = Path::new("src/ch99-foo.md");
         let content = "intro\n\nmore\n\n{{#diff missing-tag right-tag}}\n";
-        let err =
-            splice_chapter(content, &manifest, tmp.path(),
Some(chapter_path)).expect_err("err");
+        let err = splice_chapter(
+            content,
+            &manifest,
+            tmp.path(),
+            Some(chapter_path),
+            tmp.path(),
+        )
+        .expect_err("err");
         assert_eq!(err.line, 5, "directive sits on line 5; got: {err}");
         assert_eq!(err.chapter_path.as_deref(), Some(chapter_path));
         let msg = format!("{err}");
```

[1] diff-anchor-dual — Locator anchor for the capture-screenshots tool. Both operands are emitted as separate data-attributes so the tool can locate a diff block by its (LEFT, RIGHT) pair — unique even when multiple diffs share the same RIGHT tag, and unambiguous against the include splicer's `data-listing-tag` anchors.

When slice 6 shipped, the diff above rendered as the "no changes" notice — the frozen `diff-v5` was byte-identical to the live `src/diff.rs`. Readers building this book after slice 7 (the refactor) now see the diff above show real drift instead, and the drift exactly matches the `diff-v5` → `diff-v6` listing in slice 7 below. The chapter source didn't change between the two states; only the live source on disk did. That's the use case for `live:` in a nutshell: notice intended-and-unintended drift, no chapter edit required.

The freeze stability guarantee that AC 7 calls out as *defeated* by `live:` is, in this story, just words on a page — the *Verify Sync with Source* story (ch. 5) is what surfaces a warning at build time when a chapter uses `live:` operands. v0.1.0 ships the directive; ch. 5 ships the warning.

Slice 7 — refactor

With slices 1–6 in the bag and the integration suite green, the refactor slice tidies what the outside-in walk left behind. Three changes:

- **Dead code removed.** `parse_escapes`, the escape-stripping

branch
in `splice_chapter`, and the `escaped_diff_directive_is_left_literal_minus_the_backslash` integration test all go. They tested a code path that can't fire in the real mdbook pipeline (mdbook's `links` preprocessor strips backslash-escapes upstream of any custom preprocessor — see AC 6). The parser's defensive backslash-skip stays: it's cheap, harmless, and covers the case of someone driving the binary directly with a hand-built envelope.

- **`for_each_directive_position` inlined.** The fence-tracking

helper had two callers (parse_directives + parse_escapes); with `parse_escapes` gone it's down to one. Inlining cuts 25 lines of indirection and puts the fence logic right where it's used.

- **`splice_chapter` simplified.** Without the second edit

source (escapes), the function no longer needs to collect edits, sort them, and stitch in a separate pass. `parse_directives` already returns directives in span-order, so the splicer just walks them once and copies through the gaps.

The dogfood payoff lands without any chapter-source edit: the live: diff in the slice 6 subsection above (the {{#diff …}} whose right operand is `live: ../src/diff.rs`) no longer renders as the "no changes" notice — it now shows the real delta between the slice-6 freeze of `src/diff.rs` and the post-refactor source. Same directive, different output, because the live source drifted. That's the use case for `live:` made visible.

```
--- diff-v5
+++ diff-v6
@@ -25,27 +25,46 @@
     const PREFIX: &[u8] = b"{{#diff";
     let bytes = content.as_bytes();
     let mut out = Vec::new();
-    for_each_directive_position(content, |i| {
-        if i > 0 && bytes[i - 1] == b'\\' {
-            return PREFIX.len();
-        }
-        let inner_start = i + PREFIX.len();
-        let Some(end_rel) = content[inner_start..].find("}}") else {
-            return content.len() - i;
+    let mut in_fence = false;
+    let mut line_start = 0;
+    while line_start < bytes.len() {
+        let line_end = match content[line_start..].find('\n') {
+            Some(off) => line_start + off,
+            None => bytes.len(),
         };
-        let directive_end = inner_start + end_rel + 2;
-        let tokens: Vec<&str> = content[inner_start..inner_start + end_rel]
-            .split_whitespace()
-            .collect();
-        if tokens.len() == 2 {
-            out.push(DiffDirective {
-                left: tokens[0].to_string(),
-                right: tokens[1].to_string(),
-                span: i..directive_end,
-            });
+        if line_is_code_fence(&bytes[line_start..line_end]) {
+            in_fence = !in_fence;
+        } else if !in_fence {
+            let mut i = line_start;
+            while i + PREFIX.len() <= line_end {
+                if &bytes[i..i + PREFIX.len()] != PREFIX {
+                    i += 1;
+                    continue;
+                }
+                if i > 0 && bytes[i - 1] == b'\\' {
```

```
+                    i += PREFIX.len();
+                    continue;
+                }
+                let inner_start = i + PREFIX.len();
+                let Some(end_rel) = content[inner_start..].find("}}") else {
+                    break;
+                };
+                let directive_end = inner_start + end_rel + 2;
+                let tokens: Vec<&str> = content[inner_start..inner_start +
end_rel]
+                    .split_whitespace()
+                    .collect();
+                if tokens.len() == 2 {
+                    out.push(DiffDirective {
+                        left: tokens[0].to_string(),
+                        right: tokens[1].to_string(),
+                        span: i..directive_end,
+                    });
+                }
+                i = directive_end;
+            }
         }
-        directive_end - i
-    });
+        line_start = line_end + 1;
+    }
     out
 }

@@ -180,56 +199,6 @@
         .to_string()
 }

-/// Byte positions of `\` characters that immediately precede a `{{#diff`
-/// substring outside fenced code blocks. The splicer drops these so the
-/// literal directive renders to the reader.
-pub fn parse_escapes(content: &str) -> Vec<usize> {
-    const PREFIX: &[u8] = b"{{#diff";
-    let bytes = content.as_bytes();
-    let mut out = Vec::new();
-    for_each_directive_position(content, |i| {
-        if i > 0 && bytes[i - 1] == b'\\' {
-            out.push(i - 1);
-        }
-        PREFIX.len()
-    });
-    out
-}
-
-/// Walks `content` byte-wise, skipping fenced code blocks, and invokes
-/// `visit(i)` at every byte offset `i` where `{{#diff` starts. The closure
-/// returns how many bytes to advance past the match — letting callers
-/// consume the whole directive (or just the prefix) without re-scanning.
-fn for_each_directive_position<F>(content: &str, mut visit: F)
-where
```

```
-    F: FnMut(usize) -> usize,
-{
-    const PREFIX: &[u8] = b"{{#diff";
-    let bytes = content.as_bytes();
-    let mut in_fence = false;
-    let mut line_start = 0;
-    while line_start < bytes.len() {
-        let line_end = match content[line_start..].find('\n') {
-            Some(off) => line_start + off,
-            None => bytes.len(),
-        };
-        if line_is_code_fence(&bytes[line_start..line_end]) {
-            in_fence = !in_fence;
-        } else if !in_fence {
-            let mut i = line_start;
-            while i + PREFIX.len() <= line_end {
-                if &bytes[i..i + PREFIX.len()] == PREFIX {
-                    let advance = visit(i);
-                    i += advance.max(1);
-                } else {
-                    i += 1;
-                }
-            }
-        }
-        line_start = line_end + 1;
-    }
-}
-
 fn line_is_code_fence(line: &[u8]) -> bool {
     let leading_spaces = line.iter().take_while(|&&b| b == b' ').count();
     if leading_spaces > 3 {
@@ -264,22 +233,17 @@
 }

 /// Replace every `{{#diff …}}` directive in `content` with a fenced ` ```diff
`
-/// block of unified-diff text and strip the leading `\` from any
-/// `{{#diff …}}` escape so the literal directive renders to the reader.
-/// Bytes outside those spans are copied through unchanged.
+/// block of unified-diff text. Bytes outside those spans are copied through
+/// unchanged.
 pub fn splice_chapter(
     content: &str,
     manifest: &Manifest,
     book_root: &Path,
     chapter_path: Option<&Path>,
 ) -> Result<String, SpliceError> {
-    let directives = parse_directives(content);
-    let escapes = parse_escapes(content);
-
-    let mut edits: Vec<(usize, usize, String)> =
-        Vec::with_capacity(directives.len() + escapes.len());
-
-    for d in directives {
+    let mut out = String::with_capacity(content.len());
```

```
+    let mut cursor = 0;
+    for d in parse_directives(content) {
         let resolved = resolve(&d, manifest, book_root).map_err(|source|
SpliceError {
             chapter_path: chapter_path.map(Path::to_path_buf),
             line: line_number(content, d.span.start),
@@ -288,20 +252,11 @@
         let left = String::from_utf8_lossy(&resolved.left_bytes);
         let right = String::from_utf8_lossy(&resolved.right_bytes);
         let body = render(&left, &right, &resolved.left_label,
&resolved.right_label);
-        edits.push((d.span.start, d.span.end, format!("```diff\n{body}```")));
-    }
-    for pos in escapes {
-        edits.push((pos, pos + 1, String::new()));
-    }
-
-    edits.sort_by_key(|(start, _, _)| *start);
-
-    let mut out = String::with_capacity(content.len());
-    let mut cursor = 0;
-    for (start, end, replacement) in edits {
-        out.push_str(&content[cursor..start]);
-        out.push_str(&replacement);
-        cursor = end;
+        out.push_str(&content[cursor..d.span.start]);
+        out.push_str("```diff\n");
+        out.push_str(&body);
+        out.push_str("```");
+        cursor = d.span.end;
     }
     out.push_str(&content[cursor..]);
     Ok(out)
@@ -394,13 +349,6 @@
         assert!(parse_directives(s).is_empty());
     }

-    #[test]
-    fn parse_escapes_skips_inside_fenced_code_blocks() {
-        let s = "outside \{{#diff a b}}\n\n```\nlet s = \"\\{{#diff x y}}\";
\n```\n";
-        let escapes = parse_escapes(s);
-        assert_eq!(escapes.len(), 1, "fenced escape should be skipped");
-    }
-
     use crate::manifest::{MANIFEST_VERSION, Manifest};
     use std::fs;
     use tempfile::TempDir;
@@ -564,21 +512,6 @@
     }

     #[test]
-    fn parse_escapes_returns_positions_of_backslashes_before_diff_directives()
{
-        let s = "use \{{#diff a b}} verbatim and \\{{#diff c d}} again";
```

```
-        let escapes = parse_escapes(s);
-        assert_eq!(escapes.len(), 2);
-        assert_eq!(&s[escapes[0]..=escapes[0]], "\\");
-        assert_eq!(&s[escapes[1]..=escapes[1]], "\\");
-    }
-
-    #[test]
-    fn parse_escapes_ignores_unescaped_directives() {
-        let s = "{{#diff a b}}";
-        assert!(parse_escapes(s).is_empty());
-    }
-
-    #[test]
     fn
splice_chapter_replaces_directive_with_diff_fence_and_preserves_surroundings() {
         let (tmp, manifest, _) = fixture(b"line one\nline two\n", b"line
one\nline TWO\n");
         let chapter_path = Path::new("ch99.md");
@@ -603,14 +536,6 @@
             !out.contains("{{#diff"),
             "directive should be consumed; got:\n{out}",
         );
-    }
-
-    #[test]
-    fn splice_chapter_strips_leading_backslash_from_escaped_directives() {
-        let (tmp, manifest, _) = fixture(b"a", b"b");
-        let content = "Use \{{#diff a b}} to render a diff.\n";
-        let out = splice_chapter(content, &manifest, tmp.path(),
None).unwrap();
-        assert_eq!(out, "Use {{#diff a b}} to render a diff.\n");
     }

     #[test]
```

```
--- diffs-tests-v3
+++ diffs-tests-v4
@@ -81,21 +81,6 @@
     );
 }

-#[test]
-fn escaped_diff_directive_is_left_literal_minus_the_backslash() {
-    let book = MinimalDiffsBook::new();
-    let envelope =
-        book.envelope_with_chapter("Use \{{#diff old-tag new-tag}} verbatim in
prose.\n");
-
-    let returned = run_preprocessor(envelope);
-    let content = chapter_content(&returned, "Diff Test");
-
-    assert_eq!(
-        content,
-        "Use {{#diff old-tag new-tag}} verbatim in prose.\n"
```

```
-    );
-}
-
 /// Pipes the envelope through the preprocessor binary and returns the
 /// transformed `Book` parsed from stdout.
 fn run_preprocessor(envelope: String) -> Book {
```

53 → 51 tests (the three `parse_escapes` unit tests, the `splice_chapter_strips_leading_backslash_from_escaped_directives` unit test, and the `escaped_diff_directive_is_left_literal_minus_the_backslash` integration test are gone). All 51 still pass.

Slice 8 — extend ACs 6 and 7 from dogfooding

Writing this very chapter surfaced two real friction points that the original ACs 6 and 7 didn't capture, so slice 8 is a fresh red-green-refactor loop on top of the refactor:

- **AC 6: inline code spans are now a directive-skip context too.**

Twice while drafting ch. 3 a literal `{{#diff a b}}` inside inline backticks (`` `…` ``) crashed the build — the splicer saw it, tried to resolve the operands, and failed. The fix is one block in `parse_directives`: count backticks before the directive's start byte on the same line; if odd, we're inside an inline code span — skip. AC 6's wording widens from "fenced code blocks" to "inline code spans or fenced code blocks".

- **AC 7: `live:<path>` resolves relative to the chapter's source directory, not `book_root`.** Slice 6's resolution against

`book_root` is awkward: every `live:` reference in this very chapter (which lives at `book/src/ch04-…md`) had to spell out `live:../src/diff.rs` rather than the more natural `live:../../src/diff.rs` (mdbook's own `{{#include}}` already uses chapter-relative paths). The fix threads a `chapter_dir` parameter through `splice_chapter` → `resolve` → `resolve_operand`, and `preprocess()` in `main.rs` computes it as `ctx.root.join(&ctx.config.book.src).join(<chapter source dir>)`.

Three failing tests drove the loop (two in `src/diff.rs`, one in `tests/diffs.rs`), then the implementation, then green: 54 tests pass.

```
--- diff-v6
+++ diff-v7
@@ -45,6 +45,11 @@
                     i += PREFIX.len();
                     continue;
                 }
+                let backticks_before = bytes[line_start..i].iter().filter(|&&b|
b == b'`').count();
+                if backticks_before % 2 == 1 {
+                    i += PREFIX.len();
+                    continue;
+                }
                 let inner_start = i + PREFIX.len();
                 let Some(end_rel) = content[inner_start..].find("}}") else {
                     break;
@@ -138,14 +143,20 @@
```

```
 /// Returns at the first failing operand so callers surface one missing tag
 /// at a time — the second tag's resolution can wait for the rebuild after
-/// the first fix.
+/// the first fix. `live_base` is the absolute directory `live:<rel_path>`
+/// operands resolve against; the splicer passes the chapter's source
+/// directory so authors can reference siblings the same way they would for
+/// `{{#include}}`.
 pub fn resolve(
     directive: &DiffDirective,
     manifest: &Manifest,
     book_root: &Path,
+    live_base: &Path,
 ) -> Result<ResolvedDiff, ResolveError> {
-    let (left_label, left_bytes) = resolve_operand(&directive.left, manifest,
book_root)?;
-    let (right_label, right_bytes) = resolve_operand(&directive.right,
manifest, book_root)?;
+    let (left_label, left_bytes) =
+        resolve_operand(&directive.left, manifest, book_root, live_base)?;
+    let (right_label, right_bytes) =
+        resolve_operand(&directive.right, manifest, book_root, live_base)?;
     Ok(ResolvedDiff {
         left_label,
         left_bytes,
@@ -158,9 +169,10 @@
     operand: &str,
     manifest: &Manifest,
     book_root: &Path,
+    live_base: &Path,
 ) -> Result<(String, Vec<u8>), ResolveError> {
     if let Some(rel_path) = operand.strip_prefix("live:") {
-        let live_path = book_root.join(rel_path);
+        let live_path = live_base.join(rel_path);
         let bytes = std::fs::read(&live_path).map_err(|source| ResolveError {
             tag: operand.to_string(),
             kind: ResolveErrorKind::LiveFileMissing {
@@ -234,21 +246,25 @@

 /// Replace every `{{#diff …}}` directive in `content` with a fenced ` ```diff
`
 /// block of unified-diff text. Bytes outside those spans are copied through
-/// unchanged.
+/// unchanged. `chapter_dir` is the absolute directory the chapter's source
+/// markdown lives in; `live:<rel>` operands resolve against it the same way
+/// mdbook's `{{#include <rel>}}` does.
 pub fn splice_chapter(
     content: &str,
     manifest: &Manifest,
     book_root: &Path,
     chapter_path: Option<&Path>,
+    chapter_dir: &Path,
 ) -> Result<String, SpliceError> {
     let mut out = String::with_capacity(content.len());
     let mut cursor = 0;
```

```
     for d in parse_directives(content) {
-        let resolved = resolve(&d, manifest, book_root).map_err(|source|
SpliceError {
-            chapter_path: chapter_path.map(Path::to_path_buf),
-            line: line_number(content, d.span.start),
-            source,
-        })?;
+        let resolved =
+            resolve(&d, manifest, book_root, chapter_dir).map_err(|source|
SpliceError {
+                chapter_path: chapter_path.map(Path::to_path_buf),
+                line: line_number(content, d.span.start),
+                source,
+            })?;
         let left = String::from_utf8_lossy(&resolved.left_bytes);
         let right = String::from_utf8_lossy(&resolved.right_bytes);
         let body = render(&left, &right, &resolved.left_label,
&resolved.right_label);
@@ -349,6 +365,28 @@
         assert!(parse_directives(s).is_empty());
     }

+    #[test]
+    fn parse_directives_skips_inside_inline_code_spans() {
+        let s = "Use `{{#diff a b}}` in prose.\n";
+        assert!(
+            parse_directives(s).is_empty(),
+            "directive inside inline backticks should be skipped",
+        );
+    }
+
+    #[test]
+    fn parse_directives_picks_up_directive_after_a_closed_inline_code_span() {
+        let s = "the syntax is `{{#diff a b}}` and {{#diff c d}}\n";
+        let got = parse_directives(s);
+        assert_eq!(
+            got.len(),
+            1,
+            "only the bare directive should parse; got {got:?}"
+        );
+        assert_eq!(got[0].left, "c");
+        assert_eq!(got[0].right, "d");
+    }
+
     use crate::manifest::{MANIFEST_VERSION, Manifest};
     use std::fs;
     use tempfile::TempDir;
@@ -393,7 +431,7 @@
     #[test]
     fn resolve_returns_bytes_and_labels_for_known_tags() {
         let (tmp, manifest, directive) = fixture(b"line one\nline two\n",
b"line one\nline TWO\n");
-        let resolved = resolve(&directive, &manifest,
tmp.path()).expect("resolve");
+        let resolved = resolve(&directive, &manifest, tmp.path(),
```

```
tmp.path()).expect("resolve");
         assert_eq!(resolved.left_label, "left-tag");
         assert_eq!(resolved.right_label, "right-tag");
         assert_eq!(resolved.left_bytes, b"line one\nline two\n");
@@ -405,7 +443,7 @@
         let (tmp, manifest, mut directive) = fixture(b"a", b"b");
         directive.left = "nope".into();

-        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
+        let err = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect_err("should fail");
         assert_eq!(err.tag, "nope");
         assert!(matches!(err.kind, ResolveErrorKind::UnknownTag));
         let msg = format!("{err}");
@@ -420,7 +458,7 @@
         let (tmp, manifest, mut directive) = fixture(b"a", b"b");
         directive.right = "also-nope".into();

-        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
+        let err = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect_err("should fail");
         assert_eq!(err.tag, "also-nope");
         assert!(matches!(err.kind, ResolveErrorKind::UnknownTag));
     }
@@ -431,17 +469,29 @@
         fs::write(tmp.path().join("live-source.txt"), "live one\nlive
two\n").unwrap();
         directive.left = "live:live-source.txt".into();

-        let resolved = resolve(&directive, &manifest,
tmp.path()).expect("resolve");
+        let resolved = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect("resolve");
         assert_eq!(resolved.left_label, "live:live-source.txt");
         assert_eq!(resolved.left_bytes, b"live one\nlive two\n");
     }

     #[test]
+    fn resolve_resolves_live_operand_against_live_base_not_book_root() {
+        let (tmp, manifest, mut directive) = fixture(b"a", b"b");
+        let chapter_dir = tmp.path().join("src").join("chapters");
+        fs::create_dir_all(&chapter_dir).unwrap();
+        fs::write(chapter_dir.join("sibling.txt"), "from chapter
dir\n").unwrap();
+        directive.left = "live:sibling.txt".into();
+
+        let resolved = resolve(&directive, &manifest, tmp.path(),
&chapter_dir).expect("resolve");
+        assert_eq!(resolved.left_bytes, b"from chapter dir\n");
+    }
+
+    #[test]
     fn resolve_returns_live_file_missing_when_disk_lacks_live_path() {
```

```
         let (tmp, manifest, mut directive) = fixture(b"a", b"b");
         directive.left = "live:nope.txt".into();

-        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
+        let err = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect_err("should fail");
         assert_eq!(err.tag, "live:nope.txt");
         match &err.kind {
             ResolveErrorKind::LiveFileMissing { live_path, .. } => {
@@ -459,7 +509,7 @@
         let (tmp, manifest, directive) = fixture(b"a", b"b");
         fs::remove_file(tmp.path().join("src/listings/left-tag.txt")).unwrap();

-        let err = resolve(&directive, &manifest, tmp.path()).expect_err("should
fail");
+        let err = resolve(&directive, &manifest, tmp.path(),
tmp.path()).expect_err("should fail");
         assert_eq!(err.tag, "left-tag");
         match &err.kind {
             ResolveErrorKind::FrozenFileMissing { frozen_path, .. } => {
@@ -516,7 +566,14 @@
         let (tmp, manifest, _) = fixture(b"line one\nline two\n", b"line
one\nline TWO\n");
         let chapter_path = Path::new("ch99.md");
         let content = "Before paragraph.\n\n{{#diff left-tag right-
tag}}\n\nAfter paragraph.\n";
-        let out = splice_chapter(content, &manifest, tmp.path(),
Some(chapter_path)).unwrap();
+        let out = splice_chapter(
+            content,
+            &manifest,
+            tmp.path(),
+            Some(chapter_path),
+            tmp.path(),
+        )
+        .unwrap();

         assert!(out.starts_with("Before paragraph.\n"), "got:\n{out}");
         assert!(out.ends_with("After paragraph.\n"), "got:\n{out}");
@@ -543,8 +600,14 @@
         let (tmp, manifest, _) = fixture(b"a", b"b");
         let chapter_path = Path::new("src/ch99-foo.md");
         let content = "intro\n\nmore\n\n{{#diff missing-tag right-tag}}\n";
-        let err =
-            splice_chapter(content, &manifest, tmp.path(),
Some(chapter_path)).expect_err("err");
+        let err = splice_chapter(
+            content,
+            &manifest,
+            tmp.path(),
+            Some(chapter_path),
+            tmp.path(),
+        )
+        .expect_err("err");
```

```
        assert_eq!(err.line, 5, "directive sits on line 5; got: {err}");
        assert_eq!(err.chapter_path.as_deref(), Some(chapter_path));
        let msg = format!("{err}");
```

```
--- main-v4
+++ main-v5
@@ -126,6 +126,7 @@
 fn preprocess() -> Result<()> {
     let (ctx, mut book) = mdbook_preprocessor::parse_input(std::io::stdin())?;
     let manifest = Manifest::load(&ctx.root)?;
+    let src_dir = ctx.root.join(&ctx.config.book.src);

     let mut splice_err: Option<anyhow::Error> = None;
     book.for_each_mut(|item| {
@@ -133,11 +134,18 @@
             return;
         }
         if let BookItem::Chapter(chapter) = item {
+            let chapter_dir = chapter
+                .source_path
+                .as_ref()
+                .and_then(|p| p.parent())
+                .map(|d| src_dir.join(d))
+                .unwrap_or_else(|| src_dir.clone());
             match splice_chapter(
                 &chapter.content,
                 &manifest,
                 &ctx.root,
                 chapter.source_path.as_deref(),
+                &chapter_dir,
             ) {
                 Ok(new_content) => chapter.content = new_content,
                 Err(e) => {
```

```
--- diffs-tests-v4
+++ diffs-tests-v5
@@ -60,9 +60,9 @@
 }

 #[test]
-fn live_path_operand_diffs_against_disk_relative_to_book_root() {
+fn live_path_operand_resolves_relative_to_chapter_directory() {
     let book = MinimalDiffsBook::new();
-    book.write_live_file("compose-live.yaml", b"line one\nline LIVE\n");
+    book.write_live_file("src/compose-live.yaml", b"line one\nline LIVE\n");

     let envelope = book.envelope_with_chapter(
         "Diffing live source.\n\n{{#diff old-tag live:compose-live.yaml}}\n",
```

The slice-6 sub-section's live: directive (`live: ../src/diff.rs` as it shipped in slice 6) now reads `live: ../../src/diff.rs` to match the new resolution. The change is honest about the post-slice-8 state of the chapter; readers building older revisions of the book would see the old form.

This slice is a worked example of the methodology working as intended: the original outside-in walk (slices 1–6) shipped a correct, tested primitive. *Using* the primitive on the chapter that documents it surfaced spec gaps — gaps not visible from inside the original ACs. Rather than retconning slice 7's refactor, slice 8 is its own loop with new ACs, new failing tests, new impl. The chapter is longer for it, and the lesson lands.

What this story does not solve

- **Diff highlighting in typst-pdf.** mdbook-typst-pdf 0.7.x has no

`diff` language entry and emits the block as plain monospace. Authors building PDF see uncolored diffs until a later story plumbs Typst color macros around `+/-` lines (or upstream adds a `diff` language). Tracked as a separate small story.

- **Language-aware syntax highlighting *inside* the diff** (e.g.,

Rust syntax overlaid on `+/-` coloring). Neither highlight.js nor typst-pdf does this; would need server-side rendering with `syntect`. Separate story; sketched on the v0.3.0 roadmap.

- **Per-line callouts and anchors on diff output.** Covered by

ch. 4 (*Render Inline Callouts*); the diff primitive emits a bare ```` ```diff ```` fence that ch. 4 layers callouts on top of.

- **Three-way diffs or diffs across renames.** No current driver

in the dogfood book. Would surface on demand.

- **The verify-side warning when `live:<path>` is used.** Ships

with ch. 5 (*Verify Sync with Source*); ch. 3 only ships the directive itself. v0.1.0 binds the two together at the release boundary.

- **Per-chapter tag namespacing**

(`book/src/listings/<chapter>/...`). On the backlog as a separate tiny story; the global flat namespace is fine while the book is small and tags are short.

- **End-to-end browser-side rendering assertions.** This story's

integration tests verify the JSON our binary emits, but nothing exercises the rendered HTML in a real browser. ch. 4 (*Render Inline Callouts*) starts there — its slice 1 stands up a Playwright harness and a failing spec asserting on a rendered callout in the browser, because the outermost layer for callouts is the rendered DOM. Once that harness exists, retrospective browser assertions for the diff primitive (e.g., highlight.js applying `+/-` coloring) are easy follow-ons if desired.

Render Inline Callouts

> **How this chapter was built**
>
> The story shipped in v0.1.0 outside-in. The first slice was the furthest *out* this book had reached: a real Chromium driven by playwright-rs asserting on the rendered DOM of a callout in this very chapter. Each slice shipped as one commit; the **Outside-in narrative** sub-section below grew by one sub-section per slice.

Story

> As a book author, I want to attach inline annotations and named reference points to specific lines of a frozen listing so that my prose can stay keyed to the code even when the code evolves under a new tag.

Acceptance criteria

Inline form (callout markers in the source itself):

1. A frozen listing whose language has a recognised inline-marker

syntax can carry callout markers. When the chapter renders that listing to HTML — whether via `{{#include}}` or as the new side of a `{{#diff}}` (added or context lines, but not removed lines — a deleted marker shouldn't carry a current badge) — each marker produces a numbered badge at the marker's position and an expandable annotation reachable from the badge.
1. The same listing rendered to PDF produces a styled note for

each callout, ordered to match the listing.
1. A callout marker may declare just a label, with no

accompanying annotation. In that case a numbered badge appears at the marker's position but no expandable annotation is rendered. This form serves purely as a stable cross-reference target. Cross-reference and numbering:

1. Chapter prose can reference a callout by its label, and the

reference renders as the same numbered badge, hyperlinked back to the listing occurrence.
1. Badge numbers are assigned ordinally within each listing and

reset between listings. Adding or removing a callout above an existing one renumbers the badges visually but does not break label-based references. Passthrough and robustness:

1. A frozen listing whose language has no recognised inline-

marker syntax is rendered unchanged for inline-form parsing.
1. A comment that resembles a callout marker but does not parse

cleanly is left unchanged in the rendered output (no silent misparse).
1. A chapter reference to a callout label that does not exist

fails the build with a diagnostic that names the missing label and the chapter. HTML rendered shape (refining ACs 1, 3 once the splicer matures past its slice-3 placeholder dl shape):

1. The `CALLOUT:` marker comment line is **removed** from the

rendered HTML listing — the surrounding code is shown verbatim, but the comment that carries the marker metadata does not appear as visible text.
1. The numbered badge is **inline** on the line that previously

held the marker comment in the rendered HTML. Hovering it reveals the body text in a popover; there is no trailing `<dl>`/list element below the listing. Authoring ergonomics (added in slice 9 in response to the long-diff visual issue surfaced by the test-infra refactor):

1. Authors can render a fragment of a frozen listing via

`START:END` line-range syntax in `{{#diff}}` and `{{#include}}` directives. `{{#diff a b 1:30 1:30}}` renders only lines 1-30 of each operand; `{{#include listings/foo.rs:1:30}}` inlines only lines 1-30 of the file. Endpoints are inclusive and 1-based; empty endpoints (`200:`, `:100`, `:`) mean “to end” / “to start” / “whole file”. Out-of-range endpoints clamp silently to the file’s actual line count.

The slice — outside-in narrative outline

The story ships as nine slices plus two refactor passes and a wrap-up chore. Slice 1 is the outermost layer — a browser-driving acceptance test — and the inner slices fill in the layers needed to satisfy it.

Slice	What it adds
1	playwright-rs harness. A failing `#[tokio::test] #[ignore]` in `tests/e2e_callouts.rs` launches Chromium against the rendered ch. 5 HTML and asserts a `[data-callout-badge]` element exists. The test fails (no callouts in ch. 5 yet, no parser, no HTML emitter); ignore keeps the green-build chain passing while later slices grow the rest.
2	Comment-syntax table + generic `parse_callouts` parser parameterised on prefix. Pure unit tests for every prefix in the initial table; verifies body and no-body forms; ignores malformed.
3	HTML emitter — badge at line, `<details>` nearby — wires parser into preprocessor. Handles both `{{#include}}` (the source language’s comment prefix) and `{{#diff}}` (the splicer strips diff `+`/space indicators and tries every comment prefix; removed `-` lines are skipped). Slice 1’s `#[ignore]` comes off and the test goes green for AC 1. `SupportedRenderer` enum extracted here.
4	Label-only inline form (AC 3). Small addition to emitter; new playwright-rs test asserting the bare-anchor case.
5	Cross-reference directive `{{#callout <label>}}` (ACs 4, 8). New playwright-rs test asserting the prose-rendered badge is hyperlinked to the listing-rendered badge anchor.

6	typst-pdf emitter — admonish-note block after the code block (AC 2). Non-browser; assertion is visual or assert_cmd-on-PDF-bytes — decided in the slice.
7	HTML rendered-shape pivot (ACs 9, 10). The slice-3 placeholder shape (CALLOUT comment line visible + trailing `<dl>` of bodies) is replaced with the final shape: marker comment is **stripped** from the rendered listing, and an inline interactive `<span class="callout-badge">` is overlaid on the line that previously held it. Hovering the badge reveals the body in a popover (CSS-only or `<details>`-driven). The trailing `<dl>` is removed for HTML. Cross-refs from slice 5 still resolve to the new badge anchor. New playwright-rs test asserting the comment is gone, the inline badge exists, and the body becomes visible on hover.
8	Screenshot-tool subcommands and include-block locator anchors. The preprocessor intercepts `{{#include listings/TAG.ext}}` directives before mdbook's built-in `links` preprocessor runs and emits a `<div data-listing-tag="TAG">` anchor after the rendered fenced block — mirroring what `\{{#diff}}` already does. The capture-screenshots tool is split into two subcommands matching the two listing-rendering shapes (`include LISTING` and `diff LEFT RIGHT`). No new acceptance criterion (this is tooling, not user-visible book behavior).
refactor (e2e migration)	Adopt `playwright-rs-macros locator!()` for compile-time selector validation, then migrate every JS-string `evaluate_value` sweep in `tests/e2e_callouts.rs` to playwright-rs `Locator` + `expect(...).to_have_*()` assertions. Surfaces a slice-8 dedup bug (duplicate `id="callout-body-LABEL"` when the same label appears in two blocks) and fixes it. No new ACs; pure test-quality + small splicer hardening.
refactor (test infra)	Move shared e2e setup into `tests/common/e2e_harness.rs`: per-test `BrowserContext` for storage isolation, `tracing_subscriber::fmt()` so playwright-rs's `#[tracing::instrument]` spans surface under `RUST_LOG`, and per-test `BrowserContext::tracing()` recording with the trace dropped on success and saved to `target/playwright-traces/<name>.zip` on panic. The harness also dogfoods the new `playwright-rs-trace` crate by parsing the saved trace and printing failed actions to stderr inline. Sharing one `Browser` across tests via `OnceCell` was tried and reverted — `Browser` channels are bound to the `#[tokio::test]` runtime that created them, so subsequent tests deadlock;

	the per-test launch is the price of `#[tokio::test]` runtime isolation. Also folds in the upstream resolution of playwright-rust#89: bump the `playwright-rs` git pin past `401be500` and replace the lone `history.replaceState` JS string in the click-through-navigation test with the new typed `page.clear_url_fragment().await`. The e2e suite is now JS-string-free. The migration also surfaced and fixed a long-standing badge-positioning bug exposed by the slice's 600-line v6→v7 diff: the overlay's CSS positioning formula assumed each line rendered at 1.5em, but mdbook's `<pre>` uses `line-height: normal` (1.13 for monospace), so badges in long diffs drifted 3px per line above their intended row, eventually landing in sibling pres above. Fix: a per-book init script (registered via `additional-js`) measures the previous pre's actual rendered height and writes a `--callout-line-px` CSS custom property the formula picks up. New regression test `every_badge_renders_inside_its_owning_pre` guards against the drift returning.
9	Line-range support for `{{#diff}}` and `\{{#include}}` directives (AC 11). Both directives accept optional `START:END` arguments to render a fragment of a frozen listing — `\{{#diff a b 1:30 1:30}}` and `\{{#include listings/foo.rs:1:30}}`. Surfaced in the previous refactor: a 600-line diff with three callouts spread across it puts cross-ref prose 600 lines below its first badge. Authors can now break long diffs and includes into multiple smaller rendered blocks interleaved with prose, without freezing snippet listings for each fragment.
wrap-up	Mark the callouts primitive shipped in `ROADMAP.md` and materialize this chapter's "What this story does not solve" section.

Outside-in narrative

Slice 1 — playwright-rs harness + failing E2E test

The first slice introduces the outermost-layer test that the rest of the story races to satisfy: a Rust integration test that launches a real Chromium via playwright-rs, navigates to the rendered `ch05-render-inline-callouts.html` on disk, and asserts that a `[data-callout-badge]` element exists with non-empty text content. The test fails today — there's no parser, no HTML emitter, and no callout-marked listing in this chapter yet. `#[ignore]` keeps `cargo test` green for the green-build chain; the author runs `cargo test --test e2e_callouts -- --ignored` once locally to confirm the test really does fail at the badge assertion, then commits.

`Cargo.toml` gains two `[dev-dependencies]`: `playwright-rs` (the Rust bindings) and `tokio` (the async runtime the test uses).

```
--- cargo-toml-v3
+++ cargo-toml-v4
```

```
@@ -22,5 +22,7 @@

 [dev-dependencies]
 assert_cmd = "2"
+playwright-rs = "0.12"
 predicates = "3"
 tempfile = "3"
+tokio = { version = "1", features = ["macros", "rt-multi-thread"] }
```

The new test file is `tests/e2e_callouts.rs`. The naming parallels the other story-scoped integration test files (`tests/install.rs`, `tests/freeze.rs`, `tests/diffs.rs`); the `e2e_` prefix flags the harness tier so future readers don't expect assert_cmd-style assertions from it.

```
use std::path::PathBuf;

use playwright_rs::Playwright;

#[tokio::test]
#[ignore = "no rendered callouts in ch. 4 yet"]
async fn callout_badge_renders_with_data_attribute_in_ch04() {
    let chapter_html = chapter_path();
    let url = format!("file://{}", chapter_html.display());

    let pw = Playwright::launch().await.expect("launch playwright");
    let browser = pw.chromium().launch().await.expect("launch chromium");
    let page = browser.new_page().await.expect("new page");
    page.goto(&url, None).await.expect("goto chapter");

    let badge = page.locator("[data-callout-badge]").await;
    let count = badge.count().await.expect("count badges");
    assert!(
        count > 0,
        "expected at least one [data-callout-badge] element on rendered ch. 4;
got 0",
    );
    let text = badge.first().text_content().await.expect("badge text");
    assert!(
        text.as_deref().is_some_and(|s| !s.trim().is_empty()),
        "expected badge text to be non-empty; got {text:?}",
    );

    browser.close().await.expect("close browser");
}

fn chapter_path() -> PathBuf {
    let manifest_dir = env!("CARGO_MANIFEST_DIR");
    PathBuf::from(manifest_dir)
        .join("book")
        .join("build")
        .join("html")
        .join("ch05-render-inline-callouts.html")
}
```

The test file is frozen as `e2e-callouts-v1` per the per-slice freeze discipline. Slice 3 mints `e2e-callouts-v2` when it removes the `#[ignore]`; subsequent slices that add new tests mint further versions.

Slice 2 — directive parser as a pure unit

Slice 2 adds the first piece slice 3's HTML emitter will need: a parser that turns a frozen listing's source bytes into a list of `Callout { line, label, body }`. Pure function, no IO; the splicer in slice 3 wires it into the preprocessor.

A new `src/callout.rs` declares the `Callout` struct, the `parse_callouts(content, comment_prefix) → Vec<Callout>` entry point, and a `comment_prefix_for_extension(ext) -> Option<&str>` helper that maps file extensions to single-line comment syntaxes. The initial table covers seventeen languages — `#` for yaml/yml/toml/py/sh/bash/tf/hcl, `//` for rs/c/h/cpp/hpp/js/ts/jsx/tsx, `--` for sql. Block-comment-only languages (CSS, plain Markdown) take callouts via the sidecar form instead and return `None` from this lookup.

```
//! Parses inline `CALLOUT:` markers out of a frozen listing's source.

/// Position is a 1-based line number so error diagnostics and the eventual
/// rendered badge anchor can both refer to it directly.
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct Callout {
    pub line: usize,
    pub label: String,
    pub body: Option<String>,
}

/// Walks `content` line by line and returns every well-formed callout
/// marker. A marker is a line whose first non-whitespace content matches
/// `<comment_prefix> CALLOUT: <label>[ <body>]`. Malformed lines are
/// silently skipped — the splicer leaves them in the rendered listing
/// unchanged.
pub fn parse_callouts(content: &str, comment_prefix: &str) -> Vec<Callout> {
    let mut out = Vec::new();
    for (idx, raw_line) in content.lines().enumerate() {
        if let Some(callout) = parse_line(raw_line, comment_prefix, idx + 1) {
            out.push(callout);
        }
    }
    out
}

fn parse_line(raw_line: &str, comment_prefix: &str, line: usize) ->
Option<Callout> {
    let after_prefix = raw_line.trim_start().strip_prefix(comment_prefix)?;
    let after_keyword = after_prefix.strip_prefix('
')?.strip_prefix("CALLOUT:")?;
    let payload = after_keyword.strip_prefix(' ')?;
    let (label, rest) = match payload.split_once(char::is_whitespace) {
        Some((l, r)) => (l, Some(r)),
        None => (payload, None),
    };
    if label.is_empty() || !is_valid_label(label) {
        return None;
```

```
    }
    let body = rest.map(|s| s.trim().to_string()).filter(|s| !s.is_empty());
    Some(Callout {
        line,
        label: label.to_string(),
        body,
    })
}

fn is_valid_label(label: &str) -> bool {
    label
        .chars()
        .all(|c| c.is_ascii_alphanumeric() || c == '-' || c == '_')
}

/// Maps a listing's file extension to the language's single-line comment
/// prefix. Returns `None` for languages without a recognised inline-marker
/// syntax (block-comment-only languages take callouts via the sidecar form
/// instead).
pub fn comment_prefix_for_extension(ext: &str) -> Option<&'static str> {
    match ext {
        "yaml" | "yml" | "toml" | "py" | "sh" | "bash" | "tf" | "hcl" =>
Some("#"),
        "rs" | "c" | "h" | "cpp" | "hpp" | "js" | "ts" | "jsx" | "tsx" =>
Some("//"),
        "sql" => Some("--"),
        _ => None,
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn parses_label_with_body_for_hash_prefix() {
        let s = "key: value\n# CALLOUT: greeting Says hello to the user.\nfoo:
bar\n";
        let got = parse_callouts(s, "#");
        assert_eq!(
            got,
            vec![Callout {
                line: 2,
                label: "greeting".into(),
                body: Some("Says hello to the user.".into()),
            }]
        );
    }

    #[test]
    fn parses_label_only_form_for_hash_prefix() {
        let s = "# CALLOUT: anchor-only\n";
        let got = parse_callouts(s, "#");
        assert_eq!(
            got,
```

```
            vec![Callout {
                line: 1,
                label: "anchor-only".into(),
                body: None,
            }]
        );
    }

    #[test]
    fn parses_double_slash_prefix() {
        let s = "fn main() {\n    // CALLOUT: entry The program starts here.
\n}\n";
        let got = parse_callouts(s, "//");
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].line, 2);
        assert_eq!(got[0].label, "entry");
        assert_eq!(got[0].body.as_deref(), Some("The program starts here."));
    }

    #[test]
    fn parses_double_dash_prefix_for_sql() {
        let s = "SELECT *\n-- CALLOUT: filter Limits to active rows.\nFROM
users;\n";
        let got = parse_callouts(s, "--");
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].label, "filter");
    }

    #[test]
    fn skips_marker_with_wrong_prefix() {
        let s = "# CALLOUT: hash-marker\n";
        assert!(parse_callouts(s, "//").is_empty());
    }

    #[test]
    fn skips_missing_space_between_prefix_and_keyword() {
        let s = "#CALLOUT: nope\n";
        assert!(parse_callouts(s, "#").is_empty());
    }

    #[test]
    fn skips_missing_space_after_keyword() {
        let s = "# CALLOUT:nope\n";
        assert!(parse_callouts(s, "#").is_empty());
    }

    #[test]
    fn skips_empty_label() {
        let s = "# CALLOUT:  body-without-label\n";
        assert!(parse_callouts(s, "#").is_empty());
    }

    #[test]
    fn skips_label_with_invalid_characters() {
        let s = "# CALLOUT: bad/label has body\n";
```

```
        assert!(parse_callouts(s, "#").is_empty());
    }

    #[test]
    fn returns_none_body_when_label_alone_with_trailing_whitespace() {
        let s = "# CALLOUT: alone   \n";
        let got = parse_callouts(s, "#");
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].body, None);
    }

    #[test]
    fn collects_multiple_callouts_in_one_listing() {
        let s = "\
            # first comment\n\
            # CALLOUT: one Body of one.\n\
            key: value\n\
            # CALLOUT: two\n\
            other: thing\n\
            # CALLOUT: three Body of three.\n\
        ";
        let got = parse_callouts(s, "#");
        assert_eq!(got.len(), 3);
        assert_eq!((got[0].line, &got[0].label[..]), (2, "one"));
        assert_eq!((got[1].line, &got[1].label[..]), (4, "two"));
        assert_eq!((got[2].line, &got[2].label[..]), (6, "three"));
        assert_eq!(got[1].body, None);
    }

    #[test]
    fn tolerates_indented_marker() {
        let s = "    # CALLOUT: indented Body text.\n";
        let got = parse_callouts(s, "#");
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].label, "indented");
    }

    #[test]
    fn comment_prefix_for_extension_covers_initial_table() {
        for ext in ["yaml", "yml", "toml", "py", "sh", "bash", "tf", "hcl"] {
            assert_eq!(comment_prefix_for_extension(ext), Some("#"), "ext:
{ext}");
        }
        for ext in ["rs", "c", "h", "cpp", "hpp", "js", "ts", "jsx", "tsx"] {
            assert_eq!(comment_prefix_for_extension(ext), Some("//"), "ext:
{ext}");
        }
        assert_eq!(comment_prefix_for_extension("sql"), Some("--"));
    }

    #[test]
    fn comment_prefix_for_extension_returns_none_for_unknown_languages() {
        assert_eq!(comment_prefix_for_extension("css"), None);
        assert_eq!(comment_prefix_for_extension(""), None);
        assert_eq!(comment_prefix_for_extension("md"), None);
```

```
    }
}
```

The marker grammar:

```
<leading-ws><comment_prefix> CALLOUT: <label>[ <body>]
```

— exactly one space after the prefix, the literal `CALLOUT:`, exactly one space, then a label of `[A-Za-z0-9_-]+`, then either end-of-line or one whitespace + the rest as body. Anything that doesn't match this exactly is silently skipped (AC 7 — no silent misparse, the line stays in the rendered listing as-is). Fourteen unit tests cover the happy paths for all three prefixes plus the malformed-skip cases (wrong prefix, missing space after prefix, missing space after `CALLOUT:`, empty label, invalid label characters, trailing whitespace, indented marker, multiple markers in one listing).

`src/lib.rs` gains `pub mod callout;`.

```
--- lib-v3
+++ lib-v4
@@ -1,5 +1,6 @@
 //! Managed code listings for mdbook.

+pub mod callout;
 pub mod diff;
 pub mod freeze;
 pub mod install;
```

The slice-1 integration test is still `#[ignore]`'d. The parser is plumbing — slice 3 wires it into the preprocessor and emits HTML badges, at which point the test goes green.

Slice 3 — HTML emitter + slice-1 test goes green

Slice 3 wires `parse_callouts` into the preprocessor and emits HTML badges. The simplest emission shape that satisfies the slice-1 acceptance test: leave the rendered code block alone, and append a `<dl class="callouts">` after the closing fence with one `<dt>` per marker (carrying a numbered badge) and one `<dd>` per marker that has a body. Per-listing ordinal numbering (AC 5) falls out naturally — each fenced block walks its own marker list.

```
--- callout-v1
+++ callout-v2
@@ -14,6 +14,7 @@
 /// `<comment_prefix> CALLOUT: <label>[ <body>]`. Malformed lines are
 /// silently skipped — the splicer leaves them in the rendered listing
 /// unchanged.
+// CALLOUT: parse-entry The single entry point: walks lines, calls parse_line,
collects every match.
 pub fn parse_callouts(content: &str, comment_prefix: &str) -> Vec<Callout> {
     let mut out = Vec::new();
     for (idx, raw_line) in content.lines().enumerate() {
@@ -43,6 +44,7 @@
     })
```

```
 }

+// CALLOUT: label-grammar Labels are deliberately narrow: alphanumerics,
hyphens, underscores. Anything else is rejected so labels stay safe to use as
HTML id attributes and URL fragments.
 fn is_valid_label(label: &str) -> bool {
     label
         .chars()
@@ -62,6 +64,210 @@
     }
 }

+/// Maps a fenced-code-block info string to the language's single-line
+/// comment prefix. Accepts the language names authors typically write
+/// after the opening fence (`rust`, `yaml`, `python`, etc.) and falls back
+/// to [`comment_prefix_for_extension`] for any input that's already an
+/// extension (`rs`, `yml`).
+pub fn comment_prefix_for_language(language: &str) -> Option<&'static str> {
+    let normalised = match language {
+        "rust" => "rs",
+        "python" => "py",
+        "javascript" => "js",
+        "typescript" => "ts",
+        "shell" | "zsh" => "sh",
+        "c++" => "cpp",
+        other => other,
+    };
+    comment_prefix_for_extension(normalised)
+}
+
+/// Replace each fenced code block in `content` with the original block plus
+/// (when the block contains callout markers) a trailing `<dl
class="callouts">`
+/// listing each marker's badge and body. Bytes outside fenced blocks are
+/// copied through unchanged.
+// CALLOUT: splice-entry The HTML splicer entry point. The diff splicer in src/
diff.rs has a sister function with the same shape — fence-aware walk + per-block
emit.
+pub fn splice_chapter(content: &str) -> String {
+    let bytes = content.as_bytes();
+    let mut out = String::with_capacity(content.len());
+    let mut cursor = 0;
+    let mut line_start = 0;
+    let mut open: Option<OpenFence> = None;
+
+    while line_start < bytes.len() {
+        let line_end = match content[line_start..].find('\n') {
+            Some(off) => line_start + off,
+            None => bytes.len(),
+        };
+        let line = &content[line_start..line_end];
+
+        match &open {
+            None => {
+                if let Some((info, opener)) = fence_open_info(line) {
```

```
+                    open = Some(OpenFence {
+                        info,
+                        opener,
+                        body_start: line_end + 1,
+                    });
+                }
+            }
+            Some(o) => {
+                if line_closes_fence(line, o.opener) {
+                    let block_text = &content[o.body_start..line_start];
+                    let close_end = if line_end < bytes.len() {
+                        line_end + 1
+                    } else {
+                        line_end
+                    };
+                    let callouts = callouts_for_block(&o.info, block_text);
+                    if !callouts.is_empty() {
+                        out.push_str(&content[cursor..close_end]);
+                        out.push('\n');
+                        out.push_str(&render_callout_list(&callouts));
+                        out.push('\n');
+                        cursor = close_end;
+                    }
+                    open = None;
+                }
+            }
+        }
+
+        if line_end == bytes.len() {
+            break;
+        }
+        line_start = line_end + 1;
+    }
+
+    out.push_str(&content[cursor..]);
+    out
+}
+
+struct OpenFence {
+    info: String,
+    opener: Fence,
+    body_start: usize,
+}
+
+#[derive(Clone, Copy)]
+struct Fence {
+    char: u8,
+    count: usize,
+}
+
+fn fence_open_info(line: &str) -> Option<(String, Fence)> {
+    let trimmed = line.trim_start();
+    let leading_spaces = line.len() - trimmed.len();
+    if leading_spaces > 3 {
+        return None;
```

```
+    }
+    let bytes = trimmed.as_bytes();
+    let fence_char = match bytes.first()? {
+        b'`' => b'`',
+        b'~' => b'~',
+        _ => return None,
+    };
+    let count = bytes.iter().take_while(|&&b| b == fence_char).count();
+    if count < 3 {
+        return None;
+    }
+    Some((
+        trimmed[count..].trim().to_string(),
+        Fence {
+            char: fence_char,
+            count,
+        },
+    ))
+}
+
+/// CommonMark closes a fenced block only with a fence of the same character
+/// at least as long as the opener and a blank info string. Same-character
+/// fences shorter than the opener stay inside the block as literal text —
+/// which is what lets included source files contain `\`\`\`yaml` inside
+/// string literals without prematurely terminating the outer fence.
+fn line_closes_fence(line: &str, opener: Fence) -> bool {
+    let trimmed = line.trim_start();
+    let leading_spaces = line.len() - trimmed.len();
+    if leading_spaces > 3 {
+        return false;
+    }
+    let bytes = trimmed.as_bytes();
+    let count = bytes.iter().take_while(|&&b| b == opener.char).count();
+    if count < opener.count {
+        return false;
+    }
+    trimmed[count..].trim().is_empty()
+}
+
+/// Produce the callout list for a fenced block. `info` is the fence's info
+/// string (`rust`, `yaml`, `diff`, …). Diff blocks are handled specially:
+/// added (`+`) and context (` `) lines are stripped of their diff indicator
+/// before being parsed against every known comment prefix; removed (`-`)
+/// lines and diff metadata (`---`, `+++`, `@@`, `\`) are skipped, since a
+/// callout that's been deleted shouldn't carry a badge in the post-diff
+/// state.
+fn callouts_for_block(info: &str, block_text: &str) -> Vec<Callout> {
+    if info == "diff" {
+        return callouts_from_diff_block(block_text);
+    }
+    if let Some(prefix) = comment_prefix_for_language(info) {
+        return parse_callouts(block_text, prefix);
+    }
+    Vec::new()
+}
```

```
+
+const ALL_COMMENT_PREFIXES: &[&str] = &["//", "#", "--"];
+
+fn callouts_from_diff_block(block_text: &str) -> Vec<Callout> {
+    let mut out = Vec::new();
+    for (idx, raw_line) in block_text.lines().enumerate() {
+        if raw_line.starts_with("---")
+            || raw_line.starts_with("+++")
+            || raw_line.starts_with("@@")
+            || raw_line.starts_with('\\')
+        {
+            continue;
+        }
+        let stripped = if let Some(rest) = raw_line.strip_prefix('+') {
+            rest
+        } else if let Some(rest) = raw_line.strip_prefix(' ') {
+            rest
+        } else {
+            continue;
+        };
+        for prefix in ALL_COMMENT_PREFIXES {
+            if let Some(callout) = parse_line(stripped, prefix, idx + 1) {
+                out.push(callout);
+                break;
+            }
+        }
+    }
+    out
+}
+
+fn render_callout_list(callouts: &[Callout]) -> String {
+    let mut s = String::new();
+    s.push_str("<dl class=\"callouts\">\n");
+    for (idx, c) in callouts.iter().enumerate() {
+        let ordinal = idx + 1;
+        s.push_str(&format!(
+            "  <dt id=\"callout-{label}\"><span class=\"callout-badge\" data-
callout-badge=\"{label}\" data-callout-ordinal=\"{ordinal}\">{ordinal}</span></
dt>\n",
+            label = html_escape(&c.label),
+        ));
+        if let Some(body) = &c.body {
+            s.push_str(&format!("  <dd>{}</dd>\n", html_escape(body)));
+        }
+    }
+    s.push_str("</dl>");
+    s
+}
+
+fn html_escape(s: &str) -> String {
+    s.replace('&', "&amp;")
+        .replace('<', "&lt;")
+        .replace('>', "&gt;")
+        .replace('"', "&quot;")
+}
```

```
+
 #[cfg(test)]
 mod tests {
     use super::*;
@@ -193,4 +399,164 @@
         assert_eq!(comment_prefix_for_extension(""), None);
         assert_eq!(comment_prefix_for_extension("md"), None);
     }
+
+    #[test]
+    fn comment_prefix_for_language_normalises_common_fence_labels() {
+        assert_eq!(comment_prefix_for_language("rust"), Some("//"));
+        assert_eq!(comment_prefix_for_language("python"), Some("#"));
+        assert_eq!(comment_prefix_for_language("javascript"), Some("//"));
+        assert_eq!(comment_prefix_for_language("shell"), Some("#"));
+        assert_eq!(comment_prefix_for_language("c++"), Some("//"));
+        assert_eq!(comment_prefix_for_language("yaml"), Some("#"));
+        assert_eq!(comment_prefix_for_language("rs"), Some("//"));
+    }
+
+    #[test]
+    fn splice_chapter_appends_callout_dl_after_block_with_markers() {
+        let content = "Before paragraph.\n\n\
+            ```yaml\n\
+            service: greeting\n\
+            # CALLOUT: greeting-name The service identifier.\n\
+            endpoint: /hello\n\
+            # CALLOUT: endpoint-path\n\
+            ```\n\n\
+            After paragraph.\n";
+        let out = splice_chapter(content);
+        assert!(out.contains("Before paragraph.\n"));
+        assert!(out.contains("After paragraph.\n"));
+        assert!(out.contains("<dl class=\"callouts\">"));
+        assert!(out.contains("data-callout-badge=\"greeting-name\""));
+        assert!(out.contains("data-callout-ordinal=\"1\""));
+        assert!(out.contains("data-callout-badge=\"endpoint-path\""));
+        assert!(out.contains("data-callout-ordinal=\"2\""));
+        assert!(out.contains("<dd>The service identifier.</dd>"));
+        assert!(
+            !out.contains("<dd>endpoint-path"),
+            "label-only callout should have no <dd> body",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_leaves_block_alone_when_no_markers_present() {
+        let content = "```yaml\nservice: greeting\nendpoint: /hello\n```\n";
+        assert_eq!(splice_chapter(content), content);
+    }
+
+    #[test]
+    fn splice_chapter_skips_block_with_unknown_language() {
+        let content = "```\n# CALLOUT: anchor body text\n```\n";
+        let out = splice_chapter(content);
```

```
+        assert!(!out.contains("data-callout-badge"));
+    }
+
+    #[test]
+    fn
splice_chapter_handles_two_blocks_independently_for_per_listing_numbering() {
+        let content = "\
+            ```yaml\n\
+            # CALLOUT: a-one\n\
+            ```\n\n\
+            ```rust\n\
+            // CALLOUT: b-one\n\
+            // CALLOUT: b-two\n\
+            ```\n";
+        let out = splice_chapter(content);
+        assert!(out.contains("data-callout-badge=\"a-one\""));
+        assert!(out.contains("data-callout-badge=\"b-one\""));
+        assert!(out.contains("data-callout-badge=\"b-two\""));
+        let a_one_ordinal = out
+            .split("data-callout-badge=\"a-one\"")
+            .nth(1)
+            .and_then(|s| s.split("data-callout-ordinal=\"").nth(1))
+            .unwrap_or("");
+        assert!(
+            a_one_ordinal.starts_with("1\""),
+            "first listing's first marker should be ordinal 1; got prefix {}",
+            &a_one_ordinal[..a_one_ordinal.len().min(10)],
+        );
+        let b_two_ordinal = out
+            .split("data-callout-badge=\"b-two\"")
+            .nth(1)
+            .and_then(|s| s.split("data-callout-ordinal=\"").nth(1))
+            .unwrap_or("");
+        assert!(
+            b_two_ordinal.starts_with("2\""),
+            "second listing's second marker should be ordinal 2; got prefix
{}",
+            &b_two_ordinal[..b_two_ordinal.len().min(10)],
+        );
+    }
+
+    #[test]
+    fn splice_chapter_picks_up_callouts_from_added_and_context_diff_lines() {
+        let content = concat!(
+            "```diff\n",
+            "--- a-tag\n",
+            "+++ b-tag\n",
+            "@@ -1,3 +1,4 @@\n",
+            " fn unchanged() {}\n",
+            "-fn removed() {}\n",
+            "+// CALLOUT: added-marker Body for a freshly added marker.\n",
+            " // CALLOUT: context-marker Body for a marker that survived the
diff.\n",
+            "```\n",
+        );
```

```
+        let out = splice_chapter(content);
+        assert!(
+            out.contains("data-callout-badge=\"added-marker\""),
+            "added line marker should render; got:\n{out}",
+        );
+        assert!(
+            out.contains("data-callout-badge=\"context-marker\""),
+            "context line marker should render; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_skips_callouts_on_removed_diff_lines() {
+        let content = concat!(
+            "```diff\n",
+            "--- a-tag\n",
+            "+++ b-tag\n",
+            "@@ -1 +1 @@\n",
+            "-// CALLOUT: gone-marker This callout was removed.\n",
+            "+// CALLOUT: kept-marker This one stays.\n",
+            "```\n",
+        );
+        let out = splice_chapter(content);
+        assert!(out.contains("data-callout-badge=\"kept-marker\""));
+        assert!(
+            !out.contains("data-callout-badge=\"gone-marker\""),
+            "removed-line markers should not render; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_does_not_close_outer_fence_on_shorter_inner_fence() {
+        let content = "
```

rustn

1. let s = "\\n# CALLOUT: not-real-marker\\n";n
2.
3.

```
\n";
+        let out = splice_chapter(content);
+        assert!(
+            out.contains("data-callout-badge=\"real-marker\""),
+            "expected the marker outside the embedded ```yaml string to
render; got:\n{out}",
+        );
+        assert!(
+            !out.contains("data-callout-badge=\"not-real-marker\""),
+            "the marker inside the embedded string is YAML, not Rust — and
the outer fence is rust; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_html_escapes_label_and_body() {
```

```
+        let content = "```yaml\n# CALLOUT: lbl Body with <script> in it.
\n```\n";
+        let out = splice_chapter(content);
+        let dl = out.split("<dl class=\"callouts\">").nth(1).unwrap_or("");
+        assert!(
+            dl.contains("&lt;script&gt;"),
+            "dl body should escape <script>; got:\n{dl}",
+        );
+        assert!(
+            !dl.contains("<script>"),
+            "dl body must not contain raw <script>; got:\n{dl}",
+        );
+    }
 }
```

[1] **parse-entry** — The single entry point: walks lines, calls parse_line, collects every match.

[2] **label-grammar** — Labels are deliberately narrow: alphanumerics, hyphens, underscores. Anything else is rejected so labels stay safe to use as HTML id attributes and URL fragments.

[3] **splice-entry** — The HTML splicer entry point. The diff splicer in src/diff.rs has a sister function with the same shape — fence-aware walk + per-block emit.

[4] **greeting-name** — The service identifier.n

[5] **real-marker** — This one should be picked up.n

Three things are happening in the diff above. First, the `comment_prefix_for_language` helper normalises fence info strings (`rust`, `python`, `c++`, `shell`) to extensions so the same `comment_prefix_for_extension` table from slice 2 covers both shapes. Second, the `splice_chapter` walker tracks fenced code blocks line-by-line and dispatches per fence info: ```` ```rust ```` / ```` ```yaml ```` etc. parse against the language's comment prefix directly, while ```` ```diff ```` strips the `+` or space indicator from each line and tries every known comment prefix (so a diff of any source language carries its callouts through to the rendered HTML). Removed `-` lines and diff metadata (`---`, `+++`, `@@`, `\`) are skipped — a deleted callout shouldn't carry a current badge. Third, three `CALLOUT:` markers were added to the source as a dogfood demonstration; the `<dl>` you see right above is the splicer's output for the diff path on those three markers.

Snapshot (slice 3) of the diff path's dl as it looked the day slice 3 shipped:

1

The single entry point: walks lines, calls parse_line, collects every match.

2

Labels are deliberately narrow: alphanumerics, hyphens, underscores. Anything else is rejected so labels stay safe to use as HTML id attributes and URL fragments.

3

The HTML splicer entry point. The diff splicer in src/diff.rs has a sister function with the same shape — fence-aware walk + per-block emit.

4

The service identifier.\n\

5

This one should be picked up.\n\

To exercise the splicer's `{{#include}}` path on a different input shape, here is the source of the screenshot tool — a small `playwright-rs` script with one CALLOUT marker:

```
//! Capture an element-scoped screenshot of a rendered chapter and write the
//! PNG to a known path. Used by ch. 4's slice-by-slice visual record so each
//! slice's narrative can embed a snapshot of how the chapter rendered the
//! day the slice shipped.

use std::path::PathBuf;

use clap::Parser;
use playwright_rs::Playwright;

#[derive(Parser)]
#[command(version, about, long_about = None)]
struct Cli {
    /// Absolute path to the rendered chapter HTML to load.
    #[arg(long)]
    chapter_html: PathBuf,

    /// CSS selector for the element to screenshot.
    #[arg(long)]
    selector: String,

    /// Zero-based index when the selector matches multiple elements.
    /// Negative values count from the end (`-1` is the last match).
    #[arg(long, default_value_t = 0)]
    nth: i32,

    /// Absolute path to write the PNG to. Parent directories are created.
    #[arg(long)]
    out: PathBuf,
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let cli = Cli::parse();
    if let Some(parent) = cli.out.parent() {
        std::fs::create_dir_all(parent)?;
```

```
    }
    let url = format!("file://{}", cli.chapter_html.display());

    let pw = Playwright::launch().await?;
    let browser = pw.chromium().launch().await?;
    let page = browser.new_page().await?;
    page.goto(&url, None).await?;

    // CALLOUT: locator-pick `--nth` disambiguates when the selector matches
more than one element; zero-based, negative counts from the end.
    let target = page.locator(&cli.selector).await.nth(cli.nth);
    let png = target.screenshot(None).await?;
    std::fs::write(&cli.out, png)?;
    println!("✓ wrote {}", cli.out.display());

    browser.close().await?;
    Ok(())
}
```

[1] locator-pick — `--nth` disambiguates when the selector matches more than one element; zero-based, negative counts from the end.

The `<dl>` directly below this listing is what the splicer emitted for the marker on the `target` line above — one entry, showing the marker doing real work in this very chapter.

Snapshot (slice 3) of the include path's dl:

1

`--nth` disambiguates when the selector matches more than one element; zero-based, negative counts from the end.

Both images are frozen-in-time snapshots. Readers viewing this chapter on a build *after* a later slice will see the live rendered shape above each image differ from the snapshot — slice 4 onward replaces the dl form with proper inline badges, side- margin annotations, and styled themes. The images stay as the record of what slice 3 produced.

The screenshot tool above is itself a workspace member at `tools/capture-screenshots/` — kept in the repo for slice 4 onward to reuse, but excluded from the published `mdbook-listings` crate (own `Cargo.toml` with `publish = false`). Run with `cargo run -p capture-screenshots -- --chapter-html … --selector dl.callouts --nth N --out …` to snapshot a particular match in a particular chapter.

`src/main.rs`'s `preprocess` now chains the diff splicer's output into `callout::splice_chapter`, so `{{#diff}}` resolution and callout rendering both apply to every chapter.

```
--- main-v5
+++ main-v6
@@ -3,7 +3,8 @@

 use anyhow::{Context, Result};
```

```
 use clap::{Parser, Subcommand};
-use mdbook_listings::diff::splice_chapter;
+use mdbook_listings::callout::splice_chapter as splice_callouts;
+use mdbook_listings::diff::splice_chapter as splice_diffs;
 use mdbook_listings::freeze::{FreezeOptions, FreezeOutcome, freeze};
 use mdbook_listings::install::{InstallOutcome, install};
 use mdbook_listings::manifest::Manifest;
@@ -140,14 +141,16 @@
                 .and_then(|p| p.parent())
                 .map(|d| src_dir.join(d))
                 .unwrap_or_else(|| src_dir.clone());
-            match splice_chapter(
+            match splice_diffs(
                 &chapter.content,
                 &manifest,
                 &ctx.root,
                 chapter.source_path.as_deref(),
                 &chapter_dir,
             ) {
-                Ok(new_content) => chapter.content = new_content,
+                Ok(new_content) => {
+                    chapter.content = splice_callouts(&new_content);
+                }
                 Err(e) => {
                     splice_err =
                         Some(anyhow::Error::new(e).context("rendering {{#diff}}
directive failed"));
```

`tests/e2e_callouts.rs` drops its `#[ignore]`. The Playwright test now runs against the just-built ch. 5 HTML, finds the `[data-callout-badge]` elements emitted by the splicer above, and goes green — closing AC 1 end-to-end.

```
--- e2e-callouts-v1
+++ e2e-callouts-v2
@@ -3,7 +3,6 @@
 use playwright_rs::Playwright;

 #[tokio::test]
-#[ignore = "no rendered callouts in ch. 4 yet"]
 async fn callout_badge_renders_with_data_attribute_in_ch04() {
     let chapter_html = chapter_path();
     let url = format!("file://{}", chapter_html.display());
```

Slice 4 — label-only inline form

Slice 4 closes AC 3: a callout marker may declare just a label with no accompanying body, in which case a numbered badge appears but no annotation. As it turns out, the slice-3 emitter already handles this — when `body.is_none()` the emitter skips the `<dd>`, so a label-only marker renders as a `<dt>` with badge and no following `<dd>`. Slice 4's job is therefore a small one: add a label-only marker somewhere ch. 5 includes, and add a Playwright test that pins the visual contract so future slices can't regress it.

The new marker is on the `cli` parse line in the screenshot tool's source — a label-only callout, ready for slice 5's `{{#callout cli-parse}}` directive to point at:

```
--- capture-screenshots-v1
+++ capture-screenshots-v2
@@ -31,6 +31,7 @@

 #[tokio::main]
 async fn main() -> Result<(), Box<dyn std::error::Error>> {
+    // CALLOUT: cli-parse
     let cli = Cli::parse();
     if let Some(parent) = cli.out.parent() {
         std::fs::create_dir_all(parent)?;
```

[1] cli-parse

Snapshot (slice 4) of the dl that the splicer now emits below the screenshot tool's rendered source — two entries this slice (`locator-pick` from slice 3 with a body, plus `cli-parse` added just now as a bare anchor):

1

A new e2e test queries the post-render DOM for the `callout-cli-parse` `<dt>` and asserts its `nextElementSibling` is **not** a `<dd>` — i.e., the label-only form really does produce a bare badge:

```
--- e2e-callouts-v2
+++ e2e-callouts-v3
@@ -3,6 +3,39 @@
 use playwright_rs::Playwright;

 #[tokio::test]
+async fn label_only_callout_renders_badge_without_following_body() {
+    let chapter_html = chapter_path();
+    let url = format!("file://{}", chapter_html.display());
+
+    let pw = Playwright::launch().await.expect("launch playwright");
+    let browser = pw.chromium().launch().await.expect("launch chromium");
+    let page = browser.new_page().await.expect("new page");
+    page.goto(&url, None).await.expect("goto chapter");
+
+    let next_tag: String = page
+        .evaluate_value(
+            "(() => { \
+                const dt = document.querySelector('dt[id=\"callout-cli-
parse\"]'); \
+                if (!dt) return 'NOT_FOUND'; \
+                const next = dt.nextElementSibling; \
+                return next ? next.tagName : 'NONE'; \
+            })()",
+        )
+        .await
+        .expect("evaluate");
+    assert_ne!(
+        next_tag, "NOT_FOUND",
```

```
+        "expected dt#callout-cli-parse to exist on rendered ch. 4",
+    );
+    assert_ne!(
+        next_tag, "DD",
+        "label-only callout's dt must not be followed by a <dd>; got
<{next_tag}>",
+    );
+
+    browser.close().await.expect("close browser");
+}
+
+#[tokio::test]
 async fn callout_badge_renders_with_data_attribute_in_ch04() {
     let chapter_html = chapter_path();
     let url = format!("file://{}", chapter_html.display());
```

Same caveat as slice 3's snapshot: if you're reading this on a build after a later slice, the live render above will show whatever shape that slice produced; the image stays as the slice-4 record.

Slice 5 — cross-reference directive `{{#callout <label>}}`

Slice 5 closes ACs 4 and 8: chapter prose can reference a callout by label and the reference renders as the same numbered badge, hyperlinked back to the listing-side `<dt id="callout-<label>">` anchor; a reference to a label that no marker in the chapter defines fails the build with a diagnostic that names the missing label.

The splicer in `src/callout.rs` becomes two-pass. The first pass walks every fenced block in the chapter and collects a `label → ordinal` map (the ordinal is the badge number that label got at its first occurrence). The second pass scans chapter prose — i.e. the bytes outside any fenced block — for `{{#callout <label>}}` directives and replaces each with an inline anchor. A reference to a label that's not in the map raises `SpliceError::UnknownLabel` and the preprocessor exits non-zero.

The diff itself adds two new CALLOUT markers on slice 5's own new functions (`replace_callout_refs` and `render_callout_ref`), so the dl that the splicer renders below the diff has fresh anchors that this slice's prose then points back at:

```
--- callout-v2
+++ callout-v3
@@ -1,5 +1,7 @@
 //! Parses inline `CALLOUT:` markers out of a frozen listing's source.

+use std::collections::{HashMap, HashSet};
+
 /// Position is a 1-based line number so error diagnostics and the eventual
 /// rendered badge anchor can both refer to it directly.
 #[derive(Debug, Clone, PartialEq, Eq)]
@@ -82,25 +84,90 @@
     comment_prefix_for_extension(normalised)
 }

+/// Errors raised by the callout splicer.
```

```
+#[derive(Debug)]
+pub enum SpliceError {
+    /// A `{{#callout <label>}}` directive named a label that no callout
+    /// marker in the chapter defines.
+    UnknownLabel { label: String },
+}
+
+impl std::fmt::Display for SpliceError {
+    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
+        match self {
+            SpliceError::UnknownLabel { label } => write!(
+                f,
+                "{{{{#callout {label}}}}} references a label that no callout
marker defines \
+                 in this chapter",
+            ),
+        }
+    }
+}
+
+impl std::error::Error for SpliceError {}
+
 /// Replace each fenced code block in `content` with the original block plus
 /// (when the block contains callout markers) a trailing `<dl
class="callouts">`
-/// listing each marker's badge and body. Bytes outside fenced blocks are
-/// copied through unchanged.
+/// listing each marker's badge and body, and replace each
+/// `{{#callout <label>}}` directive in chapter prose (i.e. outside fenced
+/// code blocks) with an inline anchor that links back to the listing badge.
 // CALLOUT: splice-entry The HTML splicer entry point. The diff splicer in src/
diff.rs has a sister function with the same shape — fence-aware walk + per-block
emit.
-pub fn splice_chapter(content: &str) -> String {
-    let bytes = content.as_bytes();
+pub fn splice_chapter(content: &str) -> Result<String, SpliceError> {
+    let label_to_ordinal = collect_first_occurrence_ordinals(content);
+    let with_lists = splice_callout_lists(content, &label_to_ordinal);
+    replace_callout_refs(&with_lists, &label_to_ordinal)
+}
+
+/// Records each label's ordinal at its FIRST occurrence in the chapter.
+/// Subsequent occurrences (e.g. when the same source file is shown via
+/// `{{#diff}}` after being `{{#include}}`'d) are ignored: the first dt
+/// gets `id="callout-<label>"` and acts as the canonical anchor target;
+/// later dts render the badge but no `id` so the HTML stays valid.
+fn collect_first_occurrence_ordinals(content: &str) -> HashMap<String, usize> {
+    let mut map = HashMap::new();
+    for_each_fenced_block_with_span(content, |info, block_text, _body_start,
_close_end| {
+        for (idx, c) in callouts_for_block(info, block_text).iter().enumerate()
{
+            map.entry(c.label.clone()).or_insert(idx + 1);
+        }
+    });
```

```
+    map
+}
+
+fn splice_callout_lists(content: &str, label_to_ordinal: &HashMap<String,
usize>) -> String {
     let mut out = String::with_capacity(content.len());
     let mut cursor = 0;
+    let mut emitted_anchor: HashSet<String> = HashSet::new();
+    for_each_fenced_block_with_span(content, |info, block_text, _body_start,
close_end| {
+        let callouts = callouts_for_block(info, block_text);
+        if !callouts.is_empty() {
+            out.push_str(&content[cursor..close_end]);
+            out.push('\n');
+            out.push_str(&render_callout_list(
+                &callouts,
+                label_to_ordinal,
+                &mut emitted_anchor,
+            ));
+            out.push('\n');
+            cursor = close_end;
+        }
+    });
+    out.push_str(&content[cursor..]);
+    out
+}
+
+fn for_each_fenced_block_with_span<F>(content: &str, mut visit: F)
+where
+    F: FnMut(&str, &str, usize, usize),
+{
+    let bytes = content.as_bytes();
     let mut line_start = 0;
     let mut open: Option<OpenFence> = None;
-
     while line_start < bytes.len() {
         let line_end = match content[line_start..].find('\n') {
             Some(off) => line_start + off,
             None => bytes.len(),
         };
         let line = &content[line_start..line_end];
-
         match &open {
             None => {
                 if let Some((info, opener)) = fence_open_info(line) {
@@ -119,29 +186,89 @@
                     } else {
                         line_end
                     };
-                    let callouts = callouts_for_block(&o.info, block_text);
-                    if !callouts.is_empty() {
-                        out.push_str(&content[cursor..close_end]);
-                        out.push('\n');
-                        out.push_str(&render_callout_list(&callouts));
-                        out.push('\n');
```

```
-                         cursor = close_end;
-                     }
+                     visit(&o.info, block_text, o.body_start, close_end);
                      open = None;
                  }
              }
          }
-
          if line_end == bytes.len() {
              break;
          }
          line_start = line_end + 1;
      }
+}
+
+const CALLOUT_DIRECTIVE_OPEN: &str = "{{#callout ";
+const CALLOUT_DIRECTIVE_CLOSE: &str = "}}";
+
+/// Replace `{{#callout <label>}}` directives that sit outside fenced code
+/// blocks. Directives inside fenced blocks (e.g. literal documentation
+/// examples) pass through untouched so authors can show the syntax.
+// CALLOUT: cross-ref-replace Two-pass entry: skips directives inside fenced
blocks, errors on labels not in the chapter's collected map.
+fn replace_callout_refs(
+    content: &str,
+    label_to_ordinal: &HashMap<String, usize>,
+) -> Result<String, SpliceError> {
+    let mut fence_spans: Vec<(usize, usize)> = Vec::new();
+    for_each_fenced_block_with_span(content, |_info, _text, body_start,
close_end| {
+        fence_spans.push((body_start, close_end));
+    });

+    let in_fence = |pos: usize| {
+        fence_spans
+            .iter()
+            .any(|&(start, end)| pos >= start && pos < end)
+    };
+
+    let mut out = String::with_capacity(content.len());
+    let mut cursor = 0;
+    while let Some(rel) = content[cursor..].find(CALLOUT_DIRECTIVE_OPEN) {
+        let open_at = cursor + rel;
+        if in_fence(open_at) {
+            // Step past the opener so we don't loop on it forever.
+            out.push_str(&content[cursor..open_at +
CALLOUT_DIRECTIVE_OPEN.len()]);
+            cursor = open_at + CALLOUT_DIRECTIVE_OPEN.len();
+            continue;
+        }
+        let label_start = open_at + CALLOUT_DIRECTIVE_OPEN.len();
+        let close_rel = match
content[label_start..].find(CALLOUT_DIRECTIVE_CLOSE) {
+            Some(off) => off,
+            None => {
```

```
+                out.push_str(&content[cursor..label_start]);
+                cursor = label_start;
+                continue;
+            }
+        };
+        let label = content[label_start..label_start + close_rel].trim();
+        if !is_valid_label(label) {
+            out.push_str(&content[cursor..label_start]);
+            cursor = label_start;
+            continue;
+        }
+        let ordinal =
+            label_to_ordinal
+                .get(label)
+                .copied()
+                .ok_or_else(|| SpliceError::UnknownLabel {
+                    label: label.to_string(),
+                })?;
+        out.push_str(&content[cursor..open_at]);
+        out.push_str(&render_callout_ref(label, ordinal));
+        cursor = label_start + close_rel + CALLOUT_DIRECTIVE_CLOSE.len();
+    }
     out.push_str(&content[cursor..]);
-    out
+    Ok(out)
 }

+// CALLOUT: cross-ref-emit Renders the prose-side anchor: matching badge
styling + href to the listing-side dt.
+fn render_callout_ref(label: &str, ordinal: usize) -> String {
+    let label_esc = html_escape(label);
+    format!(
+        "<a class=\"callout-badge callout-ref\" href=\"#callout-{label_esc}\" \
+         data-callout-ref=\"{label_esc}\" data-callout-
ordinal=\"{ordinal}\">{ordinal}</a>",
+    )
+}
+
 struct OpenFence {
     info: String,
     opener: Fence,
@@ -244,14 +371,23 @@
     out
 }

-fn render_callout_list(callouts: &[Callout]) -> String {
+fn render_callout_list(
+    callouts: &[Callout],
+    _label_to_ordinal: &HashMap<String, usize>,
+    emitted_anchor: &mut HashSet<String>,
+) -> String {
     let mut s = String::new();
     s.push_str("<dl class=\"callouts\">\n");
     for (idx, c) in callouts.iter().enumerate() {
         let ordinal = idx + 1;
```

```
+        let label_esc = html_escape(&c.label);
+        let id_attr = if emitted_anchor.insert(c.label.clone()) {
+            format!(" id=\"callout-{label_esc}\"")
+        } else {
+            String::new()
+        };
         s.push_str(&format!(
-            "  <dt id=\"callout-{label}\"><span class=\"callout-badge\" data-
callout-badge=\"{label}\" data-callout-ordinal=\"{ordinal}\">{ordinal}</span></
dt>\n",
-            label = html_escape(&c.label),
+            "  <dt{id_attr}><span class=\"callout-badge\" data-callout-
badge=\"{label_esc}\" data-callout-ordinal=\"{ordinal}\">{ordinal}</span></
dt>\n",
         ));
         if let Some(body) = &c.body {
             s.push_str(&format!("  <dd>{}</dd>\n", html_escape(body)));
@@ -421,7 +557,7 @@
             # CALLOUT: endpoint-path\n\
             ```\n\n\
             After paragraph.\n";
-        let out = splice_chapter(content);
+   let out = splice_chapter(content).expect("splice");
         assert!(out.contains("Before paragraph.\n"));
         assert!(out.contains("After paragraph.\n"));
         assert!(out.contains("<dl class=\"callouts\">"));
@@ -439,13 +575,13 @@
     #[test]
     fn splice_chapter_leaves_block_alone_when_no_markers_present() {
         let content = "```yaml\nservice: greeting\nendpoint: /hello\n```\n";
-        assert_eq!(splice_chapter(content), content);
+   assert_eq!(splice_chapter(content).expect("splice"), content);
     }

     #[test]
     fn splice_chapter_skips_block_with_unknown_language() {
         let content = "```\n# CALLOUT: anchor body text\n```\n";
-        let out = splice_chapter(content);
+   let out = splice_chapter(content).expect("splice");
         assert!(!out.contains("data-callout-badge"));
     }

@@ -459,7 +595,7 @@
             // CALLOUT: b-one\n\
             // CALLOUT: b-two\n\
             ```\n";
-        let out = splice_chapter(content);
+        let out = splice_chapter(content).expect("splice");
         assert!(out.contains("data-callout-badge=\"a-one\""));
         assert!(out.contains("data-callout-badge=\"b-one\""));
         assert!(out.contains("data-callout-badge=\"b-two\""));
@@ -498,7 +634,7 @@
             " // CALLOUT: context-marker Body for a marker that survived the
diff.\n",
             "```\n",
```

```
         );
-        let out = splice_chapter(content);
+        let out = splice_chapter(content).expect("splice");
         assert!(
             out.contains("data-callout-badge=\"added-marker\""),
             "added line marker should render; got:\n{out}",
@@ -520,7 +656,7 @@
             "+// CALLOUT: kept-marker This one stays.\n",
             "```\n",
         );
-        let out = splice_chapter(content);
+        let out = splice_chapter(content).expect("splice");
         assert!(out.contains("data-callout-badge=\"kept-marker\""));
         assert!(
             !out.contains("data-callout-badge=\"gone-marker\""),
@@ -530,11 +666,13 @@

     #[test]
     fn splice_chapter_does_not_close_outer_fence_on_shorter_inner_fence() {
-        let content = "
```

rustn

- let s = "\\n# CALLOUT: not-real-marker\\n";n
-
-

```
\n";
-        let out = splice_chapter(content);
+        let content = concat!(
+            "
```

rustn“,

1. “let s = "\\n# CALLOUT: not-real-marker\\n";n”,
2. “
3. “

```
\n",
+        );
+        let out = splice_chapter(content).expect("splice");
         assert!(
             out.contains("data-callout-badge=\"real-marker\""),
             "expected the marker outside the embedded ```yaml string to
render; got:\n{out}",
@@ -548,7 +686,7 @@
     #[test]
     fn splice_chapter_html_escapes_label_and_body() {
         let content = "```yaml\n# CALLOUT: lbl Body with <script> in it.
\n```\n";
-        let out = splice_chapter(content);
+        let out = splice_chapter(content).expect("splice");
         let dl = out.split("<dl class=\"callouts\">").nth(1).unwrap_or("");
         assert!(
             dl.contains("&lt;script&gt;"),
```

```
@@ -559,4 +697,109 @@
             "dl body must not contain raw <script>; got:\n{dl}",
         );
     }
+
+    #[test]
+    fn
splice_chapter_replaces_callout_directive_with_anchor_to_listing_badge() {
+        let content = concat!(
+            "Prose mentions {{#callout greeting}} the marker.\n\n",
+            "```yaml\n",
+            "# CALLOUT: greeting Says hello.\n",
+            "```\n",
+        );
+        let out = splice_chapter(content).expect("splice");
+        assert!(
+            out.contains("href=\"#callout-greeting\""),
+            "expected anchor href pointing at listing badge id; got:
\n{out}",
+        );
+        assert!(
+            out.contains("data-callout-ref=\"greeting\""),
+            "expected ref-side data attribute; got:\n{out}",
+        );
+        assert!(
+            !out.contains("{{#callout greeting}}"),
+            "directive should be replaced; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_resolves_forward_reference_to_callout_defined_below()
{
+        let content = concat!(
+            "See {{#callout later}} below.\n\n",
+            "```rust\n",
+            "// CALLOUT: later Defined after the reference.\n",
+            "```\n",
+        );
+        let out = splice_chapter(content).expect("splice");
+        assert!(out.contains("href=\"#callout-later\""));
+    }
+
+    #[test]
+    fn splice_chapter_callout_ref_carries_per_listing_ordinal() {
+        let content = concat!(
+            "Reference {{#callout two}} here.\n\n",
+            "```rust\n",
+            "// CALLOUT: one First.\n",
+            "// CALLOUT: two Second.\n",
+            "```\n",
+        );
+        let out = splice_chapter(content).expect("splice");
+        let segment = out.split("data-callout-
ref=\"two\"").nth(1).unwrap_or("");
```

```
+        assert!(
+            segment.contains("data-callout-ordinal=\"2\""),
+            "ref to `two` should carry ordinal 2; got segment:\n{segment}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_unknown_callout_label_returns_error() {
+        let content = "Unknown ref {{#callout missing}} here.\n";
+        let err = splice_chapter(content).expect_err("expected unknown-label
error");
+        match err {
+            SpliceError::UnknownLabel { label } => assert_eq!(label,
"missing"),
+        }
+    }
+
+    #[test]
+    fn splice_chapter_emits_id_only_on_first_occurrence_of_repeated_label()
{
+        // Same source file shown via {{#include}} and {{#diff}} produces
two
+        // dl entries for the same label; only the first carries
id="callout-X"
+        // so the rendered HTML stays valid (no duplicate IDs).
+        let content = concat!(
+            "```rust\n",
+            "// CALLOUT: same Body.\n",
+            "```\n\n",
+            "```diff\n",
+            "+// CALLOUT: same Body.\n",
+            "```\n",
+        );
+        let out = splice_chapter(content).expect("splice");
+        let id_count = out.matches("id=\"callout-same\"").count();
+        assert_eq!(
+            id_count, 1,
+            "expected exactly one id=\"callout-same\"; got {id_count} in:
\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_does_not_replace_callout_directive_inside_code_block()
{
+        // Authors show literal directive syntax inside fenced examples; the
+        // splicer must not rewrite them into anchors.
+        let content = concat!(
+            "```text\n",
+            "See {{#callout greeting}} for details.\n",
+            "```\n\n",
+            "```yaml\n",
+            "# CALLOUT: greeting Says hello.\n",
+            "```\n",
+        );
```

```
+        let out = splice_chapter(content).expect("splice");
+        assert!(
+            out.contains("{{#callout greeting}}"),
+            "literal directive inside code block should pass through; got:
\n{out}",
+        );
+        assert!(
+            !out.contains("href=\"#callout-greeting\""),
+            "should not have rendered anchor for the in-code-block
reference; got:\n{out}",
+        );
+    }
 }
```

[1] **splice-entry** — The HTML splicer entry point. The diff splicer in src/diff.rs has a sister function with the same shape — fence-aware walk + per-block emit.

[2] **cross-ref-replace** — Two-pass entry: skips directives inside fenced blocks, errors on labels not in the chapter's collected map.

[3] **cross-ref-emit** — Renders the prose-side anchor: matching badge styling + href to the listing-side dt.

Snapshot (slice 5) of the dl rendered below the diff above. The v2→v3 diff's context window picks up `splice-entry` (carried over from slice 3 — the new code was added near it) and the two brand-new markers from this slice, `cross-ref-replace` and `cross-ref-emit`:

1

The HTML splicer entry point. The diff splicer in src/diff.rs has a sister function with the same shape — fence-aware walk + per-block emit.

2

Two-pass entry: skips directives inside fenced blocks, errors on labels not in the chapter's collected map.

3

Renders the prose-side anchor: matching badge styling + href to the listing-side dt.

`src/main.rs`'s `preprocess` chain propagates the new `SpliceError` out of `splice_callouts`, so the build stops at the chapter that contains the offending reference instead of silently emitting a broken anchor:

```
--- main-v6
+++ main-v7
@@ -147,14 +147,14 @@
                 &ctx.root,
                 chapter.source_path.as_deref(),
                 &chapter_dir,
-            ) {
-                Ok(new_content) => {
-                    chapter.content = splice_callouts(&new_content);
```

```
-                }
-                Err(e) => {
-                    splice_err =
-                        Some(anyhow::Error::new(e).context("rendering {{#diff}}
directive failed"));
-                }
+            )
+            .map_err(|e| anyhow::Error::new(e).context("rendering {{#diff}}
directive failed"))
+            .and_then(|new_content| {
+                splice_callouts(&new_content)
+                    .map_err(|e| anyhow::Error::new(e).context("rendering
callouts failed"))
+            }) {
+                Ok(new_content) => chapter.content = new_content,
+                Err(e) => splice_err = Some(e),
             }
         }
     });
```

The new e2e test queries the prose-side anchor by its `data-callout-ref` attribute, asserts its `href` matches the listing-side dt id, and confirms the target dt actually exists in the rendered DOM:

```
--- e2e-callouts-v3
+++ e2e-callouts-v4
@@ -60,6 +60,44 @@
     browser.close().await.expect("close browser");
 }

+#[tokio::test]
+async fn callout_cross_ref_renders_as_anchor_to_listing_badge() {
+    let chapter_html = chapter_path();
+    let url = format!("file://{}", chapter_html.display());
+
+    let pw = Playwright::launch().await.expect("launch playwright");
+    let browser = pw.chromium().launch().await.expect("launch chromium");
+    let page = browser.new_page().await.expect("new page");
+    page.goto(&url, None).await.expect("goto chapter");
+
+    let href: String = page
+        .evaluate_value(
+            "(() => { \
+                const a = document.querySelector('a[data-callout-ref=\"cross-
ref-emit\"]'); \
+                return a ? a.getAttribute('href') : 'NOT_FOUND'; \
+            })()",
+        )
+        .await
+        .expect("evaluate href");
+    assert_eq!(
+        href, "#callout-cross-ref-emit",
+        "expected prose-side cross-ref to point at listing badge anchor",
+    );
```

```
+
+    let target_present: String = page
+        .evaluate_value(
+            "(() ⇒ document.querySelector('dt[id=\"callout-cross-ref-
emit\"]') ? 'YES' : 'NO')()",
+        )
+        .await
+        .expect("evaluate anchor presence");
+    assert_eq!(
+        target_present, "YES",
+        "expected listing-side dt#callout-cross-ref-emit to exist as the cross-
ref's target",
+    );
+
+    browser.close().await.expect("close browser");
+}
+
 fn chapter_path() → PathBuf {
     let manifest_dir = env!("CARGO_MANIFEST_DIR");
     PathBuf::from(manifest_dir)
```

To dogfood the directive in this very chapter: the next sentence's badge is a `{{#callout cross-ref-emit}}` directive that this slice's splicer resolves to point at the `cross-ref-emit` marker introduced by the callout-v2→v3 diff above. Clicking it should jump the page to that marker's dt anchor.

See callout **[3]** for the rendering helper this reference resolves to.

Snapshot (slice 5) of the live cross-reference badge embedded in the prose paragraph above:

Same caveat as the earlier slices' snapshots: the image freezes slice 5's rendered shape, while the live badge above will track later slices' styling changes.

Slice 6 — typst-pdf emitter

Slice 6 closes AC 2: the same listing rendered to PDF produces a styled note for each callout, ordered to match the listing. Until this slice the splicer always emitted raw HTML (`<dl class="callouts">`, `<a class="callout-ref">`); typst-pdf has no `<dl>/<a>` support in its markdown→typst conversion, so PDF builds rendered the callouts as escaped raw HTML instead of styled note blocks. Slice 6 makes the splicer renderer-aware: HTML stays unchanged, but for the typst-pdf renderer the same parser output is emitted as a markdown blockquote — bold ordinal + label, optional em-dash plus body — which typst-pdf converts to a quoted note block in the PDF.

A new `SupportedRenderer` enum (Html / TypstPdf) is the dispatch key. The preprocessor reads `ctx.renderer` from the JSON envelope mdbook hands it, looks up the variant once at the top of `preprocess()`, and threads it through `splice_callouts` to the two leaf emitters (`render_callout_list` and `render_callout_ref`). Inputs that name an unrecognised renderer (e.g. a third-party backend that mdbook-listings hasn't been taught about) cause the preprocessor to error rather than silently fall back to one of the known emitters — matching what `supports()` already advertises.

The slice-6 production-code change in `src/callout.rs` is shown as a curated snippet rather than the full v3→v4 diff. The full diff includes new unit-test fixtures whose embedded triple-backtick strings overload the typst-pdf markdown→typst converter; the snippet captures the new enum, the renderer-aware dispatcher, and the new PDF emitter — together they're the entire user-visible production-code change in this slice:

```
/// Which renderer the splicer is producing output for. The HTML emitter
/// uses raw <dl> tags so the rendered DOM carries stable
/// data-callout-badge and dt[id] attributes for cross-refs and e2e
/// assertions; the PDF emitter falls back to a markdown blockquote so
/// the typst-pdf backend renders the callouts as a styled note block
/// without relying on raw HTML passthrough.
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum SupportedRenderer {
    Html,
    TypstPdf,
}

impl SupportedRenderer {
    pub fn from_renderer_name(name: &str) -> Option<Self> {
        match name {
            "html" => Some(Self::Html),
            "typst-pdf" => Some(Self::TypstPdf),
            _ => None,
        }
    }
}

fn render_callout_list(
    callouts: &[Callout],
    _label_to_ordinal: &HashMap<String, usize>,
    emitted_anchor: &mut HashSet<String>,
    renderer: SupportedRenderer,
) -> String {
    match renderer {
        SupportedRenderer::Html => render_callout_list_html(callouts,
emitted_anchor),
        SupportedRenderer::TypstPdf => render_callout_list_pdf(callouts),
    }
}

// CALLOUT: pdf-emit Markdown blockquote with bold ordinal + label, one
paragraph per callout. typst-pdf renders this as a quoted note block; bodyless
markers render as just the label.
fn render_callout_list_pdf(callouts: &[Callout]) -> String {
    let mut s = String::new();
    for (idx, c) in callouts.iter().enumerate() {
        let ordinal = idx + 1;
        if idx > 0 {
            s.push_str("> \n");
        }
        match &c.body {
            Some(body) => {
                s.push_str(&format!("> **[{ordinal}] {}** — {body}\n",
```

```
c.label));
            }
            None ⇒ {
                s.push_str(&format!("> **[{ordinal}] {}**\n", c.label));
            }
        }
    }
    s
}
```

[1] pdf-emit — Markdown blockquote with bold ordinal + label, one paragraph per callout. typst-pdf renders this as a quoted note block; bodyless markers render as just the label.

The file lives under `book/src/snippets/` rather than `book/src/listings/` because it is a hand-curated excerpt rather than a frozen tag — the `mdbook-listings freeze` discipline only applies to byte-exact mirrors of an upstream source file. Snippets are versioned by convention (`-v1`, `-v2`, …) so a later slice that needs to extend the curated excerpt mints a new file rather than mutating an earlier slice's frozen-in-time reference.

The snippet itself dogfoods a CALLOUT marker on the new `render_callout_list_pdf` function (`pdf-emit`), so the splicer emits a `<dl class="callouts">` directly below the snippet above. Snapshot (slice 6) of that HTML dl as it looked the day slice 6 shipped:

1

Markdown blockquote with bold ordinal + label, one paragraph per callout. typst-pdf renders this as a quoted note block; bodyless markers render as just the label.

`src/main.rs`'s `preprocess` resolves the renderer once and passes it through:

```
--- main-v7
+++ main-v8
@@ -3,7 +3,7 @@

 use anyhow::{Context, Result};
 use clap::{Parser, Subcommand};
-use mdbook_listings::callout::splice_chapter as splice_callouts;
+use mdbook_listings::callout::{SupportedRenderer, splice_chapter as
splice_callouts};
 use mdbook_listings::diff::splice_chapter as splice_diffs;
 use mdbook_listings::freeze::{FreezeOptions, FreezeOutcome, freeze};
 use mdbook_listings::install::{InstallOutcome, install};
@@ -128,6 +128,8 @@
     let (ctx, mut book) = mdbook_preprocessor::parse_input(std::io::stdin())?;
     let manifest = Manifest::load(&ctx.root)?;
     let src_dir = ctx.root.join(&ctx.config.book.src);
+    let renderer = SupportedRenderer::from_renderer_name(&ctx.renderer)
+        .with_context(|| format!("unsupported renderer: {}", ctx.renderer))?;

     let mut splice_err: Option<anyhow::Error> = None;
```

```
     book.for_each_mut(|item| {
@@ -150,7 +152,7 @@
             )
             .map_err(|e| anyhow::Error::new(e).context("rendering {{#diff}}
directive failed"))
             .and_then(|new_content| {
-                splice_callouts(&new_content)
+                splice_callouts(&new_content, renderer)
                     .map_err(|e| anyhow::Error::new(e).context("rendering
callouts failed"))
             }) {
                 Ok(new_content) => chapter.content = new_content,
```

A new dev-dep, `pdf-extract` (pure-Rust, no system deps), drives the PDF integration test — robust to typst version bumps because it asserts on body-text substrings rather than byte-exact PDF structure. The test is gated to the Linux CI job that has the typst fonts installed and the just-built PDF available; the cross-platform `Test on …` jobs exclude both `e2e_callouts` and `pdf_callouts` since they need a built book.

`Cargo.toml` gains the single `pdf-extract [dev-dependencies]` entry — kept narrow because the test only needs the crate's `extract_text_from_mem` function:

```
--- cargo-toml-v4
+++ cargo-toml-v5
@@ -16,13 +16,19 @@
 serde = { version = "1", features = ["derive"] }
 serde_json = "1"
 sha2 = "0.11"
-similar = "2"
+similar = "3"
 toml = "1.1"
-toml_edit = "0.22"
+toml_edit = "0.25"

 [dev-dependencies]
 assert_cmd = "2"
+pdf-extract = "0.9"
 playwright-rs = "0.12"
 predicates = "3"
 tempfile = "3"
 tokio = { version = "1", features = ["macros", "rt-multi-thread"] }
+
+# Workspace tools live alongside the crate but are not part of the published
+# crate (each carries `publish = false`). Run with `cargo run -p <name>`.
+[workspace]
+members = ["tools/capture-screenshots"]
```

The new test file is `tests/pdf_callouts.rs`, mirroring the naming convention of the other story-scoped integration test files (`tests/e2e_callouts.rs`, `tests/diffs.rs`). It reads the just-built PDF off disk, runs it through `pdf-extract`, and asserts that two known callout body fragments — `splice-entry`'s "HTML splicer entry point" and `cross-ref-emit`'s "Renders the prose-side anchor" — appear in the extracted text:

```
//! Asserts the typst-pdf renderer emits callout bodies into the PDF.
//! Runs against the just-built `book/build/typst-pdf/*.pdf` (the same
//! artifact CI publishes with the HTML site).

use std::fs;
use std::path::PathBuf;

#[test]
fn ch04_pdf_contains_callout_bodies_emitted_by_pdf_splicer() {
    let pdf_path = pdf_path();
    let bytes =
        fs::read(&pdf_path).unwrap_or_else(|e| panic!("read PDF at {}: {}",
pdf_path.display(), e));
    let text =
        pdf_extract::extract_text_from_mem(&bytes).expect("extract text from
typst-pdf output");

    // The callout-v3 splice-entry marker has a body that's stable across
    // splicer revisions; the PDF emitter should emit it as a blockquote
    // line. We match on the body text fragment so we're robust to ordinal
    // and label-formatting changes.
    assert!(
        text.contains("HTML splicer entry point") || text.contains("splicer
entry point"),
        "expected callout body text in extracted PDF; got first 4KB:\n{}",
        &text[..text.len().min(4096)],
    );
    // The cross-ref-emit body (added in slice 5) should also appear since
    // slice 5's diff exposes its callout marker.
    assert!(
        text.contains("Renders the prose-side anchor"),
        "expected cross-ref-emit body text in extracted PDF; got first 4KB:
\n{}",
        &text[..text.len().min(4096)],
    );
}

fn pdf_path() -> PathBuf {
    let manifest_dir = env!("CARGO_MANIFEST_DIR");
    PathBuf::from(manifest_dir)
        .join("book")
        .join("build")
        .join("typst-pdf")
        .join("mdbook-listings.pdf")
}
```

Snapshot (slice 6) of one PDF page that renders the slice 5 callout-v2→v3 diff. The dl that the HTML emitter produces below the diff appears here as a quoted note block — three entries, bold ordinal + label, em-dash + body — directly under the diff fence:

mdbook-listings

```
+    fn splice_chapter_does_not_replace_callout_directive_inside_code_block()
{
+        // Authors show literal directive syntax inside fenced examples; the
+        // splicer must not rewrite them into anchors.
+        let content = concat!(
+            "```text\n",
+            "See {{#callout greeting}} for details.\n",
+            "```\n\n",
+            "```yaml\n",
+            "# CALLOUT: greeting Says hello.\n",
+            "```\n",
+        );
+        let out = splice_chapter(content).expect("splice");
+        assert!(
+            out.contains("{{#callout greeting}}"),
+            "literal directive inside code block should pass through; got:
\n{out}",
+        );
+        assert!(
+            !out.contains("href=\"#callout-greeting\""),
+            "should not have rendered anchor for the in-code-block
reference; got:\n{out}",
+        );
+    }
 }
```

[1] splice-entry — The HTML splicer entry point. The diff splicer in src/diff.rs has a sister function with the same shape — fence-aware walk + per-block emit.

[2] cross-ref-replace — Two-pass entry: skips directives inside fenced blocks, errors on labels not in the chapter's collected map.

[3] cross-ref-emit — Renders the prose-side anchor: matching badge styling + href to the listing-side dt.

Snapshot (slice 5) of the dl rendered below the diff above. The v2→v3 diff's context window picks up `splice-entry` (carried over from slice 3 — the new code was added near it) and the two brand-new markers from this slice, `cross-ref-replace` and `cross-ref-emit`:

The visual on this page is a frozen snapshot of slice 6's PDF output; the page number itself shifts as the book grows. CI runs `cargo test --test pdf_callouts` against the just-built PDF

on every push, so any regression in the PDF emitter surfaces as a failed assertion rather than a quietly-broken render.

Slice 7 — HTML rendered-shape pivot

Slice 7 closes ACs 9 and 10. The slice-3 placeholder shape (CALLOUT comment line visible in the listing plus a trailing `<dl class="callouts">` of bodies) is replaced with the final shape:

- The marker comment is **stripped** from the rendered listing

for `{{#include}}` blocks (every recognised language with an inline-marker syntax). Diff blocks pass through unchanged so the diff format stays valid; the canonical badge anchor lives on the include, not on the diff history.

- The trailing `<dl>` is gone. In its place an absolutely-

positioned `<div class="callout-overlay">` sibling holds one interactive `<button class="callout-badge">` per marker, each carrying the post-strip `data-callout-line` so CSS positions it on the line that previously held the marker comment.

- The vertical positioning is completely JavaScript-free. The Rust

HTML emitter calculates the total lines in the code block and injects `--callout-listing-lines` directly into the DOM, which the bundled CSS uses to perfectly align the badge to the line.

- Hovering or keyboard-focusing the badge reveals its body in a

popover (a sibling `<div class="callout-body">`). The entrance and exit animations are choreographed purely in CSS using `clip-path`, `color`, and `visibility` transitions (eliminating the need for abrupt `[hidden]` attribute toggles). Label-only markers emit no popover at all — just the badge. Slice 7 mints a new version of the slice-6 snippet, `callout-pdf-emit-snippet-v2`, that extends the curated excerpt with the two cross-ref-related functions (`replace_callout_refs` and `render_callout_ref`) so the callout markers attached to them (`cross-ref-replace`, `cross-ref-emit`) now have rendered `<button>` anchors. Slice 5's `{{#callout cross-ref-emit}}` cross-reference resolves to that anchor:

```
/// Which renderer the splicer is producing output for. The HTML emitter
/// uses raw <dl> tags so the rendered DOM carries stable
/// data-callout-badge and dt[id] attributes for cross-refs and e2e
/// assertions; the PDF emitter falls back to a markdown blockquote so
/// the typst-pdf backend renders the callouts as a styled note block
/// without relying on raw HTML passthrough.
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum SupportedRenderer {
    Html,
    TypstPdf,
}

impl SupportedRenderer {
    pub fn from_renderer_name(name: &str) -> Option<Self> {
        match name {
            "html" => Some(Self::Html),
            "typst-pdf" => Some(Self::TypstPdf),
            _ => None,
        }
    }
}
```

```
fn render_callout_list(
    callouts: &[Callout],
    _label_to_ordinal: &HashMap<String, usize>,
    emitted_anchor: &mut HashSet<String>,
    renderer: SupportedRenderer,
) -> String {
    match renderer {
        SupportedRenderer::Html => render_callout_list_html(callouts,
emitted_anchor),
        SupportedRenderer::TypstPdf => render_callout_list_pdf(callouts),
    }
}

// CALLOUT: pdf-emit Markdown blockquote with bold ordinal + label, one
paragraph per callout. typst-pdf renders this as a quoted note block; bodyless
markers render as just the label.
fn render_callout_list_pdf(callouts: &[Callout]) -> String {
    let mut s = String::new();
    for (idx, c) in callouts.iter().enumerate() {
        let ordinal = idx + 1;
        if idx > 0 {
            s.push_str("> \n");
        }
        match &c.body {
            Some(body) => {
                s.push_str(&format!("> **[{ordinal}] {}** — {body}\n",
c.label));
            }
            None => {
                s.push_str(&format!("> **[{ordinal}] {}**\n", c.label));
            }
        }
    }
    s
}

// CALLOUT: cross-ref-replace Two-pass entry: skips directives inside fenced
blocks, errors on labels not in the chapter's collected map.
fn replace_callout_refs(
    content: &str,
    label_to_ordinal: &HashMap<String, usize>,
) -> Result<String, SpliceError> {
    /* ... directive scan + label resolve, omitted for brevity ... */
    Ok(content.to_string())
}

// CALLOUT: cross-ref-emit Renders the prose-side anchor for HTML; falls back to
a bracketed badge for typst-pdf where raw HTML doesn't carry through.
fn render_callout_ref(label: &str, ordinal: usize, renderer: SupportedRenderer)
-> String {
    match renderer {
        SupportedRenderer::Html => {
            let label_esc = html_escape(label);
            format!(
                "<a class=\"callout-badge callout-ref\" href=\"#callout-
```

```
{label_esc}\" \
                  data-callout-ref=\"{label_esc}\" data-callout-
ordinal=\"{ordinal}\">{ordinal}</a>",
            )
        }
        SupportedRenderer::TypstPdf => format!("**[{ordinal}]**"),
    }
}
```

[1] pdf-emit — Markdown blockquote with bold ordinal + label, one paragraph per callout. typst-pdf renders this as a quoted note block; bodyless markers render as just the label.

[2] cross-ref-replace — Two-pass entry: skips directives inside fenced blocks, errors on labels not in the chapter's collected map.

[3] cross-ref-emit — Renders the prose-side anchor for HTML; falls back to a bracketed badge for typst-pdf where raw HTML doesn't carry through.

Snippets are versioned by convention because slice 6's narrative references v1 and that text shouldn't silently drift when later slices extend the excerpt; minting a new file preserves the slice-6 reference verbatim.

To dogfood the label-only form (`cli-parse`) under the new shape, the slice-4 frozen `capture-screenshots-v2.rs` is included here as well. The `// CALLOUT: cli-parse` line is stripped from the rendered listing; in its place the splicer's overlay div holds a bare badge button on the `Cli::parse()` line:

```
//! Capture an element-scoped screenshot of a rendered chapter and write the
//! PNG to a known path. Used by ch. 4's slice-by-slice visual record so each
//! slice's narrative can embed a snapshot of how the chapter rendered the
//! day the slice shipped.

use std::path::PathBuf;

use clap::Parser;
use playwright_rs::Playwright;

#[derive(Parser)]
#[command(version, about, long_about = None)]
struct Cli {
    /// Absolute path to the rendered chapter HTML to load.
    #[arg(long)]
    chapter_html: PathBuf,

    /// CSS selector for the element to screenshot.
    #[arg(long)]
    selector: String,

    /// Zero-based index when the selector matches multiple elements.
    /// Negative values count from the end (`-1` is the last match).
```

```
    #[arg(long, default_value_t = 0)]
    nth: i32,

    /// Absolute path to write the PNG to. Parent directories are created.
    #[arg(long)]
    out: PathBuf,
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // CALLOUT: cli-parse
    let cli = Cli::parse();
    if let Some(parent) = cli.out.parent() {
        std::fs::create_dir_all(parent)?;
    }
    let url = format!("file://{}", cli.chapter_html.display());

    let pw = Playwright::launch().await?;
    let browser = pw.chromium().launch().await?;
    let page = browser.new_page().await?;
    page.goto(&url, None).await?;

    // CALLOUT: locator-pick `--nth` disambiguates when the selector matches
more than one element; zero-based, negative counts from the end.
    let target = page.locator(&cli.selector).await.nth(cli.nth);
    let png = target.screenshot(None).await?;
    std::fs::write(&cli.out, png)?;
    println!("✓ wrote {}", cli.out.display());

    browser.close().await?;
    Ok(())
}
```

[1] cli-parse

[2] locator-pick — `--nth` disambiguates when the selector matches more than one element; zero-based, negative counts from the end.

`src/callout.rs` gains the new `splice_callout_lists_html` emitter that walks each fenced block, calls `strip_marker_lines` to rewrite the body without marker comments, and appends `render_callout_overlay_html` after the closing fence. The PDF emitter (`splice_callout_lists_pdf`) is unchanged from slice 6 — slice 9 is the PDF-side rendered-shape pivot.

The bundled CSS asset (`assets/mdbook-listings.css`) gains the positioning + hover rules for the new shape. The `install` command writes this file into the book's theme directory just like before; users who already installed mdbook-listings need to rerun `mdbook-listings install` to pick up the slice-7 CSS.

Snapshot (slice 7) of the rendered HTML for the screenshot tool include — the marker comment is gone and the badge sits at the right margin of its line:

```
//! Capture an element-scoped screenshot of a rendered chapter and write the
//! PNG to a known path. Used by ch. 4's slice-by-slice visual record so each
//! slice's narrative can embed a snapshot of how the chapter rendered the
//! day the slice shipped.

use std::path::PathBuf;
```


```

#[derive(Parser)]
#[command(version, about, long_about = None)]
struct Cli {
    /// Absolute path to the rendered chapter HTML to load.
    #[arg(long)]
    chapter_html: PathBuf,

    /// CSS selector for the element to screenshot.
    #[arg(long)]
    selector: String,

    /// Zero-based index when the selector matches multiple elements.
    /// Negative values count from the end (`-1` is the last match).
    #[arg(long, default_value_t = 0)]
    nth: i32,

    /// Absolute path to write the PNG to. Parent directories are created.
    #[arg(long)]
    out: PathBuf,
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let cli = Cli::parse();
    if let Some(parent) = cli.out.parent() {
        std::fs::create_dir_all(parent)?;
    }
    let url = format!("file://{}", cli.chapter_html.display());

    let pw = Playwright::launch().await?;
    let browser = pw.chromium().launch().await?;
    let page = browser.new_page().await?;
    page.goto(&url, None).await?;

    let target = page.locator(&cli.selector).await.nth(cli.nth);
    let png = target.screenshot(None).await?;
    std::fs::write(&cli.out, png)?;
    println!("✓ wrote {}", cli.out.display());

    browser.close().await?;
    Ok(())
}
```

The dl is gone; the badges are interactive; the body popover shows on hover. Visual reference is from the day slice 7 shipped — later slices may restyle.

The screenshot above was produced by the workspace tool `tools/capture-screenshots/`, which slice 7 also evolved: it now takes a positional listing tag (`capture-screenshots e2e-callouts-v5`) and finds the listing in the rendered HTML via the `<div data-listing-tag>` anchor that the diff splicer just learned to emit, with a callout-badge fallback for `{{#include}}` blocks whose source has at least one `CALLOUT:` marker. That fallback has a blind spot —

listings without callouts and not on the right side of any diff aren't addressable. Slice 8 closes that gap by re-engineering the tool into two subcommands (`include LISTING` and `diff LEFT RIGHT`) backed by a preprocessor pass that injects the same kind of locator anchor after every `{{#include listings/...}}` block.

Slice 8 — screenshot-tool subcommands and include-block locator anchors

Slice 8 doesn't satisfy any chapter AC (those are 1–12 from the section above); it's tooling that closes a usability gap exposed by slice 7 dogfooding. With slice 7 the screenshot tool can address listings shown via `{{#diff}}` (locator: the `<div data-listing-tag>` anchor the diff splicer learned to emit) and listings shown via `{{#include}}` *that have at least one `CALLOUT:` marker* (fallback locator: the `button[id="callout-LABEL"]` element from the rendered overlay). Listings without callouts and not referenced by any diff were unreachable.

The fix has two parts: a preprocessor side (a new include splicer + a richer diff anchor) and a tool side (subcommands matching the two listing-rendering shapes).

Preprocessor — new `src/include.rs` module. Intercepts `{{#include listings/TAG.ext}}` directives BEFORE mdbook's built-in `links` preprocessor would expand them, replaces each with the file's bytes, and emits a `<div data-listing-tag="TAG">` anchor after the enclosing fenced block's closing fence line. To run before `links`, the `book/book.toml` listings-preprocessor entry adds `before = ["admonish", "links"]` — without that ordering, `links` expands every `{{#include}}` first and the include splicer silently no-ops. The splicer's path-prefix dispatch is at callout **[2]**; the entry point that drives the whole replace-and-emit walk is at callout {{#callout include-splice-entry}}; the line that drops the locator anchor is at callout **[5]**:

```rust
//! Intercepts `{{#include listings/...}}` and `{{#include snippets/...}}`
//! before mdbook's built-in `links` preprocessor expands them, so the
//! callout splicer downstream can find any `CALLOUT:` markers in the
//! included source and so frozen-listing includes get a locator anchor.

use std::ops::Range;
use std::path::{Path, PathBuf};

use crate::callout::for_each_fenced_block_with_span;

#[derive(Debug, Clone, PartialEq, Eq)]
pub struct IncludeDirective {
    pub tag: Option<String>,
    pub rel_path: String,
    pub span: Range<usize>,
    pub fence_close_end: Option<usize>,
}

// CALLOUT: parse-entry Two-pass scan: first collect fence body spans, then walk
lines for `{{#include ...}}` directives. Skips backslash-escaped forms and
directives inside inline code spans (chapter prose quotes the syntax verbatim).
pub fn parse_listing_includes(content: &str) -> Vec<IncludeDirective> {
    let mut fences: Vec<(usize, usize)> = Vec::new();
    for_each_fenced_block_with_span(content, |_info, _text, body_start,
close_end| {
        fences.push((body_start, close_end));
    });
```

```
    const PREFIX: &[u8] = b"{{#include ";
    let bytes = content.as_bytes();
    let mut out = Vec::new();
    let mut line_start = 0;
    while line_start < bytes.len() {
        let line_end = match content[line_start..].find('\n') {
            Some(off) => line_start + off,
            None => bytes.len(),
        };
        let mut i = line_start;
        while i + PREFIX.len() <= line_end {
            if &bytes[i..i + PREFIX.len()] != PREFIX {
                i += 1;
                continue;
            }
            if i > 0 && bytes[i - 1] == b'\\' {
                i += PREFIX.len();
                continue;
            }
            let backticks_before = bytes[line_start..i].iter().filter(|&&b| b ==
b'`').count();
            if backticks_before % 2 == 1 {
                i += PREFIX.len();
                continue;
            }
            let inner_start = i + PREFIX.len();
            let Some(end_rel) = content[inner_start..].find("}}") else {
                break;
            };
            let directive_end = inner_start + end_rel + 2;
            let path = content[inner_start..inner_start + end_rel].trim();
            // CALLOUT: snippets-intercept Two prefixes are intercepted:
`listings/` (frozen tags — emit anchor) and `snippets/` (no anchor; we expand to
give the callout splicer a shot at any CALLOUT markers in the snippet source).
Other forms fall through to mdbook's built-in `links` preprocessor.
            let intercepted = path.starts_with("listings/") ||
path.starts_with("snippets/");
            if !intercepted || path.contains(':') {
                i = directive_end;
                continue;
            }
            // CALLOUT: tag-from-stem Tag is the file stem of `listings/...`
paths so `listings/sub/foo.rs` and `listings/foo.rs` produce the same anchor;
subdirectory stem collisions would clash on the anchor, but the book has none
today.
            let tag = if path.starts_with("listings/") {
                Some(
                    std::path::Path::new(path)
                        .file_stem()
                        .and_then(|s| s.to_str())
                        .unwrap_or("")
                        .to_string(),
                )
            } else {
                None
```

```
            };
            let fence_close_end = fences
                .iter()
                .find(|(body_start, close_end)| i >= *body_start && i <
*close_end)
                .map(|(_, close_end)| *close_end);
            out.push(IncludeDirective {
                tag,
                rel_path: path.to_string(),
                span: i..directive_end,
                fence_close_end,
            });
            i = directive_end;
        }
        if line_end == bytes.len() {
            break;
        }
        line_start = line_end + 1;
    }
    out
}

#[derive(Debug)]
pub enum SpliceError {
    ListingFileMissing {
        tag: String,
        path: PathBuf,
        source: std::io::Error,
        line: usize,
        chapter_path: Option<PathBuf>,
    },
    ListingIncludeOutsideFence {
        tag: String,
        line: usize,
        chapter_path: Option<PathBuf>,
    },
}

impl std::fmt::Display for SpliceError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            SpliceError::ListingFileMissing {
                tag,
                path,
                source,
                line,
                chapter_path,
            } => {
                write!(
                    f,
                    "{}:{line}: {{{{#include listings/{tag}.…}}}} references
missing file {}: {source}",
                    chapter_path
                        .as_deref()
                        .map(|p| p.display().to_string())
```

```
                        .unwrap_or_else(|| "<chapter>".into()),
                    path.display(),
                )
            }
            SpliceError::ListingIncludeOutsideFence {
                tag,
                line,
                chapter_path,
            } => {
                write!(
                    f,
                    "{}:{line}: {{{{#include listings/{tag}.…}}}} appears
outside any fenced code block; \
                     wrap it in ```<lang> ... ``` so the anchor has a <pre>
sibling",
                    chapter_path
                        .as_deref()
                        .map(|p| p.display().to_string())
                        .unwrap_or_else(|| "<chapter>".into()),
                )
            }
        }
    }
}

impl std::error::Error for SpliceError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        match self {
            SpliceError::ListingFileMissing { source, .. } => Some(source),
            SpliceError::ListingIncludeOutsideFence { .. } => None,
        }
    }
}

// CALLOUT: include-splice-entry The HTML splicer entry point. Walks every
intercepted directive; replaces with file body and (for `listings/`) drops a
`<div data-listing-tag>` locator anchor after the closing fence.
pub fn splice_chapter(
    content: &str,
    src_dir: &Path,
    chapter_path: Option<&Path>,
) -> Result<String, SpliceError> {
    let directives = parse_listing_includes(content);
    if directives.is_empty() {
        return Ok(content.to_string());
    }

    let mut out = String::with_capacity(content.len() * 2);
    let mut cursor = 0;
    for d in &directives {
        let Some(close_end) = d.fence_close_end else {
            return Err(SpliceError::ListingIncludeOutsideFence {
                tag: d.tag.clone().unwrap_or_else(|| d.rel_path.clone()),
                line: line_number(content, d.span.start),
                chapter_path: chapter_path.map(Path::to_path_buf),
```

```
            });
        };
        let abs_path = src_dir.join(&d.rel_path);
        let mut body = std::fs::read_to_string(&abs_path).map_err(|source| {
            SpliceError::ListingFileMissing {
                tag: d.tag.clone().unwrap_or_else(|| d.rel_path.clone()),
                path: abs_path.clone(),
                source,
                line: line_number(content, d.span.start),
                chapter_path: chapter_path.map(Path::to_path_buf),
            }
        })?;
        // Why: the chapter's newline-after-directive (preserved via
        // `content[d.span.end..]`) terminates the last content line; keeping
        // the file's own trailing newline produces a blank line before the
        // closing fence.
        while body.ends_with('\n') {
            body.pop();
        }
        out.push_str(&content[cursor..d.span.start]);
        out.push_str(&body);
        out.push_str(&content[d.span.end..close_end]);
        if let Some(tag) = &d.tag {
            // CALLOUT: include-anchor-emit One `<div data-listing-tag="...">`
per `listings/` include, dropped just past the closing fence so the screenshot
tool can find the rendered `<pre>` via `previousElementSibling`.
            out.push_str(&format!(
                "<div data-listing-tag=\"{tag}\" aria-hidden=\"true\"></div>\n",
            ));
        }
        cursor = close_end;
    }
    out.push_str(&content[cursor..]);
    Ok(out)
}

fn line_number(content: &str, byte_offset: usize) -> usize {
    content[..byte_offset]
        .bytes()
        .filter(|&b| b == b'\n')
        .count()
        + 1
}

#[cfg(test)]
mod tests {
    use super::*;
    use tempfile::TempDir;

    #[test]
    fn parse_listing_includes_extracts_well_formed_directive() {
        let content = "Before.\n```rust\n{{#include listings/
foo.rs}}\n```\nAfter.\n";
        let got = parse_listing_includes(content);
        assert_eq!(got.len(), 1);
```

```
        assert_eq!(got[0].tag.as_deref(), Some("foo"));
        assert_eq!(got[0].rel_path, "listings/foo.rs");
    }

    #[test]
    fn parse_listing_includes_extracts_tag_as_file_stem() {
        let content = "```rust\n{{#include listings/some-tag-v3.rs}}\n```\n";
        let got = parse_listing_includes(content);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].tag.as_deref(), Some("some-tag-v3"));
    }

    #[test]
    fn parse_listing_includes_collects_snippets_with_no_tag() {
        let content = "```rust\n{{#include snippets/excerpt.rs}}\n```\n";
        let got = parse_listing_includes(content);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].tag, None);
        assert_eq!(got[0].rel_path, "snippets/excerpt.rs");
    }

    #[test]
    fn parse_listing_includes_skips_escaped_form() {
        let content = "Inline example: \{{#include listings/foo.rs}} should not
match.\n";
        assert!(parse_listing_includes(content).is_empty());
    }

    #[test]
    fn parse_listing_includes_skips_directive_inside_inline_code_span() {
        let content = "Prose discussing `{{#include listings/foo.rs}}` syntax.
\n";
        assert!(parse_listing_includes(content).is_empty());
    }

    #[test]
    fn parse_listing_includes_skips_unintercepted_paths_and_line_ranges() {
        let content = concat!(
            "```toml\n",
            "[package]
name = "mdbook-listings"
version = "0.1.0"
edition = "2024"
rust-version = "1.88"
license = "MIT"
description = "Managed code listings for mdbook: inline callouts, freezing, and
verification"
repository = "https://github.com/padamson/mdbook-listings"
categories = ["command-line-utilities", "text-processing"]
keywords = ["mdbook", "preprocessor", "documentation", "code-listing"]
exclude = ["book/"]

[dependencies]
anyhow = "1"
clap = { version = "4", features = ["derive"] }
```

```
mdbook-preprocessor = "0.5"
pulldown-cmark = { version = "0.13", default-features = false, features =
["html"] }
serde = { version = "1", features = ["derive"] }
serde_json = "1"
sha2 = "0.11"
similar = "3"
toml = "1.1"
toml_edit = "0.25"

[dev-dependencies]
assert_cmd = "2"
futures = "0.3"
pdf-extract = "0.10"
playwright-rs = { workspace = true }
playwright-rs-trace = { git = "https://github.com/padamson/playwright-rust",
branch = "main" }
predicates = "3"
tempfile = "3"
tokio = { workspace = true }
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }

# Workspace tools live alongside the crate but are not part of the published
# crate (each carries `publish = false`). Run with `cargo run -p <name>`.
[workspace]
members = ["tools/capture-screenshots"]

# Centralised cross-workspace deps. `playwright-rs` is dogfooded against the
# upstream main branch so this project rides ahead of the latest crates.io
# release (currently 0.12.3) and can adopt unreleased v0.13.0 features
# (richer tracing instrumentation, ARIA snapshots, screencast surface, etc.)
# before they ship. Cargo.lock pins the actual revision for reproducibility;
# bump with `cargo update -p playwright-rs`.
[workspace.dependencies]
playwright-rs = { git = "https://github.com/padamson/playwright-rust", branch =
"main" }
tokio = { version = "1", features = ["macros", "rt-multi-thread"] }\n",
            "```\n\n",
            "```rust\n",
            "{{#include listings/foo.rs:5:20}}\n",
            "```\n\n",
            "```rust\n",
            "{{#include snippets/foo.rs:setup}}\n",
            "```\n",
        );
        assert!(parse_listing_includes(content).is_empty());
    }

    #[test]
    fn parse_listing_includes_handles_subdirectory_path() {
        let content = "```rust\n{{#include listings/sub/foo.rs}}\n```\n";
        let got = parse_listing_includes(content);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].tag.as_deref(), Some("foo"));
```

```
        assert_eq!(got[0].rel_path, "listings/sub/foo.rs");
    }

    #[test]
    fn parse_listing_includes_records_fence_close_end_for_in_fence_directive() {
        let content = "```rust\n{{#include listings/foo.rs}}\n```\nafter\n";
        let got = parse_listing_includes(content);
        assert_eq!(got.len(), 1);
        assert!(got[0].fence_close_end.is_some());
    }

    #[test]
    fn
parse_listing_includes_records_no_fence_close_end_for_out_of_fence_directive() {
        let content = "Inline mention: {{#include listings/foo.rs}} not in
fence.\n";
        let got = parse_listing_includes(content);
        assert_eq!(got.len(), 1);
        assert_eq!(got[0].fence_close_end, None);
    }

    #[test]
    fn splice_chapter_replaces_directive_with_file_contents_and_emits_anchor() {
        let tmp = TempDir::new().unwrap();
        let src = tmp.path();
        std::fs::create_dir_all(src.join("listings")).unwrap();
        std::fs::write(src.join("listings/foo.rs"), "fn body() {}\n").unwrap();
        let content = "```rust\n{{#include listings/foo.rs}}\n```\n";
        let out = splice_chapter(content, src, None).expect("splice");
        assert!(out.contains("fn body() {}"), "got:\n{out}");
        assert!(!out.contains("{{#include"), "got:\n{out}");
        assert!(out.contains("data-listing-tag=\"foo\""), "got:\n{out}");
    }

    #[test]
    fn splice_chapter_emits_anchor_after_closing_fence_not_inside_block() {
        let tmp = TempDir::new().unwrap();
        let src = tmp.path();
        std::fs::create_dir_all(src.join("listings")).unwrap();
        std::fs::write(src.join("listings/foo.rs"), "fn body() {}\n").unwrap();
        let content = "```rust\n{{#include listings/foo.rs}}\n```\n";
        let out = splice_chapter(content, src, None).expect("splice");
        let anchor_pos = out.find("data-listing-tag").expect("anchor present");
        let close_fence_pos = out
            .find("```\n")
            .map(|p| p + 4)
            .expect("close fence present");
        assert!(anchor_pos > close_fence_pos, "got:\n{out}");
    }

    #[test]
    fn
splice_chapter_returns_listing_file_missing_with_chapter_line_for_absent_file()
{
        let tmp = TempDir::new().unwrap();
```

```
        let chapter = std::path::Path::new("ch99-foo.md");
        let content = "intro\n\n```rust\n{{#include listings/missing-
tag.rs}}\n```\n";
        let err = splice_chapter(content, tmp.path(),
Some(chapter)).expect_err("should fail");
        match err {
            SpliceError::ListingFileMissing {
                tag,
                line,
                chapter_path,
                ..
            } => {
                assert_eq!(tag, "missing-tag");
                assert_eq!(line, 4);
                assert_eq!(chapter_path.as_deref(), Some(chapter));
            }
            SpliceError::ListingIncludeOutsideFence { .. } => panic!("wrong
variant"),
        }
    }

    #[test]
    fn
splice_chapter_returns_listing_include_outside_fence_when_directive_has_no_enclosing_fence()
    {
        let chapter = std::path::Path::new("ch99-foo.md");
        let content = "Mid-paragraph: {{#include listings/foo.rs}} bare
directive.\n";
        let tmp = TempDir::new().unwrap();
        let err = splice_chapter(content, tmp.path(),
Some(chapter)).expect_err("should fail");
        match err {
            SpliceError::ListingIncludeOutsideFence {
                tag,
                line,
                chapter_path,
            } => {
                assert_eq!(tag, "foo");
                assert_eq!(line, 1);
                assert_eq!(chapter_path.as_deref(), Some(chapter));
            }
            SpliceError::ListingFileMissing { .. } => panic!("wrong variant"),
        }
    }

    #[test]
    fn splice_chapter_passes_through_unintercepted_includes_untouched() {
        let tmp = TempDir::new().unwrap();
        let content = concat!(
            "```toml\n",
            "[package]
name = "mdbook-listings"
version = "0.1.0"
edition = "2024"
rust-version = "1.88"
```

```
license = "MIT"
description = "Managed code listings for mdbook: inline callouts, freezing, and
verification"
repository = "https://github.com/padamson/mdbook-listings"
categories = ["command-line-utilities", "text-processing"]
keywords = ["mdbook", "preprocessor", "documentation", "code-listing"]
exclude = ["book/"]

[dependencies]
anyhow = "1"
clap = { version = "4", features = ["derive"] }
mdbook-preprocessor = "0.5"
pulldown-cmark = { version = "0.13", default-features = false, features =
["html"] }
serde = { version = "1", features = ["derive"] }
serde_json = "1"
sha2 = "0.11"
similar = "3"
toml = "1.1"
toml_edit = "0.25"

[dev-dependencies]
assert_cmd = "2"
futures = "0.3"
pdf-extract = "0.10"
playwright-rs = { workspace = true }
playwright-rs-trace = { git = "https://github.com/padamson/playwright-rust",
branch = "main" }
predicates = "3"
tempfile = "3"
tokio = { workspace = true }
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }

# Workspace tools live alongside the crate but are not part of the published
# crate (each carries `publish = false`). Run with `cargo run -p <name>`.
[workspace]
members = ["tools/capture-screenshots"]

# Centralised cross-workspace deps. `playwright-rs` is dogfooded against the
# upstream main branch so this project rides ahead of the latest crates.io
# release (currently 0.12.3) and can adopt unreleased v0.13.0 features
# (richer tracing instrumentation, ARIA snapshots, screencast surface, etc.)
# before they ship. Cargo.lock pins the actual revision for reproducibility;
# bump with `cargo update -p playwright-rs`.
[workspace.dependencies]
playwright-rs = { git = "https://github.com/padamson/playwright-rust", branch =
"main" }
tokio = { version = "1", features = ["macros", "rt-multi-thread"] }\n",
            "```\n\n",
            "```rust\n",
            "{{#include listings/foo.rs:5:20}}\n",
            "```\n",
        );
        let out = splice_chapter(content, tmp.path(), None).expect("splice");
```

```
        assert_eq!(out, content, "got:\n{out}");
    }

    #[test]
    fn splice_chapter_expands_snippet_include_without_emitting_anchor() {
        let tmp = TempDir::new().unwrap();
        let src = tmp.path();
        std::fs::create_dir_all(src.join("snippets")).unwrap();
        std::fs::write(src.join("snippets/excerpt.rs"), "fn snippet_body()
{}\n").unwrap();
        let content = "```rust\n{{#include snippets/excerpt.rs}}\n```\n";
        let out = splice_chapter(content, src, None).expect("splice");
        assert!(out.contains("fn snippet_body() {}"), "got:\n{out}");
        assert!(!out.contains("data-listing-tag"), "got:\n{out}");
        assert!(!out.contains("{{#include"), "got:\n{out}");
    }

    #[test]
    fn
splice_chapter_handles_two_includes_in_one_chapter_with_independent_anchors() {
        let tmp = TempDir::new().unwrap();
        let src = tmp.path();
        std::fs::create_dir_all(src.join("listings")).unwrap();
        std::fs::write(src.join("listings/foo.rs"), "fn body_one()
{}\n").unwrap();
        std::fs::write(src.join("listings/bar.rs"), "fn body_two()
{}\n").unwrap();
        let content = concat!(
            "```rust\n",
            "{{#include listings/foo.rs}}\n",
            "```\n\n",
            "```rust\n",
            "{{#include listings/bar.rs}}\n",
            "```\n",
        );
        let out = splice_chapter(content, src, None).expect("splice");
        assert!(out.contains("fn body_one() {}"));
        assert!(out.contains("fn body_two() {}"));
        assert!(out.contains("data-listing-tag=\"foo\""));
        assert!(out.contains("data-listing-tag=\"bar\""));
    }

    #[test]
    fn splice_chapter_appends_trailing_newline_when_file_lacks_one() {
        let tmp = TempDir::new().unwrap();
        let src = tmp.path();
        std::fs::create_dir_all(src.join("listings")).unwrap();
        std::fs::write(src.join("listings/foo.rs"), "fn body() {}").unwrap();
        let content = "```rust\n{{#include listings/foo.rs}}\n```\n";
        let out = splice_chapter(content, src, None).expect("splice");
        assert!(out.contains("fn body() {}\n```"), "got:\n{out}");
    }
}
```

[1] parse-entry — Two-pass scan: first collect fence body spans, then walk lines for `{{#include ...}}` directives. Skips backslash-escaped forms and directives inside inline code spans (chapter prose quotes the syntax verbatim).

[2] snippets-intercept — Two prefixes are intercepted: `listings/` (frozen tags — emit anchor) and `snippets/` (no anchor; we expand to give the callout splicer a shot at any CALLOUT markers in the snippet source). Other forms fall through to mdbook's built-in `links` preprocessor.

[3] tag-from-stem — Tag is the file stem of `listings/...` paths so `listings/sub/foo.rs` and `listings/foo.rs` produce the same anchor; subdirectory stem collisions would clash on the anchor, but the book has none today.

[4] include-splice-entry — The HTML splicer entry point. Walks every intercepted directive; replaces with file body and (for `listings/`) drops a `<div data-listing-tag>` locator anchor after the closing fence.

[5] include-anchor-emit — One `<div data-listing-tag="...">` per `listings/` include, dropped just past the closing fence so the screenshot tool can find the rendered `<pre>` via `previousElementSibling`.

`{{#include snippets/...}}` paths are also intercepted (callout **[2]**) — *without* an anchor — so the callout splicer downstream sees their `CALLOUT:` markers (otherwise mdbook's `links` would expand them after our callout pass and the markers would land in the rendered HTML with no overlay buttons). All other includes (`../some/path`, line-range syntax `:N:M`, anchor refs `:setup`) pass through untouched for `links` to handle.

The diff splicer's anchor also expands from the slice-7 single-attribute `data-listing-tag="RIGHT"` to the dual-attribute `data-listing-diff-left="LEFT" data-listing-diff-right="RIGHT"` (callout **[1]**), so the locator is unique even when several diffs share a right operand. Slice 8 also evolves the HTML splicer to process diff blocks for callouts: +-prefixed and -prefixed marker lines are stripped and emit a badge keyed to the line that previously held them; --prefixed marker lines are dropped silently with no badge — the callout is gone in the new state, so neither the comment nor a marker for it appears in the rendered diff:

```
--- diff-v7
+++ diff-v8
@@ -271,7 +271,12 @@
         out.push_str(&content[cursor..d.span.start]);
         out.push_str("```diff\n");
         out.push_str(&body);
-        out.push_str("```");
+        out.push_str("```\n");
+        // CALLOUT: diff-anchor-dual Locator anchor for the capture-screenshots
tool. Both operands are emitted as separate data-attributes so the tool can
locate a diff block by its (LEFT, RIGHT) pair — unique even when multiple diffs
share the same RIGHT tag, and unambiguous against the include splicer's `data-
listing-tag` anchors.
+        out.push_str(&format!(
+            "<div data-listing-diff-left=\"{}\" data-listing-diff-right=\"{}\"
aria-hidden=\"true\"></div>",
```

```
+            d.left, d.right,
+        ));
         cursor = d.span.end;
     }
     out.push_str(&content[cursor..]);
@@ -593,6 +598,14 @@
             !out.contains("{{#diff"),
             "directive should be consumed; got:\n{out}",
         );
+        assert!(
+            out.contains("data-listing-diff-left=\"left-tag\""),
+            "expected diff-left anchor attribute; got:\n{out}",
+        );
+        assert!(
+            out.contains("data-listing-diff-right=\"right-tag\""),
+            "expected diff-right anchor attribute; got:\n{out}",
+        );
     }

     #[test]
```

[1] diff-anchor-dual — Locator anchor for the capture-screenshots tool. Both operands are emitted as separate data-attributes so the tool can locate a diff block by its (LEFT, RIGHT) pair — unique even when multiple diffs share the same RIGHT tag, and unambiguous against the include splicer's `data-listing-tag` anchors.

The preprocessor wires the new include splicer into `preprocess()` as the first stage of a three-stage chain — includes → diffs → callouts (callout **[1]**). Order matters: the callout splicer needs included source bytes inline so it can parse `CALLOUT:` markers from them.

```
--- main-v8
+++ main-v9
@@ -6,6 +6,7 @@
 use mdbook_listings::callout::{SupportedRenderer, splice_chapter as
splice_callouts};
 use mdbook_listings::diff::splice_chapter as splice_diffs;
 use mdbook_listings::freeze::{FreezeOptions, FreezeOutcome, freeze};
+use mdbook_listings::include::splice_chapter as splice_includes;
 use mdbook_listings::install::{InstallOutcome, install};
 use mdbook_listings::manifest::Manifest;
 use mdbook_preprocessor::book::BookItem;
@@ -143,18 +144,27 @@
                 .and_then(|p| p.parent())
                 .map(|d| src_dir.join(d))
                 .unwrap_or_else(|| src_dir.clone());
-            match splice_diffs(
-                &chapter.content,
-                &manifest,
-                &ctx.root,
-                chapter.source_path.as_deref(),
```

```
-                &chapter_dir,
-            )
-            .map_err(|e| anyhow::Error::new(e).context("rendering {{#diff}}
directive failed"))
-            .and_then(|new_content| {
-                splice_callouts(&new_content, renderer)
-                    .map_err(|e| anyhow::Error::new(e).context("rendering
callouts failed"))
-            }) {
+            // CALLOUT: preprocessor-chain Three-stage chain per chapter:
includes (expand listings/snippets + drop locator anchors) → diffs (render
`{{#diff}}` blocks + emit dual-attribute anchors) → callouts (strip CALLOUT
comments + emit overlay). The order matters: callouts need the included source
bytes inline to find `CALLOUT:` markers.
+            match splice_includes(&chapter.content, &src_dir,
chapter.source_path.as_deref())
+                .map_err(|e| {
+                    anyhow::Error::new(e).context("expanding {{#include
listings/...}} failed")
+                })
+                .and_then(|new_content| {
+                    splice_diffs(
+                        &new_content,
+                        &manifest,
+                        &ctx.root,
+                        chapter.source_path.as_deref(),
+                        &chapter_dir,
+                    )
+                    .map_err(|e| {
+                        anyhow::Error::new(e).context("rendering {{#diff}}
directive failed")
+                    })
+                })
+                .and_then(|new_content| {
+                    splice_callouts(&new_content, renderer)
+                        .map_err(|e| anyhow::Error::new(e).context("rendering
callouts failed"))
+                }) {
                 Ok(new_content) => chapter.content = new_content,
                 Err(e) => splice_err = Some(e),
             }
```

[1] preprocessor-chain — Three-stage chain per chapter: includes (expand listings/snippets + drop locator anchors) → diffs (render `{{#diff}}` blocks + emit dual-attribute anchors) → callouts (strip CALLOUT comments + emit overlay). The order matters: callouts need the included source bytes inline to find `CALLOUT:` markers.

Five integration tests in `tests/includes.rs` exercise the new splicer end-to-end through the JSON envelope: directive replacement, anchor emission position, snippet expansion without anchor, both include and diff anchors emitted from one chapter, and the missing-file error path:

```rust
//! Integration tests for slice 8: `{{#include listings/...}}` interception
//! and the `<div data-listing-tag>` locator anchor the include splicer
//! emits after each frozen-listing fenced block.

use std::fs;
use std::path::PathBuf;

use mdbook_preprocessor::PreprocessorContext;
use mdbook_preprocessor::book::{Book, BookItem, Chapter};
use mdbook_preprocessor::config::Config;
use tempfile::TempDir;

mod common;
use common::mdbook_listings;

#[test]
fn listing_include_directive_is_replaced_with_file_contents_inline() {
    let book = MinimalIncludesBook::new();
    let envelope = book.envelope_with_chapter(
        "Before paragraph.\n\n```rust\n{{#include listings/
sample.rs}}\n```\n\nAfter paragraph.\n",
    );

    let returned = run_preprocessor(envelope);
    let content = chapter_content(&returned, "Include Test");

    assert!(
        content.contains("fn sample_body() {}"),
        "expected file body inline; got:\n{content}",
    );
    assert!(
        !content.contains("{{#include"),
        "directive should be consumed; got:\n{content}",
    );
}

#[test]
fn listing_include_emits_anchor_after_closing_fence() {
    let book = MinimalIncludesBook::new();
    let envelope = book
        .envelope_with_chapter("```rust\n{{#include listings/
sample.rs}}\n```\nAfter paragraph.\n");

    let returned = run_preprocessor(envelope);
    let content = chapter_content(&returned, "Include Test");

    assert!(
        content.contains("data-listing-tag=\"sample\""),
        "expected listing-tag anchor with file-stem tag; got:\n{content}",
    );
    let anchor_pos = content.find("data-listing-tag").expect("anchor present");
    let close_fence_pos = content
        .find("```\n")
        .map(|p| p + 4)
        .expect("close fence present");
```

```
    assert!(
        anchor_pos > close_fence_pos,
        "anchor must come AFTER the closing fence; anchor at {anchor_pos},
close-fence at {close_fence_pos}\ncontent:\n{content}",
    );
}

#[test]
fn snippet_include_is_expanded_inline_without_listing_tag_anchor() {
    let book = MinimalIncludesBook::new();
    book.write_snippet("excerpt.rs", "fn snippet_body() {}\n");
    let envelope = book.envelope_with_chapter("```rust\n{{#include snippets/
excerpt.rs}}\n```\n");

    let returned = run_preprocessor(envelope);
    let content = chapter_content(&returned, "Include Test");

    assert!(
        content.contains("fn snippet_body() {}"),
        "snippet should be expanded inline; got:\n{content}",
    );
    assert!(
        !content.contains("data-listing-tag"),
        "snippets must not produce a listing-tag anchor; got:\n{content}",
    );
    assert!(
        !content.contains("{{#include"),
        "directive should be consumed; got:\n{content}",
    );
}

#[test]
fn listing_include_followed_by_diff_emits_both_anchor_kinds() {
    let book = MinimalIncludesBook::new();
    let envelope = book.envelope_with_chapter(concat!(
        "First show as include.\n\n",
        "```rust\n{{#include listings/sample.rs}}\n```\n\n",
        "Then diff against new-tag.\n\n",
        "{{#diff sample new-tag}}\n",
    ));

    let returned = run_preprocessor(envelope);
    let content = chapter_content(&returned, "Include Test");

    assert!(
        content.contains("data-listing-tag=\"sample\""),
        "expected include-side listing-tag anchor; got:\n{content}",
    );
    assert!(
        content.contains("data-listing-diff-left=\"sample\"")
            && content.contains("data-listing-diff-right=\"new-tag\""),
        "expected diff-side dual-attribute anchor for the (sample, new-tag)
pair; got:\n{content}",
    );
}
```

```
#[test]
fn listing_include_with_missing_file_fails_with_chapter_path_in_diagnostic() {
    let book = MinimalIncludesBook::new();
    let envelope =
        book.envelope_with_chapter("intro\n\n```rust\n{{#include listings/
missing-tag.rs}}\n```\n");

    let stderr = mdbook_listings()
        .write_stdin(envelope)
        .assert()
        .failure()
        .get_output()
        .stderr
        .clone();
    let stderr = String::from_utf8_lossy(&stderr);

    assert!(
        stderr.contains("missing-tag"),
        "diagnostic should name the missing tag; got:\n{stderr}",
    );
    assert!(
        stderr.contains("expanding") || stderr.contains("include") ||
stderr.contains("missing"),
        "diagnostic should mention the include-expansion failure; got:
\n{stderr}",
    );
}

/// Pipes the envelope through the preprocessor binary and returns the
/// transformed `Book` parsed from stdout.
fn run_preprocessor(envelope: String) -> Book {
    let output = mdbook_listings()
        .write_stdin(envelope)
        .assert()
        .success()
        .get_output()
        .stdout
        .clone();
    serde_json::from_slice(&output).expect("parse stdout as Book")
}

/// Tempdir laid out as a real mdbook book root: a frozen listing under
/// `src/listings/sample.rs` and a `listings.toml` manifest registering
/// it. `MinimalIncludesBook::write_snippet` lays a snippet down on demand
/// so individual tests opt into the snippet path explicitly.
struct MinimalIncludesBook {
    _tmp: TempDir,
    root: PathBuf,
}

impl MinimalIncludesBook {
    fn new() -> Self {
        let tmp = TempDir::new().expect("tempdir");
        let root = tmp.path().to_path_buf();
```

```
        let listings_dir = root.join("src").join("listings");
        fs::create_dir_all(&listings_dir).unwrap();
        fs::write(listings_dir.join("sample.rs"), "fn sample_body()
{}\n").unwrap();
        fs::write(listings_dir.join("new-tag.rs"), "fn sample_body_v2()
{}\n").unwrap();

        fs::write(
            root.join("listings.toml"),
            "version = 1\n\n\
             [[listing]]\n\
             tag = \"sample\"\n\
             source = \"../sample.rs\"\n\
             frozen = \"src/listings/sample.rs\"\n\
             sha256 =
\"0000000000000000000000000000000000000000000000000000000000000000\"\n\n\
             [[listing]]\n\
             tag = \"new-tag\"\n\
             source = \"../new.rs\"\n\
             frozen = \"src/listings/new-tag.rs\"\n\
             sha256 =
\"0000000000000000000000000000000000000000000000000000000000000000\"\n",
        )
        .unwrap();

        Self { _tmp: tmp, root }
    }

    /// Write a file at `src/snippets/<rel>` so a chapter can reference it
    /// via `{{#include snippets/<rel>}}`.
    fn write_snippet(&self, rel: &str, content: &str) {
        let snippets_dir = self.root.join("src").join("snippets");
        fs::create_dir_all(&snippets_dir).unwrap();
        fs::write(snippets_dir.join(rel), content).unwrap();
    }

    /// Build the `[PreprocessorContext, Book]` JSON tuple mdbook would send,
    /// with one chapter carrying `chapter_content`.
    fn envelope_with_chapter(&self, chapter_content: &str) -> String {
        let ctx =
            PreprocessorContext::new(self.root.clone(), Config::default(),
"html".to_string());
        let chapter = Chapter::new(
            "Include Test",
            chapter_content.to_string(),
            "include-test.md",
            vec![],
        );
        let book = Book::new_with_items(vec![BookItem::Chapter(chapter)]);
        serde_json::to_string(&(&ctx, &book)).expect("serialize envelope")
    }
}

fn chapter_content(book: &Book, chapter_name: &str) -> String {
```

```
    book.iter()
        .find_map(|item| match item {
            BookItem::Chapter(ch) if ch.name == chapter_name =>
Some(ch.content.clone()),
            _ => None,
        })
        .unwrap_or_else(|| panic!("chapter `{chapter_name}` missing from
returned book"))
}
```

Tool — subcommand redesign. `tools/capture-screenshots/` becomes a clap `Subcommand` with `include LISTING` and `diff LEFT RIGHT` arms. Each subcommand discovers the chapter by scanning chapter `.md` files for the directive substring (`\{{#include listings/LISTING.` or `\{{#diff LEFT RIGHT`), navigates Chromium to the chapter HTML via playwright-rs, and locates the rendered `<pre>` via a single CSS selector (`[data-listing-tag="LISTING"]` or `[data-listing-diff-left="LEFT"][data-listing-diff-right="RIGHT"]`). The slice-7 callout-badge fallback is gone — anchors cover both shapes now. Default output paths are `book/src/images/<LISTING>.png` and `book/src/images/<LEFT>__to__<RIGHT>.png`. The subcommand is dispatched at callout **[2]**:

```
--- capture-screenshots-v2
+++ capture-screenshots-v3
@@ -1,54 +1,235 @@
-//! Capture an element-scoped screenshot of a rendered chapter and write the
-//! PNG to a known path. Used by ch. 4's slice-by-slice visual record so each
-//! slice's narrative can embed a snapshot of how the chapter rendered the
-//! day the slice shipped.
+//! Screenshot a named listing from the built book.
+//!
+//! Two subcommands match the two listing-rendering shapes the
+//! mdbook-listings preprocessor produces:
+//!
+//! ```text
+//! capture-screenshots include LISTING            # {{#include listings/
LISTING.ext}} block
+//! capture-screenshots diff LEFT RIGHT            # {{#diff LEFT RIGHT}} block
+//! ```
+//!
+//! Each subcommand:
+//!
+//! 1. Scans `book/src/*.md` for the chapter that references the
+//!    target tag(s).
+//! 2. Loads `book/build/html/<chapter-slug>.html`.
+//! 3. Locates the rendered `<pre>` via the locator anchor the
+//!    preprocessor emits — `[data-listing-tag]` for `include`,
+//!    `[data-listing-diff-left][data-listing-diff-right]` for `diff`.
+//! 4. Writes a PNG to `book/src/images/<default-name>.png` (or
+//!    `--out` if specified). Output defaults: `<LISTING>.png` for
+//!    include, `<LEFT>__to__<RIGHT>.png` for diff.

-use std::path::PathBuf;
+use std::path::{Path, PathBuf};
```

```
-use clap::Parser;
-use playwright_rs::Playwright;
+use clap::{Parser, Subcommand};
+use playwright_rs::{Playwright, locator};
+use tracing_subscriber::EnvFilter;
+
+const MANIFEST_DIR: &str = env!("CARGO_MANIFEST_DIR");

 #[derive(Parser)]
-#[command(version, about, long_about = None)]
+#[command(version, about = "Screenshot a named listing from the built book")]
 struct Cli {
-    /// Absolute path to the rendered chapter HTML to load.
-    #[arg(long)]
-    chapter_html: PathBuf,
+    // CALLOUT: cli-parse
+    #[command(subcommand)]
+    command: Command,

-    /// CSS selector for the element to screenshot.
-    #[arg(long)]
-    selector: String,
+    /// Root of the book workspace (directory containing book.toml).
+    /// Defaults to `book/` in the workspace root.
+    #[arg(long, global = true)]
+    book_root: Option<PathBuf>,

-    /// Zero-based index when the selector matches multiple elements.
-    /// Negative values count from the end (`-1` is the last match).
-    #[arg(long, default_value_t = 0)]
-    nth: i32,
+    /// Output PNG path. Defaults to `<book-root>/src/images/<derived-
name>.png`
+    /// where the derived name is `<LISTING>` for `include` and
+    /// `<LEFT>__to__<RIGHT>` for `diff`.
+    #[arg(long, global = true)]
+    out: Option<PathBuf>,
+}

-    /// Absolute path to write the PNG to. Parent directories are created.
-    #[arg(long)]
-    out: PathBuf,
+#[derive(Subcommand)]
+enum Command {
+    /// Screenshot a `{{#include listings/LISTING.ext}}` block.
+    Include {
+        /// Tag (file stem) of the listing to screenshot.
+        listing: String,
+    },
+
+    /// Screenshot a `{{#diff LEFT RIGHT}}` block.
+    Diff {
+        /// Left (old) operand of the diff directive.
+        left: String,
+        /// Right (new) operand of the diff directive.
```

```
+        right: String,
+    },
 }

 #[tokio::main]
 async fn main() -> Result<(), Box<dyn std::error::Error>> {
-    // CALLOUT: cli-parse
     let cli = Cli::parse();
-    if let Some(parent) = cli.out.parent() {
-        std::fs::create_dir_all(parent)?;
+
+    // playwright-rs (unreleased v0.13.0 from `padamson/playwright-rust` main)
+    // adds `#[tracing::instrument]` spans across its public async surface;
+    // wiring up tracing_subscriber here makes every `goto`, `evaluate_value`,
+    // `screenshot`, `browser.close`, etc. log a structured span. Default
+    // filter `info` keeps the top-level operations visible without
+    // descending into the per-RPC `debug` chatter; raise via
+    // `RUST_LOG=capture_screenshots=debug,playwright_rs=debug` when
+    // diagnosing locator or screenshot failures.
+    tracing_subscriber::fmt()
+        .with_env_filter(
+            EnvFilter::try_from_default_env().unwrap_or_else(|_|
EnvFilter::new("info")),
+        )
+        .with_target(true)
+        .compact()
+        .init();
+
+    let book_root = cli
+        .book_root
+        .unwrap_or_else(|| PathBuf::from(MANIFEST_DIR).join("../../book"))
+        .canonicalize()?;
+    let src_dir = book_root.join("src");
+    let html_dir = book_root.join("build").join("html");
+
+    // CALLOUT: subcommand-dispatch Each subcommand resolves into a `Job`
carrying the discovery substring (for chapter-md scanning), the CSS selector
(for the locator anchor in rendered HTML), the JavaScript that promotes the
preceding `<pre>` to `id="__capture_target__"`, and the resolved output path.
+    let job = Job::from(&cli.command, &cli.out, &src_dir);
+
+    // CALLOUT: discover Scans book/src/*.md for the chapter that includes or
diffs the tag(s).
+    let chapter_slug = find_chapter_for_pattern(&src_dir,
&job.discovery_pattern)
+        .ok_or_else(|| format!("no chapter contains `{}`",
job.discovery_pattern))?;
+    println!("→ chapter: {chapter_slug}");
+
+    let html_path = html_dir.join(format!("{chapter_slug}.html"));
+    if !html_path.exists() {
+        return Err(format!("chapter HTML not found: {}",
html_path.display()).into());
     }
-    let url = format!("file://{}", cli.chapter_html.display());
```

```
+    let url = format!("file://{}", html_path.display());

     let pw = Playwright::launch().await?;
     let browser = pw.chromium().launch().await?;
     let page = browser.new_page().await?;
     page.goto(&url, None).await?;

-    // CALLOUT: locator-pick `--nth` disambiguates when the selector matches
more than one element; zero-based, negative counts from the end.
-    let target = page.locator(&cli.selector).await.nth(cli.nth);
-    let png = target.screenshot(None).await?;
-    std::fs::write(&cli.out, png)?;
-    println!("✓ wrote {}", cli.out.display());
+    // mdbook's #menu-bar is position: sticky; element-scoped screenshots
+    // capture overlapping viewport content, so the header would otherwise
+    // appear on top of the listing. Demote it to static.
+    let _: String = page
+        .evaluate_value(
+            r#"(() => {
+                document.querySelectorAll('#menu-bar, .menu-bar')
+                    .forEach(el => el.style.position = 'static');
+                return 'ok';
+            })()"#,
+        )
+        .await?;
+
+    // CALLOUT: locate Sets id="__capture_target__" on the listing's <pre> via
JavaScript so a stable CSS selector can target it for the screenshot.
+    let result: String = page.evaluate_value(&job.locator_js).await?;
+    if result == "not-found" {
+        return Err(format!(
+            "could not locate `{}` in {chapter_slug}.html via selector `{}`",
+            job.discovery_pattern, job.css_selector,
+        )
+        .into());
+    }
+    println!("→ located via selector: {}", job.css_selector);
+
+    if let Some(parent) = job.out.parent() {
+        std::fs::create_dir_all(parent)?;
+    }
+
+    let png = page
+        .locator(locator!("#__capture_target__"))
+        .await
+        .screenshot(None)
+        .await?;
+    std::fs::write(&job.out, png)?;
+    println!("✓ wrote {}", job.out.display());

     browser.close().await?;
     Ok(())
 }
+
+/// All the per-subcommand inputs the rest of `main` needs in one place:
```

```
+/// the substring used to scan chapter `.md` files for the right chapter,
+/// the CSS selector that uniquely names the locator anchor in the rendered
+/// HTML, the JavaScript that promotes the preceding `<pre>` to
+/// `id="__capture_target__"`, and the resolved output path.
+struct Job {
+    discovery_pattern: String,
+    css_selector: String,
+    locator_js: String,
+    out: PathBuf,
+}
+
+impl Job {
+    fn from(cmd: &Command, out_override: &Option<PathBuf>, src_dir: &Path) ->
Self {
+        match cmd {
+            Command::Include { listing } => {
+                let css_selector = format!(r#"[data-listing-
tag="{listing}"]"#);
+                Job {
+                    discovery_pattern: format!("{{{{#include listings/
{listing}."),
+                    locator_js: locator_js_for(&css_selector),
+                    css_selector,
+                    out: out_override
+                        .clone()
+                        .unwrap_or_else(|| src_dir.join("images").join(format!
("{listing}.png"))),
+                }
+            }
+            Command::Diff { left, right } => {
+                let css_selector = format!(
+                    r#"[data-listing-diff-left="{left}"][data-listing-diff-
right="{right}"]"#
+                );
+                Job {
+                    discovery_pattern: format!("{{{{#diff {left} {right}"),
+                    locator_js: locator_js_for(&css_selector),
+                    css_selector,
+                    out: out_override.clone().unwrap_or_else(|| {
+                        src_dir
+                            .join("images")
+                            .join(format!("{left}__to__{right}.png"))
+                    }),
+                }
+            }
+        }
+    }
+}
+
+/// Walks back from the locator anchor to the most recent `<pre>` (skipping
+/// any `<div class="callout-overlay">` sibling between them) and tags it
+/// `id="__capture_target__"` so a stable CSS selector can drive the
+/// element-scoped screenshot.
+fn locator_js_for(css_selector: &str) -> String {
+    format!((
```

```
+        r#"(() => {{
+    const anchor = document.querySelector('{css_selector}');
+    if (!anchor) return 'not-found';
+    let el = anchor.previousElementSibling;
+    while (el && el.tagName !== 'PRE') el = el.previousElementSibling;
+    if (!el) return 'not-found';
+    el.setAttribute('id', '__capture_target__');
+    return 'located';
+}})()"#
+    )
+}
+
+/// Scans `src_dir` for a `.md` file containing `pattern` (a substring like
+/// `"{{#include listings/foo."` or `"{{#diff foo bar"`). Returns the
+/// chapter's filename stem (e.g. `ch05-render-inline-callouts`).
+fn find_chapter_for_pattern(src_dir: &Path, pattern: &str) -> Option<String> {
+    let entries = std::fs::read_dir(src_dir).ok()?;
+    for entry in entries {
+        let Ok(entry) = entry else { continue };
+        let path = entry.path();
+        if path.extension().and_then(|e| e.to_str()) != Some("md") {
+            continue;
+        }
+        let Ok(content) = std::fs::read_to_string(&path) else {
+            continue;
+        };
+        if content.contains(pattern) {
+            return path.file_stem().and_then(|s| s.to_str()).map(String::from);
+        }
+    }
+    None
+}
```

[1] cli-parse

[2] subcommand-dispatch — Each subcommand resolves into a `Job` carrying the discovery substring (for chapter-md scanning), the CSS selector (for the locator anchor in rendered HTML), the JavaScript that promotes the preceding `<pre>` to `id="__capture_target__"`, and the resolved output path.

[3] discover — Scans book/src/*.md for the chapter that includes or diffs the tag(s).

[4] locate — Sets id = “**capture_target**” on the listing’s via JavaScript so a stable CSS selector can target it for the screenshot.

The tool also dogfoods the unreleased v0.13.0 work in the `padamson/playwright-rust` repo: the workspace `Cargo.toml`’s `[workspace.dependencies]` table now sources `playwright-rs` as a git dep on `branch = "main"`, and the tool wires up `tracing_subscriber` so playwright-rs’s new `#[tracing::instrument]` spans (every `goto`, `evaluate_value`, `screenshot`, `browser.close`) log automatically once the upstream instrumentation merges. Local debugging gets richer for free with no per-callsite logging.

```
--- cargo-toml-v5
+++ cargo-toml-v6
@@ -8,6 +8,7 @@
 repository = "https://github.com/padamson/mdbook-listings"
 categories = ["command-line-utilities", "text-processing"]
 keywords = ["mdbook", "preprocessor", "documentation", "code-listing"]
+exclude = ["book/"]

 [dependencies]
 anyhow = "1"
@@ -23,12 +24,22 @@
 [dev-dependencies]
 assert_cmd = "2"
 pdf-extract = "0.9"
-playwright-rs = "0.12"
+playwright-rs = { workspace = true }
 predicates = "3"
 tempfile = "3"
-tokio = { version = "1", features = ["macros", "rt-multi-thread"] }
+tokio = { workspace = true }

 # Workspace tools live alongside the crate but are not part of the published
 # crate (each carries `publish = false`). Run with `cargo run -p <name>`.
 [workspace]
 members = ["tools/capture-screenshots"]
+
+# Centralised cross-workspace deps. `playwright-rs` is dogfooded against the
+# upstream main branch so this project rides ahead of the latest crates.io
+# release (currently 0.12.3) and can adopt unreleased v0.13.0 features
+# (richer tracing instrumentation, ARIA snapshots, screencast surface, etc.)
+# before they ship. Cargo.lock pins the actual revision for reproducibility;
+# bump with `cargo update -p playwright-rs`.
+[workspace.dependencies]
+playwright-rs = { git = "https://github.com/padamson/playwright-rust", branch =
"main" }
+tokio = { version = "1", features = ["macros", "rt-multi-thread"] }
```

Refactor (e2e migration) — `locator!()` macro and the assertion API

This refactor doesn't satisfy any chapter AC; it's pure test-quality work that takes advantage of two playwright-rs surfaces that landed on the upstream `padamson/playwright-rust` `main` branch and that slice 8 sourced via the workspace git dep:

- The **`playwright-rs-macros` crate** ships a `locator!(...)`

proc-macro that compile-time-validates Playwright selector strings for empty input, unbalanced brackets, and unknown engine prefixes. Adopted at every direct `page.locator(...)` call site (3 sites total — most of our locator usage lives inside JS strings fed to `evaluate_value`, which the proc-macro can't reach). Verified by deliberately introducing an unbalanced `[data-callout-badge` selector and observing `error: unclosed [` at compile time.

- The **`expect(locator).to_have_*()` assertion API** (added across

the v0.12.x line) auto-retries on flake, returns precise failure messages with line numbers, and reads more like the test's intent than the equivalent JS-blob `evaluate_value` returning a CSV. Migrating `tests/e2e_callouts.rs` from the slice-7/8-era JS-blob sweeps to locator

+ assertion calls is a substantial rewrite — every DOM query, every attribute check, every visibility assertion. Only one JS line remains (a `history.replaceState(...)` mutation in the click-through test, since that's a history-API operation with no playwright equivalent).

The full diff against `tests/e2e_callouts.rs` (v5 → v6) shows every JS-blob `evaluate_value` call replaced with a typed `page.locator(...).await` plus an `expect(...).to_have_*()` assertion (or a `Locator::nth(i)` iteration when the test sweeps multiple matches):

```
--- e2e-callouts-v5
+++ e2e-callouts-v6
@@ -1,6 +1,6 @@
 use std::path::PathBuf;

-use playwright_rs::{Playwright, locator};
+use playwright_rs::{Playwright, expect, locator};

 #[tokio::test]
 async fn label_only_callout_renders_badge_without_following_body() {
@@ -12,27 +12,16 @@
     let page = browser.new_page().await.expect("new page");
     page.goto(&url, None).await.expect("goto chapter");

-    // Slice 7+ shape: marker line stripped, badge is a <button> in the
-    // sibling overlay; label-only markers emit no body popover.
-    let body_present: String = page
-        .evaluate_value(
-            "(() => { \
-                const btn = document.querySelector('button[id=\"callout-cli-
parse\"]'); \
-                if (!btn) return 'NOT_FOUND'; \
-                const body = document.getElementById('callout-body-cli-parse');
\
-                return body ? 'YES' : 'NO'; \
-            })()",
-        )
+    let badge = page.locator(locator!("button#callout-cli-parse")).await;
+    expect(badge)
+        .to_have_count(1)
         .await
-        .expect("evaluate");
-    assert_ne!(
-        body_present, "NOT_FOUND",
-        "expected button#callout-cli-parse to exist on rendered ch. 4",
-    );
-    assert_eq!(
-        body_present, "NO",
-        "label-only callout must not have a body popover; got body presence:
{body_present}",
-    );
+        .expect("label-only badge button must exist");
+    let body = page.locator(locator!("#callout-body-cli-parse")).await;
+    expect(body)
+        .to_have_count(0)
+        .await
+        .expect("label-only callout must not have a body popover");
```

```
     browser.close().await.expect("close browser");
 }
@@ -47,16 +36,16 @@
     let page = browser.new_page().await.expect("new page");
     page.goto(&url, None).await.expect("goto chapter");

-    let badge = page.locator(locator!("[data-callout-badge]")).await;
-    let count = badge.count().await.expect("count badges");
+    let badges = page.locator(locator!("[data-callout-badge]")).await;
+    let count = badges.count().await.expect("count badges");
     assert!(
         count > 0,
-        "expected at least one [data-callout-badge] element on rendered ch. 4;
got 0",
+        "expected at least one [data-callout-badge]; got 0"
     );
-    let text = badge.first().text_content().await.expect("badge text");
+    let text = badges.first().text_content().await.expect("badge text");
     assert!(
         text.as_deref().is_some_and(|s| !s.trim().is_empty()),
-        "expected badge text to be non-empty; got {text:?}",
+        "expected first badge text to be non-empty; got {text:?}",
     );

     browser.close().await.expect("close browser");
@@ -72,30 +61,20 @@
     let page = browser.new_page().await.expect("new page");
     page.goto(&url, None).await.expect("goto chapter");

-    let href: String = page
-        .evaluate_value(
-            "(() => { \
-                const a = document.querySelector('a[data-callout-ref=\"cross-
ref-emit\"]'); \
-                return a ? a.getAttribute('href') : 'NOT_FOUND'; \
-            })()",
-        )
+    let cross_ref = page
+        .locator(locator!(r#"a[data-callout-ref="cross-ref-emit"]"#))
+        .await;
+    expect(cross_ref)
+        .to_have_attribute("href", "#callout-cross-ref-emit")
         .await
-        .expect("evaluate href");
-    assert_eq!(
-        href, "#callout-cross-ref-emit",
-        "expected prose-side cross-ref to point at listing badge anchor",
-    );
-
-    let target_present: String = page
-        .evaluate_value(
-            "(() => document.querySelector('button[id=\"callout-cross-ref-
emit\"]') ? 'YES' : 'NO')()",
-        )
```

```
+        .expect("cross-ref href must point at listing badge anchor");
+    let target = page
+        .locator(locator!("button#callout-cross-ref-emit"))
+        .await;
+    expect(target)
+        .to_have_count(1)
         .await
-        .expect("evaluate anchor presence");
-    assert_eq!(
-        target_present, "YES",
-        "expected listing-side button#callout-cross-ref-emit to exist as the
cross-ref's target",
-    );
+        .expect("listing-side badge button must exist as the cross-ref's
target");

     browser.close().await.expect("close browser");
 }
@@ -110,53 +89,32 @@
     let page = browser.new_page().await.expect("new page");
     page.goto(&url, None).await.expect("goto chapter");

-    // Pick a listing whose pre is known to contain a CALLOUT-bearing
-    // include: the cross-ref-emit marker is in the slice 6 snippet
-    // include, which the slice 7 splicer should have stripped from the
-    // rendered <pre>. Verify the literal "CALLOUT: cross-ref-emit"
-    // string is gone from that pre.
-    let pre_text: String = page
-        .evaluate_value(
-            "(() => { \
-                const btn = document.querySelector('button[id=\"callout-cross-
ref-emit\"]'); \
-                if (!btn) return 'NO_BTN'; \
-                const overlay = btn.closest('.callout-overlay'); \
-                const pre = overlay && overlay.previousElementSibling; \
-                return pre ? pre.textContent : 'NO_PRE'; \
-            })()",
-        )
+    // Find the <pre> whose sibling overlay carries the cross-ref-emit
+    // badge (xpath does the sibling traversal that CSS can't). The
+    // splicer should have stripped the literal marker comment from
+    // that pre's text.
+    let pre = page
+        .locator(locator!(
+            r#"xpath=//pre[following-sibling::div[1][.//button[@id="callout-
cross-ref-emit"]]]"#
+        ))
         .await
-        .expect("evaluate pre text");
-    assert!(
-        !pre_text.contains("CALLOUT: cross-ref-emit"),
```

```
-        "expected marker comment line to be stripped from the include's <pre>;
\
-         got pre.textContent containing the marker:\n{pre_text}",
-    );
+        .expect("marker comment line must be stripped from the include's
<pre>");

-    // The body popover starts hidden and becomes visible after hovering
-    // its triggering badge. Use Playwright's hover().
+    // Body popover starts hidden and becomes visible after hovering its
+    // triggering badge.
     let badge = page
-        .locator(locator!(
-            "button[id=\"callout-body-emit-source\"], button[id=\"callout-
cross-ref-emit\"]"
-        ))
+        .locator(locator!("button#callout-cross-ref-emit"))
         .await;
-    badge.first().hover(None).await.expect("hover badge");
-
-    let body_visible: String = page
-        .evaluate_value(
-            "(() => { \
-                const body = document.getElementById('callout-body-cross-ref-
emit'); \
-                if (!body) return 'NO_BODY'; \
-                const cs = window.getComputedStyle(body); \
-                return cs.visibility === 'hidden' ? 'HIDDEN' : 'VISIBLE'; \
-            })()",
-        )
+    badge.hover(None).await.expect("hover badge");
+    let body = page.locator(locator!("#callout-body-cross-ref-emit")).await;
+    expect(body)
+        .to_be_visible()
         .await
-        .expect("evaluate body visibility");
-    assert_eq!(
-        body_visible, "VISIBLE",
-        "expected body popover to become visible after hovering badge; got:
{body_visible}",
-    );
+        .expect("body popover must become visible after hovering its badge");

     browser.close().await.expect("close browser");
 }
@@ -166,20 +124,12 @@
     // Sweep guard for prose-side cross-refs: every `{{#callout LABEL}}`
     // directive renders as an `<a class="callout-badge callout-ref"
     // href="#callout-LABEL" data-callout-ref="LABEL"
-    // data-callout-ordinal="N">N</a>`. For each one in the chapter, we
-    // verify:
-    //
-    // 1. `href` matches `#callout-<data-callout-ref>` (no stale
-    //    label-to-href drift)
-    // 2. A `button[id="callout-LABEL"]` actually exists as the target
```

```
-    //    (catches refs to labels whose only occurrence is in a `{{#diff}}`
-    //    block, since the HTML splicer skips diffs for badge emission)
+    // data-callout-ordinal="N">N</a>`. For each one we verify (via
+    // playwright assertions, not JS-string sweeps) that:
+    // 1. `href` matches `#callout-<data-callout-ref>`
+    // 2. A `button[id="callout-LABEL"]` exists as the target
     // 3. The ref's `data-callout-ordinal` matches the target badge's
-    //    `data-callout-ordinal` (catches numbering drift between the
-    //    label-to-ordinal map's first-occurrence pass and the per-listing
-    //    badge emission pass)
-    // 4. The rendered text on the ref matches the target badge's rendered
-    //    text (the visible numeral readers see)
+    // 4. The rendered text on the ref matches the target badge's text
     let chapter_html = chapter_path();
     let url = format!("file://{}", chapter_html.display());

@@ -188,69 +138,76 @@
     let page = browser.new_page().await.expect("new page");
     page.goto(&url, None).await.expect("goto chapter");

-    let issues: String = page
-        .evaluate_value(
-            r#"(() => {
-                const refs = Array.from(document.querySelectorAll('a[data-
callout-ref]'));
-                if (refs.length === 0) {
-                    return 'NO_REFS_FOUND_AT_ALL';
-                }
-                const issues = [];
-                refs.forEach(a => {
-                    const label = a.getAttribute('data-callout-ref');
-                    const refOrdinal = a.getAttribute('data-callout-ordinal');
-                    const refText = (a.textContent || '').trim();
-                    const expectedHref = '#callout-' + label;
-                    const actualHref = a.getAttribute('href');
-                    if (actualHref !== expectedHref) {
-                        issues.push(`label "${label}": href="${actualHref}" but
expected "${expectedHref}"`);
-                    }
-                    const target = document.querySelector('button[id="callout-'
+ label + '"]');
-                    if (!target) {
-                        issues.push(`label "${label}": no badge button#callout-
${label} exists as the cross-ref target`);
-                        return;
-                    }
-                    const targetOrdinal = target.getAttribute('data-callout-
ordinal');
-                    const targetText = (target.textContent || '').trim();
-                    if (refOrdinal !== targetOrdinal) {
-                        issues.push(`label "${label}": ref data-callout-
ordinal="${refOrdinal}" but target badge data-callout-
ordinal="${targetOrdinal}"`);
-                    }
-                    if (refText !== targetText) {
```

```
-                        issues.push(`label "${label}": ref text "${refText}"
but target badge text "${targetText}"`);
-                    }
-                });
-                return issues.join('\n');
-            })()"#,
-        )
-        .await
-        .expect("evaluate cross-ref sweep");
-
-    assert_ne!(
-        issues, "NO_REFS_FOUND_AT_ALL",
-        "expected at least one a[data-callout-ref] in the chapter; the chapter
has no cross-refs to sweep",
-    );
+    let refs = page.locator(locator!("a[data-callout-ref]")).await;
+    let count = refs.count().await.expect("count refs");
     assert!(
-        issues.is_empty(),
-        "callout cross-ref sweep found broken refs in the chapter:\n{issues}",
+        count > 0,
+        "expected at least one a[data-callout-ref] in chapter"
     );

+    for i in 0..count {
+        let r = refs.nth(i as i32);
+        let label = r
+            .get_attribute("data-callout-ref")
+            .await
+            .expect("ref label")
+            .unwrap_or_default();
+        assert!(!label.is_empty(), "ref #{i} has empty data-callout-ref");
+
+        let expected_href = format!("#callout-{label}");
+        expect(r.clone())
+            .to_have_attribute("href", &expected_href)
+            .await
+            .unwrap_or_else(|e| panic!("ref `{label}`: href mismatch: {e:?}"));
+
+        let target = page
+            .locator(&format!(r#"button[id="callout-{label}"]"#))
+            .await;
+        expect(target.clone())
+            .to_have_count(1)
+            .await
+            .unwrap_or_else(|e| panic!("ref `{label}`: target badge missing:
{e:?}"));
+
+        let ref_ordinal = r
+            .get_attribute("data-callout-ordinal")
+            .await
+            .expect("ref ordinal")
+            .unwrap_or_default();
+        expect(target.clone())
+            .to_have_attribute("data-callout-ordinal", &ref_ordinal)
```

```
+            .await
+            .unwrap_or_else(|e| {
+                panic!("ref `{label}`: ordinal mismatch (ref={ref_ordinal}):
{e:?}")
+            });
+
+        let ref_text = r
+            .text_content()
+            .await
+            .expect("ref text")
+            .unwrap_or_default()
+            .trim()
+            .to_string();
+        expect(target)
+            .to_have_text(&ref_text)
+            .await
+            .unwrap_or_else(|e| {
+                panic!("ref `{label}`: rendered text mismatch
(ref=\"{ref_text}\"): {e:?}")
+            });
+    }
+
     browser.close().await.expect("close browser");
 }

 #[tokio::test]
 async fn every_cross_refed_label_has_a_visible_badge_in_the_chapter() {
     // Regression guard, scoped to labels the author actually points at:
-    // every `{{#callout LABEL}}` directive in chapter prose renders an
-    // `<a data-callout-ref="LABEL">`, and the matching badge
-    // `button[id="callout-LABEL"]` must exist somewhere in the rendered
-    // page. Catches the most common slice-shipping mistake — a cross-ref
-    // to a marker whose only chapter occurrence is in a `{{#diff}}` block
-    // (which the HTML splicer skips for badge emission, by design — the
-    // canonical anchor lives on a non-diff include). Test-fixture
-    // marker strings inside string literals (e.g. `// CALLOUT: greeting`
-    // inside Rust unit-test source) are intentionally not flagged: the
-    // author isn't pointing at them, so they're not part of the chapter's
-    // user-facing cross-reference contract.
+    // every `{{#callout LABEL}}` directive must have a corresponding
+    // `button[id="callout-LABEL"]` somewhere in the rendered page.
+    // Catches the most common slice-shipping mistake — a cross-ref to
+    // a marker whose only chapter occurrence is in a `{{#diff}}` block
+    // before slice 8 wired diff blocks through the badge emitter.
+    // Test-fixture marker strings inside string literals are
+    // intentionally not flagged (the author isn't pointing at them).
     let chapter_html = chapter_path();
     let url = format!("file://{}", chapter_html.display());

@@ -259,36 +216,38 @@
     let page = browser.new_page().await.expect("new page");
     page.goto(&url, None).await.expect("goto chapter");

-    // For every `a[data-callout-ref]` (the prose-side cross-ref shape),
-    // collect the label and check that a matching `button[id="callout-LABEL"]`
```

```
-    // exists. Returns a comma-separated list of labels that have a cross-ref
-    // pointing at them but no badge target.
-    let unmatched: String = page
-        .evaluate_value(
-            r#"(() => {
-                const refs = Array.from(document.querySelectorAll('a[data-
callout-ref]'));
-                const missing = new Set();
-                refs.forEach(a => {
-                    const label = a.getAttribute('data-callout-ref');
-                    if (!label) return;
-                    if (!document.querySelector('button[id="callout-' + label +
'"]')) {
-                        missing.add(label);
-                    }
-                });
-                return Array.from(missing).sort().join(',');
-            })()"#,
-        )
-        .await
-        .expect("evaluate cross-ref-target scan");
+    let refs = page.locator(locator!("a[data-callout-ref]")).await;
+    let count = refs.count().await.expect("count refs");
+
+    let mut missing: Vec<String> = Vec::new();
+    for i in 0..count {
+        let label = refs
+            .nth(i as i32)
+            .get_attribute("data-callout-ref")
+            .await
+            .expect("ref label")
+            .unwrap_or_default();
+        if label.is_empty() {
+            continue;
+        }
+        let target = page
+            .locator(&format!(r#"button[id="callout-{label}"]"#))
+            .await;
+        if target.count().await.expect("count target") == 0 {
+            missing.push(label);
+        }
+    }
+    missing.sort();
+    missing.dedup();

     assert!(
-        unmatched.is_empty(),
+        missing.is_empty(),
         "the following labels are cross-refed in chapter prose but have no \
-         `button[id=\"callout-LABEL\"]` target in the rendered chapter — \
-         most likely the cross-ref points at a marker whose only occurrence \
-         is in a `{{{{#diff}}}}` block (which the HTML splicer skips for \
-         badge emission). Add a non-diff `{{{{#include}}}}` of the source, \
-         or extract a snippet, so the badge anchor lands. Broken labels: 
{unmatched}",
```

```
+         `button[id=\"callout-LABEL\"]` target — most likely the cross-ref \
+         points at a marker whose only occurrence is in a `{{{{#diff}}}}` \
+         block. Add a non-diff `{{{{#include}}}}` of the source, or extract \
+         a snippet, so the badge anchor lands. Broken labels: {}",
+        missing.join(", "),
     );

     browser.close().await.expect("close browser");
@@ -297,17 +256,11 @@
 #[tokio::test]
 async fn clicking_each_cross_ref_scrolls_target_badge_into_viewport() {
     // End-to-end click-through guard: for every prose-side
-    // `a[data-callout-ref]` in the chapter, click it and assert that the
-    // target `button[id="callout-LABEL"]` ends up visible in the viewport
-    // (the natural in-page anchor-jump behaviour). Catches the case where
-    // the structural attributes line up (covered by the sweep tests
-    // above) but the link doesn't actually navigate — e.g. the target
-    // anchor is on an off-screen element with `display: none`, or the
-    // chapter-internal hash routing was broken by a future theme change.
+    // `a[data-callout-ref]`, click it and assert the target badge ends
+    // up visible (the natural in-page anchor-jump behaviour).
     //
-    // The rebuild discipline for cross-chapter refs is out of scope:
-    // chapter prose only references callouts in listings rendered in the
-    // same chapter, by design.
+    // Cross-chapter refs are out of scope: chapter prose only references
+    // callouts in listings rendered in the same chapter, by design.
     let chapter_html = chapter_path();
     let url = format!("file://{}", chapter_html.display());

@@ -316,27 +269,31 @@
     let page = browser.new_page().await.expect("new page");
     page.goto(&url, None).await.expect("goto chapter");

-    // Collect every cross-ref label first, in document order.
-    let labels_csv: String = page
-        .evaluate_value(
-            r#"() => Array.from(document.querySelectorAll('a[data-callout-
ref]'))
-                .map(a => a.getAttribute('data-callout-ref'))
-                .filter(Boolean)
-                .join(','))()"#,
-        )
-        .await
-        .expect("collect cross-ref labels");
-    let labels: Vec<&str> = labels_csv.split(',').filter(|s| !
s.is_empty()).collect();
+    let refs = page.locator(locator!("a[data-callout-ref]")).await;
+    let count = refs.count().await.expect("count refs");
     assert!(
-        !labels.is_empty(),
-        "expected at least one a[data-callout-ref] in the chapter for click-
through coverage",
+        count > 0,
+        "expected at least one cross-ref for click-through coverage"
```

```
     );

+    let mut labels: Vec<String> = Vec::with_capacity(count);
+    for i in 0..count {
+        if let Some(label) = refs
+            .nth(i as i32)
+            .get_attribute("data-callout-ref")
+            .await
+            .expect("ref label")
+            && !label.is_empty()
+        {
+            labels.push(label);
+        }
+    }
+
     let mut failures: Vec<String> = Vec::new();
     for label in &labels {
         // Reset the URL hash so each navigation is a fresh jump rather
-        // than a no-op when clicking a ref that already happens to point
-        // at the current hash.
+        // than a no-op when the current hash already matches. This is a
+        // history-API mutation; no playwright equivalent.
         let _: String = page
             .evaluate_value(
                 "(() => { history.replaceState(null, '', location.pathname);
return 'ok'; })()",
@@ -344,44 +301,37 @@
             .await
             .expect("reset hash");

-        // Click the ref. Use evaluate to drive the click + scroll
-        // synchronously so we don't need a brittle wait-for-scroll dance.
-        let after_click: String = page
-            .evaluate_value(&format!(
-                r##"(() => {{
-                    const a = document.querySelector('a[data-callout-
ref="{label}"]');
-                    if (!a) return 'NO_REF';
-                    a.click();
-                    const target = document.querySelector('button[id="callout-
{label}"]');
-                    if (!target) return 'NO_TARGET';
-                    // Match the in-page anchor-jump semantics: scroll the
-                    // target into view (the click on an `<a href="#...">`
-                    // already does this, but force the layout settle).
-                    target.scrollIntoView({{ block: 'center' }});
-                    const r = target.getBoundingClientRect();
-                    const inViewport = r.bottom > 0
-                        && r.top < (window.innerHeight ||
document.documentElement.clientHeight)
-                        && r.right > 0
-                        && r.left < (window.innerWidth ||
document.documentElement.clientWidth);
-                    const hash = location.hash;
-                    return inViewport
```

```
-                       ? 'OK:' + hash
-                       : 'OFFSCREEN:hash=' + hash + ' rect=' +
JSON.stringify(r);
-               }})()"##
-           ))
+       let r = page
+           .locator(&format!(r#"a[data-callout-ref="{label}"]"#))
+           .await
+           .first();
+       if let Err(e) = r.click(None).await {
+           failures.push(format!("label `{label}`: click failed: {e:?}"));
+           continue;
+       }
+
+       let target = page
+           .locator(&format!(r#"button[id="callout-{label}"]"#))
            .await
-           .expect("click cross-ref");
+           .first();
+       if let Err(e) = target.scroll_into_view_if_needed().await {
+           failures.push(format!("label `{label}`: scroll failed: {e:?}"));
+           continue;
+       }
+       if !target.is_visible().await.unwrap_or(false) {
+           failures.push(format!("label `{label}`: target not visible after
click"));
+           continue;
+       }

-       if !after_click.starts_with("OK:") {
-           failures.push(format!("label `{label}`: {after_click}"));
-       } else {
-           let expected_hash = format!("#callout-{label}");
-           let actual_hash = after_click.trim_start_matches("OK:");
-           if actual_hash != expected_hash {
-               failures.push(format!(
-                   "label `{label}`: hash after click was `{actual_hash}` but
expected `{expected_hash}`"
-               ));
-           }
+       let actual_hash: String = page
+           .evaluate_value("location.hash")
+           .await
+           .expect("read hash");
+       let expected_hash = format!("#callout-{label}");
+       if actual_hash != expected_hash {
+           failures.push(format!(
+               "label `{label}`: hash after click was `{actual_hash}` but
expected `{expected_hash}`"
+           ));
        }
    }
```

The migration surfaced a real slice-8 splicer bug. playwright-rs's strict-mode locator refused to resolve `#callout-body-cross-ref-emit` because the rendered chapter contained TWO `<div>` elements with that id — one from the snippet `{{#include}}` of `callout-pdf-emit-snippet-v2.rs`, one from the diff `+`-line marker addition slice 8 wired badge emission for. The button id was already dedup'd via the existing `emitted_anchor: HashSet<String>`, but the body div's id was not. Fix in `src/callout.rs`: lockstep dedup of the body div's `id` and the button's `aria-describedby` against the same `is_first_occurrence` boolean — callout **[2]**. The diff against `src/callout.rs` (v5 → v6) shows the splicer change:

```
--- callout-v5
+++ callout-v6
@@ -179,16 +179,23 @@
     let mut cursor = 0;
     let mut emitted_anchor: HashSet<String> = HashSet::new();
     for_each_fenced_block_with_span(content, |info, block_text, body_start,
close_end| {
-        if info == "diff" {
+        let callouts = callouts_for_block(info, block_text);
+        let is_diff = info == "diff";
+        // Diff blocks always go through the strip pass even when no `+`/` `
+        // callouts exist — `-`-side markers still need to be dropped from
+        // the rendered body.
+        if callouts.is_empty() && !is_diff {
             return;
         }
-        let callouts = callouts_for_block(info, block_text);
-        if callouts.is_empty() {
+        let (rewritten_body, post_strip_lines, total_lines) = if is_diff {
+            strip_marker_lines_diff(block_text)
+        } else {
+            strip_marker_lines(block_text, info)
+        };
+        if is_diff && callouts.is_empty() && rewritten_body == block_text {
+            // No-op diff: no markers of any kind to rewrite.
             return;
         }
-        let (rewritten_body, post_strip_lines) = strip_marker_lines(block_text,
info);
-        // Find the opening fence line to copy verbatim, then the rewritten
-        // body, then the closing fence line that follows.
         let pre_fence = &content[cursor..body_start];
         let close_fence_line = closing_fence_text(content, close_end);
         out.push_str(pre_fence);
@@ -201,6 +208,7 @@
         out.push_str(&render_callout_overlay_html(
             &callouts,
             &post_strip_lines,
+            total_lines,
             &mut emitted_anchor,
         ));
         out.push('\n');
@@ -214,7 +222,7 @@
  /// post-strip 1-based line numbers each marker now lands on (i.e. the
  /// line that took its place after the strip — typically the next non-
```

```
 /// marker code line).
-fn strip_marker_lines(block_text: &str, info: &str) -> (String, Vec<usize>) {
+fn strip_marker_lines(block_text: &str, info: &str) -> (String, Vec<usize>,
usize) {
     let prefix = comment_prefix_for_language(info);
     let lines: Vec<&str> = block_text.split_inclusive('\n').collect();
     let mut out = String::with_capacity(block_text.len());
@@ -237,9 +245,56 @@
             emitted_count += 1;
         }
     }
-    (out, post_strip_lines)
+    (out, post_strip_lines, emitted_count)
 }

+// CALLOUT: strip-diff Diff-aware strip: drop `-`-prefixed marker lines
entirely (no badge — the callout is gone in the new state); strip `+`-prefixed
and ` `-prefixed marker lines and record post-strip positions so badges land on
the line that previously held them in the diff's right-hand side.
+fn strip_marker_lines_diff(block_text: &str) -> (String, Vec<usize>, usize) {
+    let lines: Vec<&str> = block_text.split_inclusive('\n').collect();
+    let mut out = String::with_capacity(block_text.len());
+    let mut post_strip_lines: Vec<usize> = Vec::new();
+    let mut emitted_count: usize = 0;
+    for raw_line in lines {
+        let line_no_newline = raw_line.strip_suffix('\n').unwrap_or(raw_line);
+        // Diff metadata lines pass through unchanged.
+        if line_no_newline.starts_with("---")
+            || line_no_newline.starts_with("+++")
+            || line_no_newline.starts_with("@@")
+            || line_no_newline.starts_with('\\')
+        {
+            out.push_str(raw_line);
+            emitted_count += 1;
+            continue;
+        }
+        // Identify the diff-line prefix and try to parse the trailing
+        // payload as a marker against any known comment prefix.
+        let (prefix_char, payload) = if let Some(rest) =
line_no_newline.strip_prefix('+') {
+            (Some('+'), rest)
+        } else if let Some(rest) = line_no_newline.strip_prefix('-') {
+            (Some('-'), rest)
+        } else if let Some(rest) = line_no_newline.strip_prefix(' ') {
+            (Some(' '), rest)
+        } else {
+            (None, line_no_newline)
+        };
+        let is_marker = ALL_COMMENT_PREFIXES
+            .iter()
+            .any(|p| parse_line(payload, p, 0).is_some());
+        if is_marker {
+            // `+` and ` ` markers: strip the line, record post-strip position
+            // for badge placement. `-` markers: drop silently.
+            if matches!(prefix_char, Some('+') | Some(' ')) {
```

```
+                let target = (emitted_count + 1).max(1);
+                post_strip_lines.push(target);
+            }
+        } else {
+            out.push_str(raw_line);
+            emitted_count += 1;
+        }
+    }
+    (out, post_strip_lines, emitted_count)
+}
+
 fn closing_fence_text(content: &str, close_end: usize) -> &str {
     // close_end is one past the trailing newline of the closing fence
     // (or equal to bytes.len() if the file ends without a trailing newline).
@@ -276,7 +331,7 @@
     out
 }

-fn for_each_fenced_block_with_span<F>(content: &str, mut visit: F)
+pub(crate) fn for_each_fenced_block_with_span<F>(content: &str, mut visit: F)
 where
     F: FnMut(&str, &str, usize, usize),
 {
@@ -523,6 +578,7 @@
 fn render_callout_overlay_html(
     callouts: &[Callout],
     post_strip_lines: &[usize],
+    total_lines: usize,
     emitted_anchor: &mut HashSet<String>,
 ) -> String {
     let mut s = String::new();
@@ -531,24 +587,46 @@
         let ordinal = idx + 1;
         let label_esc = html_escape(&c.label);
         let line = post_strip_lines.get(idx).copied().unwrap_or(1);
-        let id_attr = if emitted_anchor.insert(c.label.clone()) {
+        // CALLOUT: body-id-dedup The button id, body div id, and the button's
`aria-describedby` are all dedup'd in lockstep on the first occurrence per
label. Without lockstep dedup, the same label appearing in two blocks (e.g. an
include and a diff `+` line, both processed for badges in slice 8) would emit
duplicate body div ids — invalid HTML, rejected by playwright-rs's strict-mode
locator.
+        let is_first_occurrence = emitted_anchor.insert(c.label.clone());
+        let id_attr = if is_first_occurrence {
             format!(" id=\"callout-{label_esc}\"")
         } else {
             String::new()
         };
+        // The body div's `id` and the button's `aria-describedby` are
+        // dedup'd identically: only the first occurrence per label gets
+        // them. Subsequent occurrences still hover-reveal (CSS uses the
+        // adjacent-sibling combinator inside .callout-entry, not the id),
+        // but cannot be cross-referenced from prose — by design, since
+        // `{{#callout LABEL}}` resolves to the canonical first-occurrence
+        // anchor.
```

```
+        let body_id_attr = if is_first_occurrence {
+            format!(" id=\"callout-body-{label_esc}\"")
+        } else {
+            String::new()
+        };
+        let aria_describedby_attr = if is_first_occurrence {
+            format!(" aria-describedby=\"callout-body-{label_esc}\"")
+        } else {
+            String::new()
+        };
         s.push_str(&format!(
-            "  <button type=\"button\" class=\"callout-badge\"{id_attr} \
-             data-callout-badge=\"{label_esc}\" data-callout-
ordinal=\"{ordinal}\" \
-             data-callout-line=\"{line}\" \
-             aria-describedby=\"callout-body-{label_esc}\" \
-             style=\"--callout-line: {line};\">{ordinal}</button>\n",
+            "  <div class=\"callout-entry\" data-callout-line=\"{line}\" \
+             style=\"--callout-line: {line}; --callout-listing-lines:
{total_lines};\">\n",
         ));
+        s.push_str(&format!(
+            "    <button type=\"button\" class=\"callout-badge\"{id_attr} \
+             data-callout-badge=\"{label_esc}\" data-callout-
ordinal=\"{ordinal}\"\
+             {aria_describedby_attr}>{ordinal}</button>\n",
+        ));
         if let Some(body) = &c.body {
             s.push_str(&format!(
-                "  <div class=\"callout-body\" id=\"callout-body-{label_esc}\"
role=\"tooltip\" hidden>{}</div>\n",
+                "    <div class=\"callout-body\"{body_id_attr}
role=\"tooltip\">{}</div>\n",
                 html_escape(body),
             ));
         }
+        s.push_str("  </div>\n");
     }
     s.push_str("</div>");
     s
@@ -814,6 +892,134 @@
     }

     #[test]
+    fn
splice_chapter_html_strips_added_marker_lines_from_diff_and_emits_badge() {
+        let content = concat!(
+            "```diff\n",
+            "--- a-tag\n",
+            "+++ b-tag\n",
+            "@@ -1,1 +1,2 @@\n",
+            " fn unchanged() {}\n",
+            "+// CALLOUT: added-marker Body for an added marker.\n",
+            "+fn added() {}\n",
+            "```\n",
```

```
+        );
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        assert!(
+            !out.contains("// CALLOUT: added-marker"),
+            "added marker comment line should be stripped from rendered diff;
got:\n{out}",
+        );
+        assert!(
+            out.contains("data-callout-badge=\"added-marker\""),
+            "expected badge for the added marker; got:\n{out}",
+        );
+        assert!(
+            out.contains("+fn added() {}"),
+            "non-marker `+` line should survive; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn
splice_chapter_html_strips_context_marker_lines_from_diff_and_emits_badge() {
+        let content = concat!(
+            "```diff\n",
+            "--- a-tag\n",
+            "+++ b-tag\n",
+            "@@ -1,2 +1,2 @@\n",
+            " // CALLOUT: kept-marker A marker carried over unchanged.\n",
+            " fn carried() {}\n",
+            "```\n",
+        );
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        assert!(
+            !out.contains("// CALLOUT: kept-marker"),
+            "context marker comment line should be stripped; got:\n{out}",
+        );
+        assert!(
+            out.contains("data-callout-badge=\"kept-marker\""),
+            "expected badge for the carried-over marker; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_html_drops_removed_marker_lines_from_diff_with_no_badge()
{
+        let content = concat!(
+            "```diff\n",
+            "--- a-tag\n",
+            "+++ b-tag\n",
+            "@@ -1,2 +1,1 @@\n",
+            "-// CALLOUT: gone-marker Removed in this slice.\n",
+            " fn unchanged() {}\n",
+            "```\n",
+        );
+        let out = splice_chapter(content,
```

```
SupportedRenderer::Html).expect("splice");
+        assert!(
+            !out.contains("// CALLOUT: gone-marker"),
+            "removed marker comment line should be dropped, not visible; got:
\n{out}",
+        );
+        assert!(
+            !out.contains("data-callout-badge=\"gone-marker\""),
+            "removed-side marker must not produce a badge; got:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_html_dedups_body_id_when_label_appears_in_two_blocks() {
+        // The button id and the body div id are dedup'd in lockstep: the
+        // first occurrence per label gets `id="callout-LABEL"` AND
+        // `id="callout-body-LABEL"`; subsequent occurrences emit neither.
+        // Otherwise the rendered HTML would have duplicate ids and the
+        // browser's strict-mode locator would refuse to resolve the body.
+        let content = concat!(
+            "```rust\n",
+            "// CALLOUT: shared-label First body.\n",
+            "fn one() {}\n",
+            "```\n\n",
+            "```rust\n",
+            "// CALLOUT: shared-label Second body.\n",
+            "fn two() {}\n",
+            "```\n",
+        );
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        let id_count = out.matches("id=\"callout-shared-label\"").count();
+        let body_id_count = out.matches("id=\"callout-body-shared-
label\"").count();
+        assert_eq!(
+            id_count, 1,
+            "expected exactly one id=\"callout-shared-label\"; got {id_count}
in:\n{out}",
+        );
+        assert_eq!(
+            body_id_count, 1,
+            "expected exactly one id=\"callout-body-shared-label\"; got
{body_id_count} in:\n{out}",
+        );
+    }
+
+    #[test]
+    fn splice_chapter_html_dedups_id_when_label_appears_in_diff_then_include()
{
+        // First non-empty fenced block to contain a label gets the
+        // `id="callout-LABEL"` anchor. Subsequent occurrences (same label
+        // in another block) emit the badge but skip the id so the HTML
+        // stays valid (no duplicate IDs).
+        let content = concat!(
+            "```diff\n",
```

```
+            "--- a-tag\n",
+            "+++ b-tag\n",
+            "@@ -1 +1,2 @@\n",
+            " fn unchanged() {}\n",
+            "+// CALLOUT: same-label First occurrence is in a diff.\n",
+            "```\n\n",
+            "```rust\n",
+            "// CALLOUT: same-label Second occurrence is in an include.\n",
+            "fn body() {}\n",
+            "```\n",
+        );
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        let id_count = out.matches("id=\"callout-same-label\"").count();
+        assert_eq!(
+            id_count, 1,
+            "expected exactly one id=\"callout-same-label\" across the chapter;
got {id_count} in:\n{out}",
+        );
+    }
+
+    #[test]
     fn splice_chapter_pdf_picks_up_callouts_from_added_and_context_diff_lines()
{
         // The PDF emitter still emits per-block callouts for diff fences as
         // a markdown blockquote (slice 6 shape). The HTML emitter (slice 7+)
```

[1] strip-diff — Diff-aware strip: drop `--`-prefixed marker lines entirely (no badge — the callout is gone in the new state); strip `+`-prefixed and ` `-prefixed marker lines and record post-strip positions so badges land on the line that previously held them in the diff's right-hand side.

[2] body-id-dedup — The button id, body div id, and the button's `aria-describedby` are all dedup'd in lockstep on the first occurrence per label. Without lockstep dedup, the same label appearing in two blocks (e.g. an include and a diff `+` line, both processed for badges in slice 8) would emit duplicate body div ids — invalid HTML, rejected by playwright-rs's strict-mode locator.

The fix is small but the lesson is bigger: the JS-blob sweeps silently ignored the duplicate-id violation because `document. getElementById` returns the first match. The locator-API migration made the bug observable.

Refactor (test infra) — shared e2e harness, tracing, and trace-on-failure

Continues the playwright-rs adoption from the previous refactor by moving the per-test browser setup into a shared `tests/common/e2e_harness.rs` and dogfooding two more upstream surfaces:

- **`tracing_subscriber` integration.** The harness initialises

`tracing_subscriber::fmt().with_test_writer()` once per test process, scoped through `EnvFilter` (defaults to `info`, `RUST_LOG` overrides). playwright-rs's `#[tracing::instrument]`

spans on every `goto`, `evaluate_value`, `screenshot`, `browser.close`, etc. now surface in test output on demand — drop in `RUST_LOG=playwright_rs=info cargo test --test e2e_callouts -- --nocapture` to watch the protocol play out.

- **`playwright-rs-trace` recording on failure.** Each test wraps

its body in `BrowserContext::tracing().start(...)` and stops with a save path only on panic. Failing tests leave `target/playwright-traces/<test>.zip` — drag into https://trace.playwright.dev or `npx playwright show-trace` for step-through inspection. The harness also runs the saved trace through `playwright_rs_trace::open()` and prints any errored-action summary inline so quick triage doesn't require leaving the terminal.

- **Per-test `BrowserContext`** for storage isolation between

tests. `file://` URLs don't really need it today but the pattern is correct. The harness wraps each test body in `std::panic::AssertUnwindSafe(...).catch_unwind()` (via the `futures` crate) so a panic in the test body still flows through trace cleanup before re-raising.

The diff against `tests/e2e_callouts.rs` (v6 → v7) shows every test body collapsing into a `with_traced_chapter("test-name", CH05, |page| async move { ... }).await` call — the per-test `Playwright::launch`, `pw.chromium().launch()`, `browser.new_page()`, and `browser.close()` move into the harness, and the test body inherits a `Page` already navigated to the chapter HTML.

```
--- e2e-callouts-v6
+++ e2e-callouts-v7
@@ -1,122 +1,112 @@
-use std::path::PathBuf;
+use playwright_rs::{expect, locator};

-use playwright_rs::{Playwright, expect, locator};
+mod common;
+// CALLOUT: harness-import Pulls in the shared per-test e2e harness (tests/
common/e2e_harness.rs) — every test in this file goes through
`with_traced_chapter`, so per-test Playwright launch + trace recording +
tracing_subscriber init all live in one place.
+use common::e2e_harness::with_traced_chapter;
+
+const CH05: &str = "ch05-render-inline-callouts";

 #[tokio::test]
 async fn label_only_callout_renders_badge_without_following_body() {
-    let chapter_html = chapter_path();
-    let url = format!("file://{}", chapter_html.display());
-
-    let pw = Playwright::launch().await.expect("launch playwright");
-    let browser = pw.chromium().launch().await.expect("launch chromium");
-    let page = browser.new_page().await.expect("new page");
-    page.goto(&url, None).await.expect("goto chapter");
-
-    let badge = page.locator(locator!("button#callout-cli-parse")).await;
-    expect(badge)
-        .to_have_count(1)
-        .await
-        .expect("label-only badge button must exist");
-    let body = page.locator(locator!("#callout-body-cli-parse")).await;
-    expect(body)
```

```
-        .to_have_count(0)
-        .await
-        .expect("label-only callout must not have a body popover");
-
-    browser.close().await.expect("close browser");
+    // CALLOUT: harness-call Canonical call shape. The harness opens a per-test
BrowserContext, navigates to the chapter HTML, starts a Playwright trace, runs
the closure body with the resulting Page, and on panic saves the trace to
target/playwright-traces/<name>.zip + prints a failed-action summary parsed via
playwright-rs-trace.
+    with_traced_chapter(
+        "label_only_callout_renders_badge_without_following_body",
+        CH05,
+        |page| async move {
+            let badge = page.locator(locator!("button#callout-cli-
parse")).await;
+            expect(badge)
+                .to_have_count(1)
+                .await
+                .expect("label-only badge button must exist");
+            let body = page.locator(locator!("#callout-body-cli-parse")).await;
+            expect(body)
+                .to_have_count(0)
+                .await
+                .expect("label-only callout must not have a body popover");
+        },
+    )
+    .await;
 }

 #[tokio::test]
 async fn callout_badge_renders_with_data_attribute_in_ch04() {
-    let chapter_html = chapter_path();
-    let url = format!("file://{}", chapter_html.display());
-
-    let pw = Playwright::launch().await.expect("launch playwright");
-    let browser = pw.chromium().launch().await.expect("launch chromium");
-    let page = browser.new_page().await.expect("new page");
-    page.goto(&url, None).await.expect("goto chapter");
-
-    let badges = page.locator(locator!("[data-callout-badge]")).await;
-    let count = badges.count().await.expect("count badges");
-    assert!(
-        count > 0,
-        "expected at least one [data-callout-badge]; got 0"
-    );
-    let text = badges.first().text_content().await.expect("badge text");
-    assert!(
-        text.as_deref().is_some_and(|s| !s.trim().is_empty()),
-        "expected first badge text to be non-empty; got {text:?}",
-    );
-
-    browser.close().await.expect("close browser");
+    with_traced_chapter(
+        "callout_badge_renders_with_data_attribute_in_ch04",
```

```
+        CH05,
+        |page| async move {
+            let badges = page.locator(locator!("[data-callout-badge]")).await;
+            let count = badges.count().await.expect("count badges");
+            assert!(count > 0, "expected at least one [data-callout-badge]; got
0");
+            let text = badges.first().text_content().await.expect("badge
text");
+            assert!(
+                text.as_deref().is_some_and(|s| !s.trim().is_empty()),
+                "expected first badge text to be non-empty; got {text:?}",
+            );
+        },
+    )
+    .await;
 }

 #[tokio::test]
 async fn callout_cross_ref_renders_as_anchor_to_listing_badge() {
-    let chapter_html = chapter_path();
-    let url = format!("file://{}", chapter_html.display());
-
-    let pw = Playwright::launch().await.expect("launch playwright");
-    let browser = pw.chromium().launch().await.expect("launch chromium");
-    let page = browser.new_page().await.expect("new page");
-    page.goto(&url, None).await.expect("goto chapter");
-
-    let cross_ref = page
-        .locator(locator!(r#"a[data-callout-ref="cross-ref-emit"]"#))
-        .await;
-    expect(cross_ref)
-        .to_have_attribute("href", "#callout-cross-ref-emit")
-        .await
-        .expect("cross-ref href must point at listing badge anchor");
-    let target = page
-        .locator(locator!("button#callout-cross-ref-emit"))
-        .await;
-    expect(target)
-        .to_have_count(1)
-        .await
-        .expect("listing-side badge button must exist as the cross-ref's
target");
-
-    browser.close().await.expect("close browser");
+    with_traced_chapter(
+        "callout_cross_ref_renders_as_anchor_to_listing_badge",
+        CH05,
+        |page| async move {
+            let cross_ref = page
+                .locator(locator!(r#"a[data-callout-ref="cross-ref-emit"]"#))
+                .await;
+            expect(cross_ref)
+                .to_have_attribute("href", "#callout-cross-ref-emit")
+                .await
+                .expect("cross-ref href must point at listing badge anchor");
```

```
+            let target = page
+                .locator(locator!("button#callout-cross-ref-emit"))
+                .await;
+            expect(target)
+                .to_have_count(1)
+                .await
+                .expect("listing-side badge button must exist as the cross-
ref's target");
+        },
+    )
+    .await;
 }

 #[tokio::test]
 async fn callout_marker_comment_is_stripped_and_body_reveals_on_hover() {
-    let chapter_html = chapter_path();
-    let url = format!("file://{}", chapter_html.display());
-
-    let pw = Playwright::launch().await.expect("launch playwright");
-    let browser = pw.chromium().launch().await.expect("launch chromium");
-    let page = browser.new_page().await.expect("new page");
-    page.goto(&url, None).await.expect("goto chapter");
-
-    // Find the <pre> whose sibling overlay carries the cross-ref-emit
-    // badge (xpath does the sibling traversal that CSS can't). The
-    // splicer should have stripped the literal marker comment from
-    // that pre's text.
-    let pre = page
-        .locator(locator!(
-            r#"xpath=//pre[following-sibling::div[1][.//button[@id="callout-
cross-ref-emit"]]]"#
-        ))
-        .await;
-    expect(pre.clone())
-        .not()
-        .to_contain_text("CALLOUT: cross-ref-emit")
-        .await
-        .expect("marker comment line must be stripped from the include's
<pre>");
-
-    // Body popover starts hidden and becomes visible after hovering its
-    // triggering badge.
-    let badge = page
-        .locator(locator!("button#callout-cross-ref-emit"))
-        .await;
-    badge.hover(None).await.expect("hover badge");
-    let body = page.locator(locator!("#callout-body-cross-ref-emit")).await;
-    expect(body)
-        .to_be_visible()
-        .await
-        .expect("body popover must become visible after hovering its badge");
+    with_traced_chapter(
+        "callout_marker_comment_is_stripped_and_body_reveals_on_hover",
+        CH05,
+        |page| async move {
```

```
+            // Find the <pre> whose sibling overlay carries the cross-ref-emit
+            // badge (xpath does the sibling traversal that CSS can't). The
+            // splicer should have stripped the literal marker comment from
+            // that pre's text.
+            let pre = page
+                .locator(locator!(
+                    r#"xpath=//pre[following-sibling::div[1][.//
button[@id="callout-cross-ref-emit"]]]"#
+                ))
+                .await;
+            expect(pre.clone())
+                .not()
+                .to_contain_text("CALLOUT: cross-ref-emit")
+                .await
+                .expect("marker comment line must be stripped from the
include's <pre>");

-    browser.close().await.expect("close browser");
+            // Body popover starts hidden and becomes visible after hovering
its
+            // triggering badge.
+            let badge = page
+                .locator(locator!("button#callout-cross-ref-emit"))
+                .await;
+            badge.hover(None).await.expect("hover badge");
+            let body = page.locator(locator!("#callout-body-cross-ref-
emit")).await;
+            expect(body)
+                .to_be_visible()
+                .await
+                .expect("body popover must become visible after hovering its
badge");
+        },
+    )
+    .await;
 }

 #[tokio::test]
@@ -130,72 +120,69 @@
     // 2. A `button[id="callout-LABEL"]` exists as the target
     // 3. The ref's `data-callout-ordinal` matches the target badge's
     // 4. The rendered text on the ref matches the target badge's text
-    let chapter_html = chapter_path();
-    let url = format!("file://{}", chapter_html.display());
+    with_traced_chapter(
+
"every_callout_cross_ref_resolves_to_a_badge_with_matching_ordinal_and_text",
+        CH05,
+        |page| async move {
+            let refs = page.locator(locator!("a[data-callout-ref]")).await;
+            let count = refs.count().await.expect("count refs");
+            assert!(
+                count > 0,
+                "expected at least one a[data-callout-ref] in chapter"
+            );
```

```
-    let pw = Playwright::launch().await.expect("launch playwright");
-    let browser = pw.chromium().launch().await.expect("launch chromium");
-    let page = browser.new_page().await.expect("new page");
-    page.goto(&url, None).await.expect("goto chapter");
+            for i in 0..count {
+                let r = refs.nth(i as i32);
+                let label = r
+                    .get_attribute("data-callout-ref")
+                    .await
+                    .expect("ref label")
+                    .unwrap_or_default();
+                assert!(!label.is_empty(), "ref #{i} has empty data-callout-
ref");

-    let refs = page.locator(locator!("a[data-callout-ref]")).await;
-    let count = refs.count().await.expect("count refs");
-    assert!(
-        count > 0,
-        "expected at least one a[data-callout-ref] in chapter"
-    );
+                let expected_href = format!("#callout-{label}");
+                expect(r.clone())
+                    .to_have_attribute("href", &expected_href)
+                    .await
+                    .unwrap_or_else(|e| panic!("ref `{label}`: href mismatch:
{e:?}"));

-    for i in 0..count {
-        let r = refs.nth(i as i32);
-        let label = r
-            .get_attribute("data-callout-ref")
-            .await
-            .expect("ref label")
-            .unwrap_or_default();
-        assert!(!label.is_empty(), "ref #{i} has empty data-callout-ref");
+                let target = page
+                    .locator(&format!(r#"button[id="callout-{label}"]"#))
+                    .await;
+                expect(target.clone())
+                    .to_have_count(1)
+                    .await
+                    .unwrap_or_else(|e| panic!("ref `{label}`: target badge
missing: {e:?}"));

-        let expected_href = format!("#callout-{label}");
-        expect(r.clone())
-            .to_have_attribute("href", &expected_href)
-            .await
-            .unwrap_or_else(|e| panic!("ref `{label}`: href mismatch: {e:?}"));
-
-        let target = page
-            .locator(&format!(r#"button[id="callout-{label}"]"#))
-            .await;
-        expect(target.clone())
```

```
-            .to_have_count(1)
-            .await
-            .unwrap_or_else(|e| panic!("ref `{label}`: target badge missing:
{e:?}"));
-
-        let ref_ordinal = r
-            .get_attribute("data-callout-ordinal")
-            .await
-            .expect("ref ordinal")
-            .unwrap_or_default();
-        expect(target.clone())
-            .to_have_attribute("data-callout-ordinal", &ref_ordinal)
-            .await
-            .unwrap_or_else(|e| {
-                panic!("ref `{label}`: ordinal mismatch (ref={ref_ordinal}):
{e:?}")
-            });
-
-        let ref_text = r
-            .text_content()
-            .await
-            .expect("ref text")
-            .unwrap_or_default()
-            .trim()
-            .to_string();
-        expect(target)
-            .to_have_text(&ref_text)
-            .await
-            .unwrap_or_else(|e| {
-                panic!("ref `{label}`: rendered text mismatch
(ref=\"{ref_text}\"): {e:?}")
-            });
-    }
+                let ref_ordinal = r
+                    .get_attribute("data-callout-ordinal")
+                    .await
+                    .expect("ref ordinal")
+                    .unwrap_or_default();
+                expect(target.clone())
+                    .to_have_attribute("data-callout-ordinal", &ref_ordinal)
+                    .await
+                    .unwrap_or_else(|e| {
+                        panic!("ref `{label}`: ordinal mismatch
(ref={ref_ordinal}): {e:?}")
+                    });

-    browser.close().await.expect("close browser");
+                let ref_text = r
+                    .text_content()
+                    .await
+                    .expect("ref text")
+                    .unwrap_or_default()
+                    .trim()
+                    .to_string();
+                expect(target)
```

```
+                    .to_have_text(&ref_text)
+                    .await
+                    .unwrap_or_else(|e| {
+                        panic!("ref `{label}`: rendered text mismatch
(ref=\"{ref_text}\"): {e:?}")
+                    });
+            }
+        },
+    )
+    .await;
 }

 #[tokio::test]
@@ -203,54 +190,43 @@
     // Regression guard, scoped to labels the author actually points at:
     // every `{{#callout LABEL}}` directive must have a corresponding
     // `button[id="callout-LABEL"]` somewhere in the rendered page.
-    // Catches the most common slice-shipping mistake — a cross-ref to
-    // a marker whose only chapter occurrence is in a `{{#diff}}` block
-    // before slice 8 wired diff blocks through the badge emitter.
-    // Test-fixture marker strings inside string literals are
-    // intentionally not flagged (the author isn't pointing at them).
-    let chapter_html = chapter_path();
-    let url = format!("file://{}", chapter_html.display());
+    with_traced_chapter(
+        "every_cross_refed_label_has_a_visible_badge_in_the_chapter",
+        CH05,
+        |page| async move {
+            let refs = page.locator(locator!("a[data-callout-ref]")).await;
+            let count = refs.count().await.expect("count refs");

-    let pw = Playwright::launch().await.expect("launch playwright");
-    let browser = pw.chromium().launch().await.expect("launch chromium");
-    let page = browser.new_page().await.expect("new page");
-    page.goto(&url, None).await.expect("goto chapter");
-
-    let refs = page.locator(locator!("a[data-callout-ref]")).await;
-    let count = refs.count().await.expect("count refs");
-
-    let mut missing: Vec<String> = Vec::new();
-    for i in 0..count {
-        let label = refs
-            .nth(i as i32)
-            .get_attribute("data-callout-ref")
-            .await
-            .expect("ref label")
-            .unwrap_or_default();
-        if label.is_empty() {
-            continue;
-        }
-        let target = page
-            .locator(&format!(r#"button[id="callout-{label}"]"#))
-            .await;
-        if target.count().await.expect("count target") == 0 {
-            missing.push(label);
+            let mut missing: Vec<String> = Vec::new();
+            for i in 0..count {
+                let label = refs
+                    .nth(i as i32)
+                    .get_attribute("data-callout-ref")
+                    .await
+                    .expect("ref label")
+                    .unwrap_or_default();
+                if label.is_empty() {
+                    continue;
+                }
+                let target = page
+                    .locator(&format!(r#"button[id="callout-{label}"]"#))
+                    .await;
+                if target.count().await.expect("count target") == 0 {
+                    missing.push(label);
```

```
-        }
-    }
-    missing.sort();
-    missing.dedup();
-
-    assert!(
-        missing.is_empty(),
-        "the following labels are cross-refed in chapter prose but have no \
-         `button[id=\"callout-LABEL\"]` target — most likely the cross-ref \
-         points at a marker whose only occurrence is in a `{{{{#diff}}}}` \
-         block. Add a non-diff `{{{{#include}}}}` of the source, or extract \
-         a snippet, so the badge anchor lands. Broken labels: {}",
-        missing.join(", "),
-    );
+            let mut missing: Vec<String> = Vec::new();
+            for i in 0..count {
+                let label = refs
+                    .nth(i as i32)
+                    .get_attribute("data-callout-ref")
+                    .await
+                    .expect("ref label")
+                    .unwrap_or_default();
+                if label.is_empty() {
+                    continue;
+                }
+                let target = page
+                    .locator(&format!(r#"button[id="callout-{label}"]"#))
+                    .await;
+                if target.count().await.expect("count target") == 0 {
+                    missing.push(label);
+                }
+            }
+            missing.sort();
+            missing.dedup();

-    browser.close().await.expect("close browser");
+            assert!(
+                missing.is_empty(),
+                "the following labels are cross-refed in chapter prose but have
no \
+                 `button[id=\"callout-LABEL\"]` target. Broken labels: {}",
+                missing.join(", "),
+            );
+        },
+    )
+    .await;
 }

 #[tokio::test]
@@ -258,99 +234,77 @@
     // End-to-end click-through guard: for every prose-side
     // `a[data-callout-ref]`, click it and assert the target badge ends
     // up visible (the natural in-page anchor-jump behaviour).
-    //
-    // Cross-chapter refs are out of scope: chapter prose only references
```

```
-    // callouts in listings rendered in the same chapter, by design.
-    let chapter_html = chapter_path();
-    let url = format!("file://{}", chapter_html.display());
-
-    let pw = Playwright::launch().await.expect("launch playwright");
-    let browser = pw.chromium().launch().await.expect("launch chromium");
-    let page = browser.new_page().await.expect("new page");
-    page.goto(&url, None).await.expect("goto chapter");
-
-    let refs = page.locator(locator!("a[data-callout-ref]")).await;
-    let count = refs.count().await.expect("count refs");
-    assert!(
-        count > 0,
-        "expected at least one cross-ref for click-through coverage"
-    );
-
-    let mut labels: Vec<String> = Vec::with_capacity(count);
-    for i in 0..count {
-        if let Some(label) = refs
-            .nth(i as i32)
-            .get_attribute("data-callout-ref")
-            .await
-            .expect("ref label")
-            && !label.is_empty()
-        {
-            labels.push(label);
-        }
-    }
-
-    let mut failures: Vec<String> = Vec::new();
-    for label in &labels {
-        // Reset the URL hash so each navigation is a fresh jump rather
-        // than a no-op when the current hash already matches. This is a
-        // history-API mutation; no playwright equivalent.
-        let _: String = page
-            .evaluate_value(
-                "(() => { history.replaceState(null, '', location.pathname);
return 'ok'; })()",
-            )
-            .await
-            .expect("reset hash");
+    with_traced_chapter(
+        "clicking_each_cross_ref_scrolls_target_badge_into_viewport",
+        CH05,
+        |page| async move {
+            let refs = page.locator(locator!("a[data-callout-ref]")).await;
+            let count = refs.count().await.expect("count refs");
+            assert!(
+                count > 0,
+                "expected at least one cross-ref for click-through coverage"
+            );

-        let r = page
-            .locator(&format!(r#"a[data-callout-ref="{label}"]"#))
-            .await
```

```
-            .first();
-        if let Err(e) = r.click(None).await {
-            failures.push(format!("label `{label}`: click failed: {e:?}"));
-            continue;
-        }
+            let mut labels: Vec<String> = Vec::with_capacity(count);
+            for i in 0..count {
+                if let Some(label) = refs
+                    .nth(i as i32)
+                    .get_attribute("data-callout-ref")
+                    .await
+                    .expect("ref label")
+                    && !label.is_empty()
+                {
+                    labels.push(label);
+                }
+            }

-        let target = page
-            .locator(&format!(r#"button[id="callout-{label}"]"#))
-            .await
-            .first();
-        if let Err(e) = target.scroll_into_view_if_needed().await {
-            failures.push(format!("label `{label}`: scroll failed: {e:?}"));
-            continue;
-        }
-        if !target.is_visible().await.unwrap_or(false) {
-            failures.push(format!("label `{label}`: target not visible after
click"));
-            continue;
-        }
+            let mut failures: Vec<String> = Vec::new();
+            for label in &labels {
+                // CALLOUT: clear-url-fragment Reset the URL hash so each click
is a fresh navigation rather than a no-op when the current hash already matches.
The typed `Page::clear_url_fragment()` shipped upstream as `padamson/playwright-
rust@401be500` in response to padamson/playwright-rust#89 — eliminates the last
JS string from the entire e2e suite.
+                page.clear_url_fragment().await.expect("reset hash");

-        let actual_hash: String = page
-            .evaluate_value("location.hash")
-            .await
-            .expect("read hash");
-        let expected_hash = format!("#callout-{label}");
-        if actual_hash != expected_hash {
-            failures.push(format!(
-                "label `{label}`: hash after click was `{actual_hash}` but
expected `{expected_hash}`"
-            ));
-        }
-    }
+                let r = page
+                    .locator(&format!(r#"a[data-callout-ref="{label}"]"#))
+                    .await
```

```
+                    .first();
+                if let Err(e) = r.click(None).await {
+                    failures.push(format!("label `{label}`: click failed:
{e:?}"));
+                    continue;
+                }

-    assert!(
-        failures.is_empty(),
-        "click-through navigation failed for {} of {} cross-ref(s):\n  - {}",
-        failures.len(),
-        labels.len(),
-        failures.join("\n  - "),
-    );
+                let target = page
+                    .locator(&format!(r#"button[id="callout-{label}"]"#))
+                    .await
+                    .first();
+                if let Err(e) = target.scroll_into_view_if_needed().await {
+                    failures.push(format!("label `{label}`: scroll failed:
{e:?}"));
+                    continue;
+                }
+                if !target.is_visible().await.unwrap_or(false) {
+                    failures.push(format!("label `{label}`: target not visible
after click"));
+                    continue;
+                }

-    browser.close().await.expect("close browser");
-}
+                let actual_hash: String = page
+                    .evaluate_value("location.hash")
+                    .await
+                    .expect("read hash");
+                let expected_hash = format!("#callout-{label}");
+                if actual_hash != expected_hash {
+                    failures.push(format!(
+                        "label `{label}`: hash after click was `{actual_hash}`
but expected `{expected_hash}`"
+                    ));
+                }
+            }

-fn chapter_path() -> PathBuf {
-    let manifest_dir = env!("CARGO_MANIFEST_DIR");
-    PathBuf::from(manifest_dir)
-        .join("book")
-        .join("build")
-        .join("html")
-        .join("ch05-render-inline-callouts.html")
+            assert!(
+                failures.is_empty(),
+                "click-through navigation failed for {} of {} cross-ref(s):\n
- {}",
```

```
+                failures.len(),
+                labels.len(),
+                failures.join("\n  - "),
+            );
+        },
+    )
+    .await;
 }
```

[1] harness-import — Pulls in the shared per-test e2e harness (tests/common/e2e_harness.rs) — every test in this file goes through `with_traced_chapter`, so per-test Playwright launch + trace recording + tracing_subscriber init all live in one place.

[2] harness-call — Canonical call shape. The harness opens a per-test BrowserContext, navigates to the chapter HTML, starts a Playwright trace, runs the closure body with the resulting Page, and on panic saves the trace to target/playwright-traces/.zip + prints a failed-action summary parsed via playwright-rs-trace.

[3] clear-url-fragment — Reset the URL hash so each click is a fresh navigation rather than a no-op when the current hash already matches. The typed `Page::clear_url_fragment()` shipped upstream as `padamson/playwright-rust@401be500` in response to padamson/playwright-rust#89 — eliminates the last JS string from the entire e2e suite.

Three strategically placed callout markers anchor the long diff above: the harness import (callout **[1]**), the canonical call shape every test now follows (callout **[2]**), and the line that drops the last JS string in the suite (callout **[3]**).

The harness landed without the originally-planned shared `Browser` optimisation. Each `#[tokio::test]` creates its own tokio runtime; a `Browser` handle's internal channels are bound to the runtime that created them. Sharing a `Browser` via `tokio::sync::OnceCell` across tests deadlocks once the first test's runtime exits — subsequent tests block forever waiting for responses on dead channels. Per-test `Playwright::launch + browser.launch` is the price of `#[tokio::test]` runtime isolation.

The dogfooding cycle filed this as playwright-rust#90 and the upstream response landed within the same pass: a debug-build assertion that captures the launching runtime's ID at `Connection` construction and panics in `Connection::send_message` when the current runtime differs. Silent deadlock is now a loud panic with a clear message. The mdbook-listings harness also gained verbose chatter at `playwright_rs=debug` from the same upstream — each protocol message dispatched dumped tens of KB per test. Filed as playwright-rust#91 upstream demoted the per-message log lines from `debug` to `trace`, making `RUST_LOG=playwright_rs=debug` usable for triage again.

This refactor also folds in the upstream resolution of playwright-rust#89. The previous refactor's locator/assertion migration left exactly one JS string in the suite — a `history.replaceState(null, '', location.pathname)` mutation in the click-through-navigation test, since playwright-rs had no typed equivalent. Filed #89 upstream; resolved within the dogfooding pass as commit `401be500` which adds `Page::clear_url_fragment()`. Bumping

the workspace `playwright-rs` git pin past that commit and replacing the JS line with `page.clear_url_fragment().await` (visible in the v6 → v7 diff above) makes the e2e suite JS-string-free.

The migration also surfaced a long-standing positioning bug. The v6 → v7 diff above is 600 lines tall — its first two callouts (`harness-import` at line 9, `harness-call` at line 35) sit near the top, the third (`clear-url-fragment` at line 529) near the bottom. In the browser, only the third one rendered where it should; the other two visually appeared inside the *previous* diff (the `callout-v5 → callout-v6` block in the previous refactor section). The cross-references still navigated to the right anchors — the IDs were correct — but the badges themselves had drifted 1800px above their intended rows.

Root cause: the overlay's CSS positioning formula assumed each listing line rendered at 1.5em (in the overlay's 0.875em font = 21px). mdbook's `<pre>` actually uses `line-height: normal`, which Chromium computes as 1.13 for monospace = 18px. A 3px per-line gap × 600 lines compounds to 1800px of cumulative drift — enough to push the line-9 and line-35 badges entirely out of the v6 → v7 diff and into the previous sibling pre.

Fix: ship a tiny per-book init script (registered via `additional-js` in `book.toml`, alongside the existing `additional-css` entry) that walks every `.callout-overlay` on page load, measures the previous `<pre>`'s actual rendered height, divides by the listing's line count, and writes the per-line pixel value as a CSS custom property `--callout-line-px` on the overlay. The CSS formula then uses `var(--callout-line-px, 1.5em)` instead of the bare `1.5em`, so the bug doesn't reappear no matter what font or theme an author drops in. A new e2e regression test — `every_badge_renders_inside_its_owning_pre` — asserts that every badge's bounding box lies within its sibling pre's, so this can't silently regress. The diff against `tests/e2e_callouts.rs` (v7 → v8) shows the new test:

```
--- e2e-callouts-v7
+++ e2e-callouts-v8
@@ -36,7 +36,10 @@
         |page| async move {
             let badges = page.locator(locator!("[data-callout-badge]")).await;
             let count = badges.count().await.expect("count badges");
-            assert!(count > 0, "expected at least one [data-callout-badge]; got
0");
+            assert!(
+                count > 0,
+                "expected at least one [data-callout-badge]; got 0"
+            );
             let text = badges.first().text_content().await.expect("badge
text");
             assert!(
                 text.as_deref().is_some_and(|s| !s.trim().is_empty()),
@@ -308,3 +311,57 @@
     )
     .await;
 }
+
+#[tokio::test]
+async fn every_badge_renders_inside_its_owning_pre() {
+    // Regression guard for the long-diff badge mispositioning bug:
+    // each callout badge must visually land within the y-range of the
```

```
+    // <pre> it belongs to (the one immediately preceding its
+    // .callout-overlay parent). Pre-fix, badges in long diffs drifted
+    // ~3px per line above their intended row because the overlay's
+    // assumed line-height (1.5em at 0.875em font = 21px) didn't match
+    // the pre's rendered line-height (`normal` ~ 18px for monospace).
+    // For a 600-line diff that compounds to ~1800px, landing badges
+    // inside the wrong sibling pre.
+    with_traced_chapter(
+        "every_badge_renders_inside_its_owning_pre",
+        CH05,
+        |page| async move {
+            // For each .callout-overlay, locate its sibling <pre> and
+            // every .callout-badge inside, and verify each badge's y
+            // sits within the pre's y-range.
+            let report: String = page
+                .evaluate_value(
+                    r#"(() => {
+                      const failures = [];
+                      const overlays = document.querySelectorAll('.callout-
overlay');
+                      overlays.forEach((o, i) => {
+                        const pre = o.previousElementSibling;
+                        if (!pre || pre.tagName !== 'PRE') return;
+                        const preBox = pre.getBoundingClientRect();
+                        const preTopAbs = preBox.top + window.scrollY;
+                        const preBotAbs = preBox.bottom + window.scrollY;
+                        o.querySelectorAll('.callout-badge').forEach(b => {
+                          const bb = b.getBoundingClientRect();
+                          const bAbs = bb.top + window.scrollY;
+                          if (bAbs < preTopAbs - 2 || bAbs > preBotAbs + 2) {
+                            failures.push(
+                              `overlay#${i} badge#${b.id ||
b.dataset.calloutBadge}: ` +
+                                `y=${bAbs.toFixed(0)}
pre=[${preTopAbs.toFixed(0)}..${preBotAbs.toFixed(0)}]`
+                            );
+                          }
+                        });
+                      });
+                      return failures.join('\n');
+                    })()"#,
+                )
+                .await
+                .expect("evaluate badges-vs-pre");
+            assert!(
+                report.is_empty(),
+                "badges rendered outside their owning <pre>:\n{report}"
+            );
+        },
+    )
+    .await;
+}
```

Slice 9 — line-range support for `{{#diff}}` and `{{#include}}`

The previous refactor's badge-positioning fix put every callout back on the line that previously held its `CALLOUT:` marker. But the visual UX problem that prompted the fix has another side: a 600-line diff is taller than the browser viewport regardless of where badges land within it. Cross-ref prose at the bottom of a long diff anchor-jumps the reader hundreds of em above; on the way back down, the prior section's overlay (with its own ordinal-1 and ordinal-2 badges) sits in the line of sight long before the intended diff scrolls back into view. The author can fight this by freezing snippet listings — extract just the first 30 lines of `tests/e2e_callouts.rs` v7 as `e2e-callouts-v7-imports.rs`, freeze it, diff *that* against the corresponding slice of v6 — but that's a lot of boilerplate per fragment, plus a maintenance tax every time the parent listing evolves.

This slice adds line-range support to the two directives that render frozen-listing bytes: `{{#diff}}` and `{{#include}}`. Both accept optional `START:END` arguments to render only the corresponding fragment of the source files. Endpoints are inclusive and 1-based; empty endpoints (`200:`, `:100`, `:`) mean "to end" / "to start" / "whole file". Out-of-range endpoints clamp silently — authors using `200:` to mean "from line 200 to whatever the end happens to be" don't have to know the file's exact length.

The `{{#diff}}` form takes two ranges (one per operand) since line numbers shift between versions:

```
{{#diff a b}}                    # whole files (today's behaviour)
{{#diff a b 1:50 1:60}}          # left lines 1-50 vs right lines 1-60
{{#diff a b 200: 220:}}          # from line N to end-of-file
{{#diff a b :100 :100}}          # from start to line 100
```

The `{{#include}}` form re-uses mdBook's existing `:start:end` suffix syntax — same shape readers already learn for the built-in include directive:

```
{{#include listings/foo.rs}}              # whole file
{{#include listings/foo.rs:1:30}}         # lines 1-30
{{#include listings/foo.rs:200:}}         # from line 200 to EOF
```

The diff splicer slices both source files to their respective ranges before running the diff algorithm; the include splicer slices the file body before inlining. CALLOUT-marker stripping and badge emission run on the post-slice content, so `--callout-line` values refer to the rendered slice (line 5 in the slice = line 5 in the rendered `<pre>`, regardless of where that line was in the original file). The locator anchors emitted for the screenshot tool gain a range data-attribute when ranges are present — `data-listing-diff-left-range="1:50"`, `data-listing-tag-range="1:30"` — keeping the (tag, range) pair unique even when the same listing is shown sliced more than once in a chapter.

The new `LineRange` struct lives in `src/diff.rs` (and is re-used by the include splicer). The diff against the slice-8 state of `src/diff.rs` (v8 → v9), sliced to the prelude where the struct and its `slice()` / `render()` helpers live, shows the new types landing as a pure addition between the existing `DiffDirective` struct and the parser:

```
--- diff-v8
+++ diff-v9
```

```
@@ -9,22 +9,105 @@

 /// `span` covers the directive in full (`{{#diff …}}` inclusive) so callers
 /// can replace the whole substring in one pass.
+///
+/// `left_range` and `right_range` are present when the directive carries
+/// optional 3rd and 4th `START:END` arguments — `{{#diff a b 1:50 1:60}}`
+/// renders only those slices of each operand. Empty endpoints mean "to
+/// start" or "to end". Two ranges (one per operand) because line numbers
+/// shift between versions.
 #[derive(Debug, Clone, PartialEq, Eq)]
 pub struct DiffDirective {
     pub left: String,
     pub right: String,
+    pub left_range: Option<LineRange>,
+    pub right_range: Option<LineRange>,
     pub span: Range<usize>,
 }

+/// 1-based inclusive line range. `None` endpoints mean "to start"
+/// (`start`) or "to end" (`end`). Out-of-range endpoints are clamped to
+/// the file's actual line count silently.
+#[derive(Debug, Clone, Copy, PartialEq, Eq)]
+pub struct LineRange {
+    pub start: Option<usize>,
+    pub end: Option<usize>,
+}
+
+impl LineRange {
+    /// Render as the `START:END` form used in directives and anchors.
+    /// Empty endpoints render as the empty string.
+    pub fn render(&self) -> String {
+        let s = self.start.map(|n| n.to_string()).unwrap_or_default();
+        let e = self.end.map(|n| n.to_string()).unwrap_or_default();
+        format!("{s}:{e}")
+    }
+
+    /// Slice `text` to the range's line span, 1-based inclusive. Returns
+    /// the substring that includes lines `[start..=end]` (clamped). The
+    /// returned substring preserves trailing newlines.
+    pub fn slice<'a>(&self, text: &'a str) -> &'a str {
+        let total = text.lines().count();
+        let start_1 = self.start.unwrap_or(1).max(1);
+        let end_1 = self.end.unwrap_or(total).min(total);
+        if start_1 > end_1 || total == 0 {
+            return "";
+        }
+        // Find byte offsets for line `start_1` and `end_1 + 1` (or EOF).
+        let mut byte_start = 0usize;
+        let mut current_line = 1usize;
+        let bytes = text.as_bytes();
+        while current_line < start_1 && byte_start < bytes.len() {
+            match text[byte_start..].find('\n') {
+                Some(off) => byte_start += off + 1,
+                None => return "",
```

```
+            }
+            current_line += 1;
+        }
+        let mut byte_end = byte_start;
+        let mut line = current_line;
+        while line ⩽ end_1 && byte_end < bytes.len() {
+            match text[byte_end..].find('\n') {
+                Some(off) ⇒ byte_end += off + 1,
+                None ⇒ {
+                    byte_end = bytes.len();
+                    break;
+                }
+            }
+            line += 1;
+        }
+        &text[byte_start..byte_end]
+    }
+}
+
+/// Parses one `START:END` token. Returns `None` when the form is malformed
+/// (the directive is then skipped just like a wrong-arity `{{#diff}}`).
+/// Empty endpoints are allowed; `:` (both empty) means whole file. Numeric
+/// endpoints must be positive (zero rejected).
+pub fn parse_line_range(tok: &str) → Option<LineRange> {
+    let (s, e) = tok.split_once(':')?;
+    let start = if s.is_empty() {
+        None
+    } else {
+        let n: usize = s.parse().ok()?;
+        if n == 0 {
+            return None;
+        }
+        Some(n)
+    };
+    let end = if e.is_empty() {
+        None
+    } else {
+        let n: usize = e.parse().ok()?;
+        if n == 0 {
+            return None;
+        }
+        Some(n)
+    };
+    Some(LineRange { start, end })
+}
+
 /// Returns every well-formed `{{#diff a b}}` directive in `content`.
 /// Backslash-escaped directives, wrong-arity matches, and any directive
 /// whose start byte falls inside a fenced code block are skipped — the
 /// fence rule lets a chapter quote literal directive examples (e.g. a
 /// frozen test fixture) without the preprocessor consuming them.
-pub fn parse_directives(content: &str) → Vec<DiffDirective> {
-    const PREFIX: &[u8] = b"{{#diff";
-    let bytes = content.as_bytes();
-    let mut out = Vec::new();
```

```
-    let mut in_fence = false;
-    let mut line_start = 0;
-    while line_start < bytes.len() {
```

The directive parser grows from a single 2-token shape to a 3-armed match that accepts `2` tokens (whole-file, today's shape) or `4` tokens (with two `START:END` ranges). Anything else — malformed ranges, wrong arity — falls through and skips the directive, same shape as today's wrong-arity handling, so authors who fat-finger a range get a literal `{{#diff …}}` in the rendered chapter rather than an opaque silent failure:

```
--- diff-v8
+++ diff-v9
@@ -58,18 +148,28 @@
                 let tokens: Vec<&str> = content[inner_start..inner_start +
end_rel]
                     .split_whitespace()
                     .collect();
-                if tokens.len() == 2 {
+                let parsed = match tokens.as_slice() {
+                    [l, r] => Some((l.to_string(), r.to_string(), None, None)),
+                    [l, r, lr, rr] => match (parse_line_range(lr),
parse_line_range(rr)) {
+                        (Some(left_range), Some(right_range)) => Some((
+                            l.to_string(),
+                            r.to_string(),
+                            Some(left_range),
+                            Some(right_range),
+                        )),
+                        _ => None,
+                    },
+                    _ => None,
+                };
+                if let Some((left, right, left_range, right_range)) = parsed {
                     out.push(DiffDirective {
-                        left: tokens[0].to_string(),
-                        right: tokens[1].to_string(),
+                        left,
+                        right,
+                        left_range,
+                        right_range,
                         span: i..directive_end,
                     });
                 }
                 i = directive_end;
             }
         }
-        line_start = line_end + 1;
-    }
-    out
-}
-
```

The splicer applies the slice between the byte-load and the diff render, branching on `Option<LineRange>` so the no-range case still loans bytes through the existing diff engine without an intermediate copy. The locator anchor's data attributes pick up the new `data-listing-diff-{left,right}-range` keys when ranges are present:

```
--- diff-v8
+++ diff-v9
@@ -263,33 +435,61 @@
+        let resolved =
             resolve(&d, manifest, book_root, chapter_dir).map_err(|source|
SpliceError {
                 chapter_path: chapter_path.map(Path::to_path_buf),
                 line: line_number(content, d.span.start),
                 source,
             })?;
-        let left = String::from_utf8_lossy(&resolved.left_bytes);
-        let right = String::from_utf8_lossy(&resolved.right_bytes);
-        let body = render(&left, &right, &resolved.left_label,
&resolved.right_label);
+        let left_full = String::from_utf8_lossy(&resolved.left_bytes);
+        let right_full = String::from_utf8_lossy(&resolved.right_bytes);
+        let left_sliced: &str = match &d.left_range {
+            Some(r) => r.slice(&left_full),
+            None => &left_full,
+        };
+        let right_sliced: &str = match &d.right_range {
+            Some(r) => r.slice(&right_full),
+            None => &right_full,
+        };
+        let body = render(
+            left_sliced,
+            right_sliced,
+            &resolved.left_label,
+            &resolved.right_label,
+        );
+        // When a range is set, similar's hunk headers are relative to the
+        // slice (line 1 of the slice = line N of the original). Shift them
+        // back to absolute line numbers so readers can map a +/- line in
+        // the rendered diff to its real position in the parent listing.
+        let left_offset = d
+            .left_range
+            .and_then(|r| r.start)
+            .map(|n| n - 1)
+            .unwrap_or(0);
+        let right_offset = d
+            .right_range
+            .and_then(|r| r.start)
+            .map(|n| n - 1)
+            .unwrap_or(0);
+        let body = shift_hunk_headers(&body, left_offset, right_offset);
         out.push_str(&content[cursor..d.span.start]);
         out.push_str("```diff\n");
         out.push_str(&body);
         out.push_str("```\n");
         // CALLOUT: diff-anchor-dual Locator anchor for the capture-screenshots
```

```
tool. Both operands are emitted as separate data-attributes so the tool can
locate a diff block by its (LEFT, RIGHT) pair — unique even when multiple diffs
share the same RIGHT tag, and unambiguous against the include splicer's `data-
listing-tag` anchors.
-        out.push_str(&format!(
-            "<div data-listing-diff-left=\"{}\" data-listing-diff-right=\"{}\"
aria-hidden=\"true\"></div>",
+        let mut anchor = format!(
+            "<div data-listing-diff-left=\"{}\" data-listing-diff-
right=\"{}\"",
             d.left, d.right,
-        ));
+        );
+        if let Some(r) = &d.left_range {
+            anchor.push_str(&format!(" data-listing-diff-left-range=\"{}\"",
r.render()));
+        }
+        if let Some(r) = &d.right_range {
+            anchor.push_str(&format!(
+                " data-listing-diff-right-range=\"{}\"",
+                r.render()
+            ));
+        }
+        anchor.push_str(" aria-hidden=\"true\"></div>");
+        out.push_str(&anchor);
         cursor = d.span.end;
     }
     out.push_str(&content[cursor..]);
     Ok(out)
-}
-
-fn line_number(content: &str, byte_offset: usize) -> usize {
-    content[..byte_offset]
-        .bytes()
-        .filter(|&b| b == b'\n')
-        .count()
-        + 1
-}
-
-#[cfg(test)]
-mod tests {
```

[1] diff-anchor-dual — Locator anchor for the capture-screenshots tool. Both operands are emitted as separate data-attributes so the tool can locate a diff block by its (LEFT, RIGHT) pair — unique even when multiple diffs share the same RIGHT tag, and unambiguous against the include splicer's `data-listing-tag` anchors.

The include splicer mirrors the diff splicer's shape. Its parser splits the directive's path on the first `:` to peel off any `:start:end` suffix — falling through to mdBook's built-in `links` preprocessor for unrecognised suffix forms (anchor names like `:setup`) so authors who already use those keep their expected behaviour:

```
--- include-v1
+++ include-v2
@@ -52,13 +57,29 @@
                 break;
             };
             let directive_end = inner_start + end_rel + 2;
-            let path = content[inner_start..inner_start + end_rel].trim();
+            let raw = content[inner_start..inner_start + end_rel].trim();
             // CALLOUT: snippets-intercept Two prefixes are intercepted:
`listings/` (frozen tags — emit anchor) and `snippets/` (no anchor; we expand to
give the callout splicer a shot at any CALLOUT markers in the snippet source).
Other forms fall through to mdbook's built-in `links` preprocessor.
-            let intercepted = path.starts_with("listings/") ||
path.starts_with("snippets/");
-            if !intercepted || path.contains(':') {
+            let intercepted = raw.starts_with("listings/") ||
raw.starts_with("snippets/");
+            if !intercepted {
                 i = directive_end;
                 continue;
             }
+            // Split on the first `:` to separate the path from an optional
+            // `:start:end` suffix (mdBook's built-in include slicing form).
+            // We accept the suffix here so listings/snippets includes can
+            // address a fragment of the file the same way mdBook's `links`
+            // preprocessor would for any other path. Other forms (anchor
+            // names, `=anchor`) fall through to `links`.
+            let (path, range) = match raw.split_once(':') {
+                Some((p, suffix)) => match parse_line_range(suffix) {
+                    Some(r) => (p, Some(r)),
+                    None => {
+                        i = directive_end;
+                        continue;
+                    }
+                },
+                None => (raw, None),
+            };
             // CALLOUT: tag-from-stem Tag is the file stem of `listings/...`
paths so `listings/sub/foo.rs` and `listings/foo.rs` produce the same anchor;
subdirectory stem collisions would clash on the anchor, but the book has none
today.
             let tag = if path.starts_with("listings/") {
                 Some(
@@ -78,13 +99,7 @@
             out.push(IncludeDirective {
                 tag,
                 rel_path: path.to_string(),
+                range,
                 span: i..directive_end,
                 fence_close_end,
             });
-            i = directive_end;
-        }
-        if line_end == bytes.len() {
-            break;
```

```
-        }
-        line_start = line_end + 1;
-    }
```

[1] snippets-intercept — Two prefixes are intercepted: `listings/` (frozen tags — emit anchor) and `snippets/` (no anchor; we expand to give the callout splicer a shot at any CALLOUT markers in the snippet source). Other forms fall through to mdbook's built-in `links` preprocessor.

[2] tag-from-stem — Tag is the file stem of `listings/...` paths so `listings/sub/foo.rs` and `listings/foo.rs` produce the same anchor; subdirectory stem collisions would clash on the anchor, but the book has none today.

The splicer slices the file body before the inline expansion and emits a `data-listing-tag-range` attribute on the locator anchor when a range is set:

```
--- include-v1
+++ include-v2
@@ -186,30 +208,33 @@
                 chapter_path: chapter_path.map(Path::to_path_buf),
             }
         })?;
-        // Why: the chapter's newline-after-directive (preserved via
-        // `content[d.span.end..]`) terminates the last content line; keeping
-        // the file's own trailing newline produces a blank line before the
-        // closing fence.
-        while body.ends_with('\n') {
-            body.pop();
-        }
-        out.push_str(&content[cursor..d.span.start]);
-        out.push_str(&body);
-        out.push_str(&content[d.span.end..close_end]);
-        if let Some(tag) = &d.tag {
-            // CALLOUT: include-anchor-emit One `<div data-listing-tag="...">`
per `listings/` include, dropped just past the closing fence so the screenshot
tool can find the rendered `<pre>` via `previousElementSibling`.
-            out.push_str(&format!(
-                "<div data-listing-tag=\"{tag}\" aria-hidden=\"true\"></
div>\n",
-            ));
+        if let Some(range) = &d.range {
+            // Prepend a two-line header that mirrors a unified-diff's
+            // `--- left-tag\n@@ -A,B +C,D @@` shape: filename basename on
+            // line 1 (analogous to `--- TAG`), `@@ start,end @@` on
+            // line 2 (analogous to the hunk header). Both lines are
+            // comment-prefixed when the file extension maps to a known
+            // single-line syntax, so syntax highlighters render them as
+            // metadata rather than invalid code.
+            let basename = std::path::Path::new(&d.rel_path)
+                .file_name()
```

```
+                .and_then(|s| s.to_str())
+                .unwrap_or(d.rel_path.as_str());
+            let prefix = std::path::Path::new(&d.rel_path)
+                .extension()
+                .and_then(|e| e.to_str())
+                .and_then(comment_prefix_for_extension)
+                .map(|p| format!("{p} "))
+                .unwrap_or_default();
+            let header = format!(
+                "{prefix}{basename}\n{prefix}@@ {},{} @@",
+                range.start.unwrap_or(1),
+                range
+                    .end
+                    .map(|n| n.to_string())
+                    .unwrap_or_else(|| "EOF".to_string()),
+            );
+            let sliced = range.slice(&body);
+            body = format!("{header}\n{sliced}");
         }
-        cursor = close_end;
-    }
-    out.push_str(&content[cursor..]);
-    Ok(out)
-}
-
-fn line_number(content: &str, byte_offset: usize) -> usize {
-    content[..byte_offset]
-        .bytes()
-        .filter(|&b| b == b'\n')
-        .count()
+        // Why: the chapter's newline-after-directive (preserved via
```

Both call sites use the same `slice()` method and `render()` formatter from `src/diff.rs`, so range semantics stay consistent across the two directives.

A subtle correctness consideration: when the diff splicer hands a slice of each operand to `similar`, the resulting unified-diff hunk headers (`@@ -A,B +C,D @@`) are keyed to slice-relative line numbers. A reader looking at a `+` line in the rendered diff would have no way to map it back to its position in the parent listing. The splicer post-processes `similar`'s output to shift every hunk header's start lines by `(range.start - 1)` per side, so `{{#diff a b 56:75 146:175}}` renders `@@ -58,18 +148,28 @@` rather than the raw `@@ -3,18 +3,28 @@` `similar` emits. The include splicer mirrors the diff splicer's two-line header shape: where a unified diff opens with `--- left-tag\n+++ right-tag\n@@ -A,B +C,D @@`, a sliced include opens with `// basename\n// @@ start,end @@` (language-aware comment prefix when the file extension has a known one). Line 1 plays the role of the diff's `--- TAG`; line 2 plays the role of the diff's `@@ -A,B +C,D @@`. Readers can tell at a glance that they're looking at a fragment, not the whole file, and they know which file the fragment came from.

Both behaviours are covered by integration tests in `tests/diff_line_ranges.rs` and `tests/include_line_ranges.rs`. The include test file is itself a useful demo: it's about 165 lines of Rust split across five tests + a small harness, and slice 9 shows the whole thing in chunks via the very `{{#include listings/foo.rs:start:end}}` syntax it tests. Each chunk gets a brief

intro paragraph; the `// basename\n// @@ start,end @@` header prepended by the splicer makes each fragment self-locating against the unsliced file.

Doc comment + imports + harness use:

```
// include-line-ranges-v1.rs
// @@ 1,17 @@
//! Integration tests for slice 9: the `{{#include path:START:END}}` line-
//! range form. Each test exercises one facet of the feature; the file is
//! frozen as `include-line-ranges-v1.rs` and shown in ch.5 slice 9 via
//! `` — the slice
//! demonstrates the include-line-range syntax by using it on this very
//! file.

use std::fs;
use std::path::PathBuf;

use mdbook_preprocessor::PreprocessorContext;
use mdbook_preprocessor::book::{Book, BookItem, Chapter};
use mdbook_preprocessor::config::Config;
use tempfile::TempDir;

mod common;
use common::mdbook_listings;
```

The first test asserts the basic slicing contract — lines outside the requested range never appear in the rendered chapter (callout **[1]**):

```
// include-line-ranges-v1.rs
// @@ 19,32 @@
// CALLOUT: include-range-slices The line-range form `path:start:end` slices the
file body before inlining; lines outside the range never appear in the rendered
chapter.
#[test]
fn include_with_line_range_inlines_only_the_sliced_lines() {
    let book = MinimalIncludeLineRangeBook::new();
    book.write_listing(
        "ranged.rs",
        b"line1\nline2\nline3\nline4\nline5\nline6\nline7\n",
    );
    let envelope =
        book.envelope_with_chapter("```rust\n{{#include listings/
ranged.rs:3:5}}\n```\n");
    let content = chapter_content(&run_preprocessor(envelope));
    assert!(content.contains("line3\nline4\nline5"));
    assert!(!content.contains("line1") && !content.contains("line7"));
}
```

[1] include-range-slices — The line-range form `path:start:end` slices the file body before inlining; lines outside the range never appear in the rendered chapter.

The header-line test pins the contract that the rendered slice is prefixed with a two-line `// basename\n// @@ start,end @@` banner (callout **[1]**):

```
// include-line-ranges-v1.rs
// @@ 34,49 @@
// CALLOUT: include-range-header The rendered listing prepends a two-line header
mirroring unified-diff's `--- TAG\n@@ -A,B +C,D @@` shape: file basename on line
1, `@@ start,end @@` on line 2. Both are comment-prefixed when the file
extension has a known single-line syntax, so highlighters render them as
metadata rather than invalid code.
#[test]
fn include_with_line_range_prepends_a_two_line_diff_style_header() {
    let book = MinimalIncludeLineRangeBook::new();
    book.write_listing(
        "ranged.rs",
        b"line1\nline2\nline3\nline4\nline5\nline6\nline7\n",
    );
    let envelope =
        book.envelope_with_chapter("```rust\n{{#include listings/
ranged.rs:3:5}}\n```\n");
    let content = chapter_content(&run_preprocessor(envelope));
    assert!(
        content.contains("// ranged.rs\n// @@ 3,5 @@"),
        "expected two-line `// basename\\n// @@ start,end @@` header; got:
\n{content}",
    );
}
```

[1] include-range-header — The rendered listing prepends a two-line header mirroring unified-diff's `--- TAG\n@@ -A,B +C,D @@` shape: file basename on line 1, `@@ start,end @@` on line 2. Both are comment-prefixed when the file extension has a known single-line syntax, so highlighters render them as metadata rather than invalid code.

The header's comment prefix is language-aware — Rust gets `//`, Python/YAML/TOML/shell get `#`, SQL gets `--`, anything else gets a raw header with no prefix at all. Three more tests pin each case, so the contract is explicit (callout **[1]**):

```
// include-line-ranges-v1.rs
// @@ 51,93 @@
// CALLOUT: include-range-header-language-aware The header's comment prefix
tracks the file extension via the same `comment_prefix_for_extension` table the
callout parser uses. Python/YAML/TOML/shell get `#`; SQL gets `--`; files with
no recognised extension get a raw header with no prefix at all (which the
highlighter may render as plain text).
#[test]
fn include_range_header_uses_hash_comment_prefix_for_python_extension() {
    let book = MinimalIncludeLineRangeBook::new();
    book.write_listing("script.py", b"a\nb\nc\nd\n");
    let envelope =
```

```
        book.envelope_with_chapter("```python\n{{#include listings/
script.py:1:2}}\n```\n");
    let content = chapter_content(&run_preprocessor(envelope));
    assert!(
        content.contains("# script.py\n# @@ 1,2 @@"),
        "expected `#`-prefixed header for `.py` extension; got:\n{content}",
    );
}

#[test]
fn include_range_header_uses_double_dash_comment_prefix_for_sql_extension() {
    let book = MinimalIncludeLineRangeBook::new();
    book.write_listing("schema.sql", b"a\nb\nc\nd\n");
    let envelope =
        book.envelope_with_chapter("```sql\n{{#include listings/
schema.sql:1:2}}\n```\n");
    let content = chapter_content(&run_preprocessor(envelope));
    assert!(
        content.contains("-- schema.sql\n-- @@ 1,2 @@"),
        "expected `--`-prefixed header for `.sql` extension; got:\n{content}",
    );
}

#[test]
fn include_range_header_omits_comment_prefix_for_unknown_extension() {
    let book = MinimalIncludeLineRangeBook::new();
    book.write_listing("readme.txt", b"a\nb\nc\nd\n");
    let envelope =
        book.envelope_with_chapter("```text\n{{#include listings/
readme.txt:1:2}}\n```\n");
    let content = chapter_content(&run_preprocessor(envelope));
    assert!(
        content.contains("readme.txt\n@@ 1,2 @@"),
        "expected raw header (no prefix) for unknown `.txt` extension; got:
\n{content}",
    );
    assert!(
        !content.contains("// readme.txt") && !content.contains("# readme.txt"),
        "no comment prefix expected; got:\n{content}",
    );
}
```

[1] include-range-header-language-aware — The header's comment prefix tracks the file extension via the same `comment_prefix_for_extension` table the callout parser uses. Python/YAML/TOML/shell get `#`; SQL gets `--`; files with no recognised extension get a raw header with no prefix at all (which the highlighter may render as plain text).

The data-attribute test pins the locator-anchor contract — the screenshot tool can address the sliced include via `[data-listing-tag-range="..."]` (callout **[1]**):

```
// include-line-ranges-v1.rs
// @@ 95,105 @@
// CALLOUT: include-range-anchor The locator anchor for a sliced include carries
a `data-listing-tag-range` attribute, so the screenshot tool can address the
same listing sliced two different ways without selector collisions.
#[test]
fn include_with_line_range_emits_range_data_attribute_on_locator_anchor() {
    let book = MinimalIncludeLineRangeBook::new();
    book.write_listing("ranged.rs", b"a\nb\nc\nd\ne\n");
    let envelope =
        book.envelope_with_chapter("```rust\n{{#include listings/
ranged.rs:2:4}}\n```\n");
    let content = chapter_content(&run_preprocessor(envelope));
    assert!(content.contains(r#"data-listing-tag="ranged""#));
    assert!(content.contains(r#"data-listing-tag-range="2:4""#));
}
```

[1] include-range-anchor — The locator anchor for a sliced include carries a `data-listing-tag-range` attribute, so the screenshot tool can address the same listing sliced two different ways without selector collisions.

The callout-composition test verifies that a `// CALLOUT:` marker inside the slice window flows through the include splicer, then the callout splicer, and emerges as a `<button id="callout-LABEL">` badge — the line-range form composes with callouts the same way whole-file includes do (callout **[1]**):

```
// include-line-ranges-v1.rs
// @@ 107,131 @@
// CALLOUT: include-range-callout-composes A `// CALLOUT:` marker that lives
inside the slice window flows through the full pipeline: the include splicer
slices the file bytes, the callout splicer strips the marker line and emits a
`<button id="callout-LABEL">` badge — the line-range form composes with callouts
the same way whole-file includes do.
#[test]
fn include_with_line_range_renders_a_badge_for_a_callout_inside_the_window() {
    let book = MinimalIncludeLineRangeBook::new();
    let mut body = String::new();
    for i in 1..=20 {
        if i == 10 {
            body.push_str("// CALLOUT: in-slice-callout Demo body for sliced-
include callouts.\n");
        } else {
            body.push_str(&format!("// row {i}\n"));
        }
    }
    book.write_listing("with-callouts.rs", body.as_bytes());
    let envelope =
        book.envelope_with_chapter("```rust\n{{#include listings/with-
callouts.rs:5:15}}\n```\n");
    let content = chapter_content(&run_preprocessor(envelope));
```

```
    assert!(
        content.contains(r#"id="callout-in-slice-callout""#),
        "expected a badge with id=callout-in-slice-callout from the slice; got:
\n{content}",
    );
    assert!(
        !content.contains("CALLOUT: in-slice-callout"),
        "the marker line itself should be stripped from the rendered listing;
got:\n{content}",
    );
}
```

[1] include-range-callout-composes — A `// CALLOUT:` marker that lives inside the slice window flows through the full pipeline: the include splicer slices the file bytes, the callout splicer strips the marker line and emits a `<button id="callout-LABEL">` badge — the line-range form composes with callouts the same way whole-file includes do.

The cross-reference test pins the half of the contract you've been clicking through this slice — chapter prose can `{{#callout LABEL}}` to a marker that lives inside a sliced include, and the directive resolves to the same `id="callout-LABEL"` anchor the badge gets (callout **[1]**):

```
// include-line-ranges-v1.rs
// @@ 133,160 @@
// CALLOUT: include-range-cross-ref-resolves Chapter prose can `{{#callout
LABEL}}` a callout whose marker lives inside a sliced include — the cross-
reference resolves to the same `id="callout-LABEL"` anchor the badge gets,
regardless of which range the include used.
#[test]
fn cross_ref_to_callout_inside_sliced_include_resolves_to_badge_anchor() {
    let book = MinimalIncludeLineRangeBook::new();
    let mut body = String::new();
    for i in 1..=20 {
        if i == 10 {
            body.push_str("// CALLOUT: refed-from-prose Cross-ref target.\n");
        } else {
            body.push_str(&format!("// row {i}\n"));
        }
    }
    book.write_listing("with-callouts.rs", body.as_bytes());
    let envelope = book.envelope_with_chapter(concat!(
        "Cross-ref demo: see callout {{#callout refed-from-prose}}.\n\n",
        "```rust\n{{#include listings/with-callouts.rs:5:15}}\n```\n",
    ));
    let content = chapter_content(&run_preprocessor(envelope));
    assert!(
        content.contains(r##"href="#callout-refed-from-prose""##),
        "cross-ref must resolve to the badge anchor; got:\n{content}",
    );
    assert!(
```

```
        content.contains(r#"id="callout-refed-from-prose""#),
        "badge id must exist as the anchor target; got:\n{content}",
    );
}
```

[1] include-range-cross-ref-resolves — Chapter prose can `{{#callout LABEL}}` a callout whose marker lives inside a sliced include — the cross-reference resolves to the same `id="callout-LABEL"` anchor the badge gets, regardless of which range the include used.

And the harness, completing the file:

```
// include-line-ranges-v1.rs
// @@ 162,210 @@
/// the `[ctx, book]` envelope mdbook hands a preprocessor.
struct MinimalIncludeLineRangeBook {
    _tmp: TempDir,
    root: PathBuf,
}

impl MinimalIncludeLineRangeBook {
    fn new() -> Self {
        let tmp = TempDir::new().unwrap();
        let root = tmp.path().to_path_buf();
        fs::create_dir_all(root.join("src/listings")).unwrap();
        // Empty manifest — the include splicer doesn't consult it; only
        // the diff splicer does.
        fs::write(root.join("listings.toml"), "version = 1\n").unwrap();
        Self { _tmp: tmp, root }
    }

    fn write_listing(&self, rel: &str, bytes: &[u8]) {
        fs::write(self.root.join("src/listings").join(rel), bytes).unwrap();
    }

    fn envelope_with_chapter(&self, content: &str) -> String {
        let ctx =
            PreprocessorContext::new(self.root.clone(), Config::default(),
"html".to_string());
        let chapter = Chapter::new("Include Line Ranges", content.to_string(),
"ilr.md", vec![]);
        let book = Book::new_with_items(vec![BookItem::Chapter(chapter)]);
        serde_json::to_string(&(&ctx, &book)).expect("serialize envelope")
    }
}

fn run_preprocessor(envelope: String) -> Book {
    let out = mdbook_listings()
        .write_stdin(envelope)
        .assert()
        .success()
```

```
        .get_output()
        .stdout
        .clone();
    serde_json::from_slice(&out).expect("Book")
}

fn chapter_content(book: &Book) -> String {
    for item in &book.items {
        if let BookItem::Chapter(c) = item {
            return c.content.clone();
        }
    }
    panic!("no chapter in book");
}
```

A future refactor could extract the test-infra refactor's {{#diff e2e-callouts-v6 e2e-callouts-v7}} into three smaller ranged diffs interleaved with prose — one for the imports + first test, one for the click-through test fragment, and one tying back to the cross-refs. We're leaving the existing diff intact for now since the badge-positioning fix already addressed the correctness issue; the ergonomics improvement here is for future authoring rather than retroactive cleanup of the current materialization.

What this story does not solve

This chapter handles inline callouts (markers embedded in source code as `// CALLOUT: <label> <body>` lines) plus prose-side {{#callout LABEL}} cross-references and the line-range support that breaks long diffs and includes into reader-friendly fragments. Two related features sit outside the chapter's scope and are sketched in ch.9 (Future Work):

- **Out-of-band callouts via sidecar TOML files.** Some listings

can't carry inline markers — third-party code the author doesn't own, or block-comment-only languages like CSS where a single-line `//  CALLOUT:` form doesn't apply. A sidecar `<tag>.callouts.toml` file alongside the frozen listing would let an author attach annotations without modifying the source.

- **PDF inline-badge rendering.** In HTML, the marker comment

is stripped from the rendered listing and replaced with an inline interactive badge (slice 7). In PDF, the rendering uses a complementary shape: the marker comment stays visible, and bodies render as a styled blockquote below the listing (slice 6). A future iteration could match the HTML inline-badge form in PDF too — the design lives in ch.9. A retrospective chore — adding callouts via the sidecar form to the listings already frozen by ch.2 (Install), ch.3 (Freeze), and ch.4 (Show Diffs) — also lives in ch.9. It demonstrates how callouts replace inline-comment-style code documentation, but it needs the sidecar form available first.

Dogfooding-Driven Polish

> **Why this chapter exists**
>
> Chapters 2–5 shipped the v0.1.0 primitives (install, freeze, diff, callouts). The first real downstream project to take a dependency on those primitives — the t2t book — surfaced a handful of rendering and ergonomic gaps that the in-house book never exercised hard enough to notice. This chapter collects the resulting polish work, one slice per gap. The verify story (ch.7) is still placeholder; it'll close the v0.1.0 loop separately.
>
> If we identify dogfood, we eat it. New gaps that surface on later downstream passes get appended as new acceptance criteria and new slices — there is no "out of scope" exit door.

Story

> As a downstream book author, I want the v0.1.0 primitives to feel finished when I write real annotated prose against them — not just "the happy path runs to completion," but "the rendered output is the output I wrote, and the CLI tells me what I need to know to keep going."

Acceptance criteria

1. **Inline markdown in callout body text.** A callout body that

contains inline markdown (backticks for code spans, `*emphasis*`, `**strong**`, `[text](url)`) renders as the corresponding inline HTML in the body popover — not as literal punctuation. Block-level markdown (lists, blockquotes, headings) is out of scope: callouts are inline annotations. Raw HTML in a callout body renders as escaped text, not as pass-through HTML.

1. **Bundled assets refresh on every build, not just at install time.** Today `install` writes `mdbook-listings.css` and

`mdbook-listings.js` into the book source tree as a one-time snapshot, then the bytes drift as the binary version moves forward — `additional-css`/`additional-js` keep referencing the stale on-disk copies until the author manually re-runs `install`. The preprocessor — which already runs on every `mdbook build` — instead writes the bundled bytes into the book root, refreshing them automatically when the binary is upgraded. `install` keeps the `book.toml` registration job and adds the two asset paths to `.gitignore` so downstream books treat them as build artifacts (matches `target/`). Author override works the same way it does for any other mdbook stylesheet: drop `theirs.css` into the book directory and add `additional-css = ["./theirs.css"]` to `book.toml`. mdbook cascades the second `additional-css` entry after the first, so author rules win.

1. **Callout popover never covers the line it annotates.** The

default opens the popover to the right of the badge (the un-annotated gutter), an author override switches a specific callout to the left for narrow viewports, and a transparent `/ backdrop-filter: blur` fallback keeps the underlying code legible when overlap is unavoidable.

1. **`freeze` output closes the authoring loop.** Every successful

`freeze` prints the frozen path AND the ready-to-paste `{{#include listings/<tag>.<ext>}}` directive — the author shouldn't have to grep `listings.toml` to find the include path.

1. **A `list` (or `status`) subcommand prints `tag → frozen path → source` rows** so authors can browse the manifest as a book

accumulates listings.
1. **`install` is idempotent.** Re-running `install` on an

already-configured book is a no-op with a friendly “already installed” message; never duplicates registrations.
1. **`freeze` derives a default tag when `--tag` is omitted.** The

default `<basename>-v<next>` removes the “invent your own scheme” tax on every first-time author. Already on the v0.2.0 ROADMAP; downstream surfaced it as a real pain point, so it lives here.

The slice — outside-in narrative outline

Slice	What it adds
1	Inline markdown in callout body text (AC 1). Downstream dogfooding noticed that backticks around an identifier in a callout body rendered as literal backtick characters rather than a `<code>` span. The fix routes the body through pulldown-cmark’s inline parser before wrapping it in the `<div class="callout-body">`, strips the synthetic `<p>` wrapper, and re-applies the `{` → `&#123;` escape for cross-ref-scanner safety. Raw HTML events are remapped to text events so a body containing `<script>` still renders as `&lt;script&gt;`, not as pass-through HTML.
2	Preprocessor refreshes assets on every build (AC 2). Today `install` writes `mdbook-listings.css` and `mdbook-listings.js` into the book source tree as a one-time snapshot, then the bytes drift as the binary version moves forward — t2t Pass 3 hit this: bumping the locally-installed binary forward without re-running `install` left the rendered book mixing new HTML emission with stale CSS, producing subtle (and sometimes loud) breakage. The slice moves the asset write from `install` to the preprocessor’s run hook so the bytes refresh on every build (no-op when bytes already match). `install` keeps the `book.toml` registration job and now also adds the two asset paths to `.gitignore` so downstream books treat them as build artifacts. Migration for existing books: re-run `install`, then `git rm --cached` the two old committed copies.
3	Open the popover to the right by default (AC 3, fix 1 of 3). CSS-only positioning change on the `<div class="callout-body">` so the natural reading direction (left-to-right) drops the popover into the un-annotated gutter rather than over the line it annotates.
4	Per-callout `--align` override (AC 3, fix 2 of 3). Tiny extension to the `// CALLOUT: <label>` grammar — `// CALLOUT: <label> --align=left <body>` flips a single callout when the right-side gutter isn’t usable (sidebar, narrow viewport, badge near the page edge). The extension is shaped to scale to other per-callout options later (width, theme).

5	Transparent / `backdrop-filter: blur` fallback (AC 3, fix 3 of 3). Pure CSS. When the popover must cover the listing (narrow viewport, author override, very long body), a translucent background + backdrop blur keeps the underlying code legible behind it.
6	`freeze` output closes the loop (AC 4). Augments the `created: <tag>` line with the frozen path and the exact `{{#include listings/<tag>.<ext>}}` directive to copy-paste into the chapter.
7	`mdbook-listings list` subcommand (AC 5). Prints one row per `[[listing]]` in `listings.toml`: tag, frozen path, source path. No filtering options yet — just the basic catalogue view.
8	`install` idempotency (AC 6). After slice 2 the only things `install` writes are `book.toml` registrations and the `.gitignore` entries. The first run continues to register the preprocessor + `additional-css/additional-js` and to add the asset paths to `.gitignore`. A second run detects everything already present and prints "already installed" with no writes.
9	Default tag derivation (AC 7). When `--tag` is omitted, derive `<basename>-v<next>` by reading existing `[[listing]]` entries for the same source path and bumping the highest `vN` suffix. Surfaces a clean error if any existing tag for the same source doesn't match the `<basename>-vN` shape (the heuristic is opinionated; an author who's invented their own scheme keeps using `--tag` explicitly).

Outside-in narrative

Sections appear here as slices ship. Slices 1 and 2 have shipped; slices 3–9 are sketched in the table above.

Slice 1 — inline markdown in callout body text

The symptom: a callout body whose author reached for inline backticks — say, to call out a name like `PORT` — rendered to the popover with the literal backtick characters intact instead of a `<code>` span around the name. Annotated technical prose leans on inline-code formatting to distinguish identifiers from prose; a callout body that can't render inline code reads worse than the surrounding chapter, which defeats the whole point of attaching context to a specific line.

The diff is between the two frozen snapshots of `src/callout.rs` that bracket this slice — `callout-v6` (the last freeze, made when ch.5 wrapped) and `callout-v7` (frozen as part of this slice). It's the full file diff: there's no freeze between them. Two earlier commits modified `callout.rs` without refreezing, so their changes show up here too: the e2e-harness refactor rescoped the `splice_chapter_html_escapes_label_and_body` test assertion, and ch.5's slice 9 added the `in_inline_backticks` check near the top of `replace_callout_refs` plus the `// CALLOUT: html-escape` comment and `.replace('{', "&#123;")` line on `html_escape`. This slice's contribution is the call-site swap (line 640 of v7), the new `render_inline_markdown` function just below `html_escape`, and the unit tests at the bottom.

```
--- callout-v6
+++ callout-v7
```

```
@@ -397,11 +397,25 @@
              .any(|&(start, end)| pos >= start && pos < end)
     };

+    let bytes = content.as_bytes();
+    // Same shape as the diff/include parsers: count single backticks on
+    // the line BEFORE the directive's opening offset; an odd count means
+    // the directive sits between `…` markers (inline code span) and is a
+    // documentation example, not a real cross-ref.
+    let in_inline_backticks = |pos: usize| {
+        let line_start = content[..pos].rfind('\n').map(|i| i +
1).unwrap_or(0);
+        bytes[line_start..pos]
+            .iter()
+            .filter(|&&b| b == b'`')
+            .count()
+            % 2
+            == 1
+    };
     let mut out = String::with_capacity(content.len());
     let mut cursor = 0;
     while let Some(rel) = content[cursor..].find(CALLOUT_DIRECTIVE_OPEN) {
         let open_at = cursor + rel;
-        if in_fence(open_at) {
+        if in_fence(open_at) || in_inline_backticks(open_at) {
             // Step past the opener so we don't loop on it forever.
             out.push_str(&content[cursor..open_at +
CALLOUT_DIRECTIVE_OPEN.len()]);
             cursor = open_at + CALLOUT_DIRECTIVE_OPEN.len();
@@ -623,7 +637,7 @@
         if let Some(body) = &c.body {
             s.push_str(&format!(
                 "    <div class=\"callout-body\"{body_id_attr}
role=\"tooltip\">{}</div>\n",
-                html_escape(body),
+                render_inline_markdown(body),
             ));
         }
         s.push_str("  </div>\n");
@@ -652,11 +666,34 @@
     s
 }

+// CALLOUT: html-escape Standard HTML escapes plus `{` → `&#123;` so a callout
body that documents a `{{#callout LABEL}}` or `{{#diff a b}}` directive
(rendered into the overlay HTML, which sits OUTSIDE its fenced code block)
doesn't get its example syntax mistaken for a real directive by the cross-ref
scanner downstream. The browser still renders `&#123;&#123;` as `{{` visually.
 fn html_escape(s: &str) -> String {
     s.replace('&', "&amp;")
         .replace('<', "&lt;")
         .replace('>', "&gt;")
         .replace('"', "&quot;")
+        .replace('{', "&#123;")
+}
```

```
+
+// Render `body` as inline markdown (backticks → <code>, *em*, **strong**,
+// [text](url)) for emission into the callout overlay popover.
+fn render_inline_markdown(body: &str) -> String {
+    use pulldown_cmark::{Event, Parser, html};
+    // CALLOUT: raw-html-neutralisation Callout bodies come from code comments,
not trusted markdown — `<script>` in a YAML comment must render as
`&lt;script&gt;`, not execute. Remapping every `Event::Html`/`Event::InlineHtml`
to `Event::Text` forces raw HTML through pulldown-cmark's text-escaping path.
+    let parser = Parser::new(body).map(|event| match event {
+        Event::Html(s) | Event::InlineHtml(s) => Event::Text(s),
+        other => other,
+    });
+    let mut rendered = String::new();
+    html::push_html(&mut rendered, parser);
+    let trimmed = rendered.trim_end_matches('\n');
+    // CALLOUT: inline-only-output pulldown-cmark wraps inline content in a
single `<p>...</p>`. Callout bodies are inline annotations — the synthetic
paragraph would break popover layout — so we strip it. Block-level markdown
still parses but won't strip cleanly; that's a deliberate cue that the body
shape isn't right for the construct.
+    let stripped = trimmed
+        .strip_prefix("<p>")
+        .and_then(|s| s.strip_suffix("</p>"))
+        .unwrap_or(trimmed);
+    // CALLOUT: curly-brace-escape pulldown-cmark escapes `&`, `<`, `>`, `"`
for text but leaves `{` alone. The cross-ref scanner downstream looks for
`{{...}}`; breaking the opening `{{` is enough to neutralise it, matching the
pre-slice `html_escape` behaviour (only `{` was ever escaped — `}` always
survived).
+    stripped.replace('{', "&#123;")
 }

 #[cfg(test)]
@@ -1090,20 +1127,142 @@
     }

     #[test]
+    fn replace_callout_refs_skips_directives_inside_inline_backticks_in_prose()
{
+        // A chapter that documents the cross-ref syntax in prose like
+        // "use `{{#callout LABEL}}` to ..." must not have the example
+        // text resolve as a real cross-ref — the inline backticks mark
+        // it as a documentation example, mirroring how the diff parser
+        // skips directives between `…` on the same line.
+        let content =
+            "```rust\n// CALLOUT: greeting Hello.\n```\n\nUse `{{#callout
LABEL}}` to refer.\n";
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        assert!(
+            out.contains("`{{#callout LABEL}}`"),
+            "literal example syntax in inline backticks must survive verbatim;
got:\n{out}",
+        );
```

```
+    }
+
+    #[test]
+    fn
splice_chapter_html_escapes_curly_braces_in_body_to_protect_cross_ref_scanner()
{
+        // A callout body that documents the `{{#callout LABEL}}` syntax
+        // would, post-overlay-emit, land OUTSIDE its fenced code block
+        // — the overlay div is a sibling of the pre. Without escaping,
+        // the cross-ref scanner downstream sees the literal directive
+        // text and tries to resolve `LABEL`, failing the build.
+        let content =
+            "```rust\n// CALLOUT: lbl Authors write `{{#callout LABEL}}` to
cross-ref.\n```\n";
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        let body = out
+            .split("<div class=\"callout-body\"")
+            .nth(1)
+            .unwrap_or("")
+            .split("</div>")
+            .next()
+            .unwrap_or("");
+        assert!(
+            body.contains("&#123;&#123;#callout LABEL"),
+            "expected `{{` escaped to `&#123;` so the cross-ref scanner can't
see it; got body:\n{body}",
+        );
+        assert!(
+            !body.contains("{{#callout LABEL"),
+            "raw `{{#callout LABEL}}` must not survive into the overlay body;
got body:\n{body}",
+        );
+    }
+
+    #[test]
     fn splice_chapter_html_escapes_label_and_body() {
         let content = "```yaml\n# CALLOUT: lbl Body with <script> in it.
\n```\n";
         let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
-        let overlay = out
-            .split("<div class=\"callout-overlay\"")
+        // Scope the check to the rendered callout-body div, since the
+        // overlay is now followed by a measurement <script> emitted by
+        // the splicer itself (not user content).
+        let body = out
+            .split("<div class=\"callout-body\"")
             .nth(1)
+            .unwrap_or("")
+            .split("</div>")
+            .next()
             .unwrap_or("");
         assert!(
-            overlay.contains("&lt;script&gt;"),
```

```
-            "overlay body should escape <script>; got:\n{overlay}",
+            body.contains("&lt;script&gt;"),
+            "callout body must escape user-supplied <script>; got:\n{body}",
+        );
+        assert!(
+            !body.contains("<script>"),
+            "callout body must not contain raw <script>; got:\n{body}",
+        );
+    }
+
+    fn extract_callout_body(out: &str) -> &str {
+        out.split("<div class=\"callout-body\"")
+            .nth(1)
+            .unwrap_or("")
+            .split("</div>")
+            .next()
+            .unwrap_or("")
+    }
+
+    #[test]
+    fn callout_body_renders_inline_backticks_as_code_spans() {
+        let content =
+            "```rust\n// CALLOUT: lbl Read the `PORT` env var, fall back to
`3000`.\n```\n";
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        let body = extract_callout_body(&out);
+        assert!(
+            body.contains("<code>PORT</code>") && body.contains("<code>3000</
code>"),
+            "expected backticks rendered as <code> spans; got body:\n{body}",
         );
+    }
+
+    #[test]
+    fn callout_body_renders_strong_and_emphasis() {
+        let content = "```rust\n// CALLOUT: lbl A **bold** and *italic* note.
\n```\n";
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        let body = extract_callout_body(&out);
         assert!(
-            !overlay.contains("<script>"),
-            "overlay body must not contain raw <script>; got:\n{overlay}",
+            body.contains("<strong>bold</strong>") &&
body.contains("<em>italic</em>"),
+            "expected **/* rendered as <strong>/<em>; got body:\n{body}",
+        );
+    }
+
+    #[test]
+    fn callout_body_renders_inline_link() {
+        let content = "```rust\n// CALLOUT: lbl See [docs](https://example.
com/).\n```\n";
+        let out = splice_chapter(content,
```

```
SupportedRenderer::Html).expect("splice");
+        let body = extract_callout_body(&out);
+        assert!(
+            body.contains("<a href=\"https://example.com/\">docs</a>"),
+            "expected [text](url) rendered as anchor; got body:\n{body}",
+        );
+    }
+
+    #[test]
+    fn callout_body_curly_brace_escape_survives_inside_code_span() {
+        // Authors documenting the `{{#callout LABEL}}` directive will
+        // wrap it in backticks for clarity. The inline-markdown render
+        // must produce <code>...</code>, AND the `{` escape must still
+        // apply inside that code span so the cross-ref scanner downstream
+        // (which searches for `{{...}}`) doesn't see a real directive.
+        // Only `{` needs escaping — breaking the opening `{{` is
+        // sufficient; trailing `}}` survives, matching pre-markdown behaviour.
+        let content =
+            "```rust\n// CALLOUT: lbl Authors write `{{#callout LABEL}}` to
cross-ref.\n```\n";
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        let body = extract_callout_body(&out);
+        assert!(
+            body.contains("<code>&#123;&#123;#callout LABEL}}</code>"),
+            "expected `{{` escaped inside <code> (and `}}` left as-is, matching
old behaviour); got body:\n{body}",
+        );
+    }
+
+    #[test]
+    fn callout_body_plain_text_passes_through_unchanged() {
+        let content = "```rust\n// CALLOUT: lbl Just a plain sentence with no
markup.\n```\n";
+        let out = splice_chapter(content,
SupportedRenderer::Html).expect("splice");
+        let body = extract_callout_body(&out);
+        assert!(
+            body.contains("role=\"tooltip\">Just a plain sentence with no
markup."),
+            "plain body must follow the opening tag directly (no <p> wrapper);
got body:\n{body}",
         );
     }
```

[1] html-escape — Standard HTML escapes plus `{` → `&#123;` so a callout body that documents a `{{#callout LABEL}}` or `{{#diff a b}}` directive (rendered into the overlay HTML, which sits OUTSIDE its fenced code block) doesn't get its example syntax mistaken for a real directive by the cross-ref scanner downstream. The browser still renders `&#123;&#123;` as `{{` visually.

> **[2] raw-html-neutralisation** — Callout bodies come from code comments, not trusted markdown — `<script>` in a YAML comment must render as `&lt;script&gt;`, not execute. Remapping every `Event::Html`/`Event::InlineHtml` to `Event::Text` forces raw HTML through pulldown-cmark's text-escaping path.
>
> **[3] inline-only-output** — pulldown-cmark wraps inline content in a single `<p>…</p>`. Callout bodies are inline annotations — the synthetic paragraph would break popover layout — so we strip it. Block-level markdown still parses but won't strip cleanly; that's a deliberate cue that the body shape isn't right for the construct.
>
> **[4] curly-brace-escape** — pulldown-cmark escapes `&`, `<`, `>`, `"` for text but leaves `{` alone. The cross-ref scanner downstream looks for `{{…}}`; breaking the opening `{{` is enough to neutralise it, matching the pre-slice `html_escape` behaviour (only `{` was ever escaped — `}` always survived).

Three details inside `render_inline_markdown` earn their own callout: **[2]** guards against untrusted HTML in source comments; **[3]** explains the `<p>` strip and what happens if an author reaches for block markdown anyway; **[4]** preserves the cross-ref-scanner safety property the original `html_escape` provided.

The PDF path needs no change. `render_callout_list_pdf` interpolates the body into a markdown blockquote that typst-pdf re-parses, so markdown in the body has always rendered correctly in print — the gap was HTML-only.

Tests added in this slice:

- `callout_body_renders_inline_backticks_as_code_spans` — backticks → `<code>`.
- `callout_body_renders_strong_and_emphasis` — `**bold**` and `*italic*` → `<strong>` and `<em>`.
- `callout_body_renders_inline_link` — `[docs](https://example.com/)` → `<a href>`.
- `callout_body_curly_brace_escape_survives_inside_code_span` — the cross-ref-scanner safety property holds inside a `<code>` span.
- `callout_body_plain_text_passes_through_unchanged` — the synthetic `<p>` wrapper is stripped on plain bodies.
- The pre-existing `splice_chapter_html_escapes_label_and_body` guards the raw-HTML neutralisation (it asserts `<script>` → `&lt;script&gt;`). A new e2e assertion in `tests/e2e_callouts.rs` — `callout_body_renders_inline_backticks_as_code_spans` — closes the loop end-to-end: it hovers the `snippets-intercept` badge in the rendered ch.5 HTML and asserts that the popover contains a `<code>` element with the expected text.

The diff between `e2e-callouts-v8` (last freeze, made when ch.5 wrapped) and `e2e-callouts-v9` (frozen as part of this slice) shows the new test plus a couple of mechanical changes that came with this commit's chapter renumbering — `CH04` was renamed to `CH05` and its value bumped to `"ch05-render-inline-callouts"`. Ch.5 slice 9 also modified this file without refreezing (the `callout_inside_a_sliced_include_renders_with_resolvable_cross_ref` and

`cross_ref_badges_in_prose_render_with_full_opacity_not_subdued` tests), so those appear in the diff too.

```
--- e2e-callouts-v8
+++ e2e-callouts-v9
@@ -313,6 +313,73 @@
 }

 #[tokio::test]
+async fn cross_ref_badges_in_prose_render_with_full_opacity_not_subdued() {
+    // Regression guard: a bare-anchor listing badge (label-only marker
+    // with no body popover) is styled muted/dashed via
+    // `.callout-entry .callout-badge:only-child`. Pre-fix that rule was
+    // unscoped (`.callout-badge:only-child`) and matched every cross-ref
+    // <a> in chapter prose — they're typically the only ELEMENT child
+    // of their <p> parent (text nodes don't count for :only-child), so
+    // every inline cross-ref ended up muted/dashed. The scoping fix
+    // requires the badge to live inside a `.callout-entry` overlay
+    // before muting kicks in.
+    with_traced_chapter(
+        "cross_ref_badges_in_prose_render_with_full_opacity_not_subdued",
+        CH05,
+        |page| async move {
+            let opacity: String = page
+                .evaluate_value(
+                    r#"(() => {
+                      const a = document.querySelector('a.callout-
badge.callout-ref');
+                      if (!a) return 'no-cross-ref-found';
+                      return getComputedStyle(a).opacity;
+                    })()"#,
+                )
+                .await
+                .expect("read computed opacity");
+            assert_eq!(
+                opacity, "1",
+                "cross-ref badge in prose should have full opacity; got
`{opacity}` \
+                 (subdued styling means the .callout-entry scope on `:only-
child` regressed)",
+            );
+        },
+    )
+    .await;
+}
+
+#[tokio::test]
+async fn callout_inside_a_sliced_include_renders_with_resolvable_cross_ref() {
+    // Slice 9 demo: the chapter slices `include-line-ranges-v1.rs:73:96`
+    // and the slice carries a `// CALLOUT: include-range-cross-ref-resolves`
+    // marker. Verify the full pipeline end-to-end: the badge button has
+    // the expected id, and the prose-side `{{#callout ...}}` cross-ref
+    // resolves to that id.
+    with_traced_chapter(
+        "callout_inside_a_sliced_include_renders_with_resolvable_cross_ref",
```

```
+        CH05,
+        |page| async move {
+            let badge = page
+                .locator(locator!("button#callout-include-range-cross-ref-
resolves"))
+                .await;
+            expect(badge)
+                .to_have_count(1)
+                .await
+                .expect("badge for callout inside sliced include must exist");
+            let cross_ref = page
+                .locator(locator!(
+                    r#"a[data-callout-ref="include-range-cross-ref-resolves"]"#
+                ))
+                .await;
+            expect(cross_ref)
+                .to_have_attribute("href", "#callout-include-range-cross-ref-
resolves")
+                .await
+                .expect("cross-ref href must point at the badge anchor");
+        },
+    )
+    .await;
+}
+
+#[tokio::test]
 async fn every_badge_renders_inside_its_owning_pre() {
     // Regression guard for the long-diff badge mispositioning bug:
     // each callout badge must visually land within the y-range of the
@@ -365,3 +432,38 @@
     )
     .await;
 }
+
+#[tokio::test]
+async fn callout_body_renders_inline_backticks_as_code_spans() {
+    // ch.6 slice 1: a callout body that contains inline backticks must
+    // render the wrapped span as <code>, not as literal punctuation.
+    // The `snippets-intercept` callout in listings/include-v1.rs has
+    // four backtick spans (`listings/`, `snippets/`, `CALLOUT:`,
+    // `links`); asserting one <code> with the right text is enough to
+    // confirm the inline-markdown render path is wired up end-to-end.
+    // The body popover starts hidden — `to_have_text` uses innerText,
+    // which respects visibility, so we hover the badge first.
+    with_traced_chapter(
+        "callout_body_renders_inline_backticks_as_code_spans",
+        CH05,
+        |page| async move {
+            let badge = page
+                .locator(locator!("button#callout-snippets-intercept"))
+                .await;
+            badge.hover(None).await.expect("hover badge to reveal body");
+            let body = page
+                .locator(locator!("#callout-body-snippets-intercept"))
+                .await;
```

```
+            expect(body.clone())
+                .to_be_visible()
+                .await
+                .expect("body popover must be visible after hover");
+            let code = body.locator("code").first();
+            expect(code)
+                .to_have_text("listings/")
+                .await
+                .expect("first <code> in body must be the rendered `listings/`
backtick span");
+        },
+    )
+    .await;
+}
```

Slice 2 — preprocessor refreshes assets on every build

The symptom: a downstream book installs `mdbook-listings` once, runs `install` to drop the bundled CSS/JS into the book directory, and ships fine. Some weeks later the author bumps the binary forward via `cargo install --force` to pick up a fix. The next `mdbook build` renders the chapter against the *new* HTML emission paired with the *old* on-disk CSS/JS — silent visual breakage until the author remembers to also re-run `install`. This is exactly what t2t Pass 3 hit after we shipped slice 1's hljs-fade CSS fix.

The fix moves the asset write from "one-time at install" to "every build, idempotent." Two reusable helpers land in `src/install.rs`:

- `ensure_assets_fresh(book_root)` reads each asset path and compares

to the binary's bundled bytes; only writes when they differ. Returns `true` iff anything was written.

- `ensure_gitignore(book_root)` appends the two asset filenames to

`<book>/.gitignore` (creating the file if missing); skips entries that are already present. Returns `true` iff the file was written. `install()` is refactored to use both helpers — keeping its existing idempotency contract while now also seeding `.gitignore`. The preprocessor's `preprocess()` calls only `ensure_assets_fresh` (the gitignore is one-time setup, not per-build).

```
--- install-v8
+++ install-v9
@@ -9,14 +9,17 @@
 /// Compiled in so `cargo install mdbook-listings` produces a self-contained
 /// binary with nothing external to fetch at install time.
 pub const CSS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.css");
+pub const JS_ASSET: &[u8] = include_bytes!("../assets/mdbook-listings.js");

 /// Catches builds that stripped or replaced the asset — a missing sentinel
 /// means the bundled bytes are not the expected build-time asset.
-pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v1";
+pub const CSS_ASSET_SENTINEL: &str = "mdbook-listings-css-v3";
+pub const JS_ASSET_SENTINEL: &str = "mdbook-listings-js-v1";

-/// Shared between [`write_css_asset`] and
-/// [`BookConfig::register_listings_css`] so the two can't drift.
```

```
+/// Shared between the writer and the registrar so the two can't drift.
 pub const CSS_ASSET_FILENAME: &str = "mdbook-listings.css";
+pub const JS_ASSET_FILENAME: &str = "mdbook-listings.js";
+pub const GITIGNORE_FILENAME: &str = ".gitignore";

 /// Always overwrites — install ships the bundled bytes, not whatever a
 /// stale on-disk copy happens to contain.
@@ -25,6 +28,63 @@
     fs::write(&path, CSS_ASSET).with_context(|| format!("writing CSS asset to
{}", path.display()))
 }

+pub fn write_js_asset(book_root: &Path) -> Result<()> {
+    let path = book_root.join(JS_ASSET_FILENAME);
+    fs::write(&path, JS_ASSET).with_context(|| format!("writing JS asset to
{}", path.display()))
+}
+
+/// Idempotent: writes the bundled CSS/JS to the book root only when the
+/// on-disk bytes differ from the binary's embedded asset. Called by both
+/// `install` (one-time setup) and the preprocessor (every build), so a
+/// downstream book always renders against assets matching the binary
+/// version — no manual reinstall required after `cargo install --force`.
+/// Returns `true` iff anything was written.
+pub fn ensure_assets_fresh(book_root: &Path) -> Result<bool> {
+    let css_path = book_root.join(CSS_ASSET_FILENAME);
+    let css_already_correct = fs::read(&css_path)
+        .ok()
+        .is_some_and(|bytes| bytes.as_slice() == CSS_ASSET);
+    if !css_already_correct {
+        write_css_asset(book_root)?;
+    }
+    let js_path = book_root.join(JS_ASSET_FILENAME);
+    let js_already_correct = fs::read(&js_path)
+        .ok()
+        .is_some_and(|bytes| bytes.as_slice() == JS_ASSET);
+    if !js_already_correct {
+        write_js_asset(book_root)?;
+    }
+    Ok(!css_already_correct || !js_already_correct)
+}
+
+/// Idempotent: ensures both asset filenames are present as whole-line
+/// entries in the book's `.gitignore`. Creates the file if missing.
+/// Existing entries are left untouched; missing ones are appended.
+/// Returns `true` iff `.gitignore` was written.
+pub fn ensure_gitignore(book_root: &Path) -> Result<bool> {
+    let path = book_root.join(GITIGNORE_FILENAME);
+    let original = fs::read_to_string(&path).unwrap_or_default();
+    let needed = [CSS_ASSET_FILENAME, JS_ASSET_FILENAME];
+    let missing: Vec<&str> = needed
+        .iter()
+        .copied()
+        .filter(|entry| !original.lines().any(|l| l.trim() == *entry))
+        .collect();
```

```
+    if missing.is_empty() {
+        return Ok(false);
+    }
+    let mut new_contents = original.clone();
+    if !new_contents.is_empty() && !new_contents.ends_with('\n') {
+        new_contents.push('\n');
+    }
+    for entry in missing {
+        new_contents.push_str(entry);
+        new_contents.push('\n');
+    }
+    fs::write(&path, new_contents).with_context(|| format!("writing {}",
path.display()))?;
+    Ok(true)
+}
+
 /// Idempotent: book.toml and the CSS asset on disk are only rewritten if
 /// they differ from what install would produce.
 pub fn install(book_root: &Path) -> Result<InstallOutcome> {
@@ -45,23 +105,18 @@
     let mut config = BookConfig::parse(&original)?;
     config.register_listings_preprocessor();
     config.register_listings_css();
+    config.register_listings_js();
     let new = config.render();

-    let css_path = book_root.join(CSS_ASSET_FILENAME);
-    let css_already_correct = fs::read(&css_path)
-        .ok()
-        .is_some_and(|bytes| bytes.as_slice() == CSS_ASSET);
-
     let toml_changed = new != original;
     if toml_changed {
         fs::write(&book_toml_path, new)
             .with_context(|| format!("writing book config at {}",
book_toml_path.display()))?;
-    }
-    if !css_already_correct {
-        write_css_asset(book_root)?;
     }
+    let assets_written = ensure_assets_fresh(book_root)?;
+    let gitignore_changed = ensure_gitignore(book_root)?;

-    Ok(if toml_changed || !css_already_correct {
+    Ok(if toml_changed || assets_written || gitignore_changed {
         InstallOutcome::Installed
     } else {
         InstallOutcome::Unchanged
@@ -93,36 +148,49 @@
     }

     /// Idempotent: a second call on an already-registered config is a no-op
-    /// in the rendered output. If `[preprocessor.admonish]` is registered,
-    /// the listings entry gets `before = ["admonish"]` so the
-    /// callout → admonish-note pipeline produces correctly styled PDF
```

```
+    /// in the rendered output. The listings entry always declares
+    /// `before = ["links"]` so the include splicer sees raw
+    /// `{{#include listings/...}}` directives before mdbook's built-in
+    /// `links` preprocessor expands them. If `[preprocessor.admonish]` is
+    /// also registered, `"admonish"` is added to the same `before` list so
+    /// the callout → admonish-note pipeline produces correctly styled PDF
     /// output.
     pub fn register_listings_preprocessor(&mut self) {
         let preprocessor = subtable_mut(self.0.as_table_mut(), "preprocessor");
         let admonish_present = preprocessor.contains_key("admonish");
         let listings = subtable_mut(preprocessor, "listings");
         listings["command"] = toml_edit::value("mdbook-listings");
+        let mut before = Array::new();
         if admonish_present {
-            let mut before = Array::new();
             before.push("admonish");
-            listings["before"] = toml_edit::value(before);
         }
+        before.push("links");
+        listings["before"] = toml_edit::value(before);
     }

     /// Idempotent: duplicate entries are not appended.
     pub fn register_listings_css(&mut self) {
-        let entry = format!("./{CSS_ASSET_FILENAME}");
-        let html = subtable_mut(subtable_mut(self.0.as_table_mut(), "output"),
"html");
-        let array = html
-            .entry("additional-css")
-            .or_insert_with(|| Item::Value(Value::Array(Array::new())))
-            .as_value_mut()
-            .expect("additional-css must be a value")
-            .as_array_mut()
-            .expect("additional-css must be an array");
-        if !array.iter().any(|v| v.as_str() == Some(entry.as_str())) {
-            array.push(entry);
-        }
+        register_html_asset(self.0.as_table_mut(), "additional-css",
CSS_ASSET_FILENAME);
+    }
+
+    /// Idempotent: duplicate entries are not appended.
+    pub fn register_listings_js(&mut self) {
+        register_html_asset(self.0.as_table_mut(), "additional-js",
JS_ASSET_FILENAME);
+    }
+}
+
+fn register_html_asset(root: &mut Table, key: &'static str, filename: &str) {
+    let entry = format!("./{filename}");
+    let html = subtable_mut(subtable_mut(root, "output"), "html");
+    let array = html
+        .entry(key)
+        .or_insert_with(|| Item::Value(Value::Array(Array::new())))
+        .as_value_mut()
```

```
+        .unwrap_or_else(|| panic!("{key} must be a value"))
+        .as_array_mut()
+        .unwrap_or_else(|| panic!("{key} must be an array"));
+    if !array.iter().any(|v| v.as_str() == Some(entry.as_str())) {
+        array.push(entry);
     }
 }

@@ -156,6 +224,20 @@
     }

     #[test]
+    fn js_asset_is_non_empty() {
+        assert!(!JS_ASSET.is_empty(), "bundled JS asset must not be empty");
+    }
+
+    #[test]
+    fn js_asset_contains_sentinel() {
+        let contents = std::str::from_utf8(JS_ASSET).expect("JS asset must be
UTF-8");
+        assert!(
+            contents.contains(JS_ASSET_SENTINEL),
+            "bundled JS asset must contain sentinel `{JS_ASSET_SENTINEL}`; got:
\n{contents}",
+        );
+    }
+
+    #[test]
     fn book_config_round_trip_preserves_comments_and_ordering() {
         let input = "\
 # top comment
@@ -227,14 +309,41 @@
     }

     #[test]
-    fn
book_config_register_listings_preprocessor_orders_before_admonish_when_present()
{
+    fn book_config_register_listings_js_adds_entry() {
+        let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
+        cfg.register_listings_js();
+        let rendered = cfg.render();
+        assert!(
+            rendered.contains(r#"additional-js = ["./mdbook-listings.js"]"#),
+            "rendered config should reference the JS asset; got:\n{rendered}",
+        );
+    }
+
+    #[test]
+    fn book_config_register_listings_js_is_idempotent() {
+        let input = "[book]\ntitle = \"Test\"\n";
+        let mut cfg = BookConfig::parse(input).unwrap();
+        cfg.register_listings_js();
+        let after_first = cfg.render();
+        let mut cfg2 = BookConfig::parse(&after_first).unwrap();
```

```
+        cfg2.register_listings_js();
+        let after_second = cfg2.render();
+        assert_eq!(
+            after_first, after_second,
+            "register_listings_js must be idempotent"
+        );
+    }
+
+    #[test]
+    fn
book_config_register_listings_preprocessor_orders_before_admonish_and_links_when_admonish_prese
+     {
         let input = "[preprocessor.admonish]\ncommand = \"mdbook-admonish\"\n";
         let mut cfg = BookConfig::parse(input).unwrap();
         cfg.register_listings_preprocessor();
         let rendered = cfg.render();
         assert!(
-            rendered.contains(r#"before = ["admonish"]"#),
-            "listings should declare before = [\"admonish\"]; got:
\n{rendered}",
+            rendered.contains(r#"before = ["admonish", "links"]"#),
+            "listings should declare before = [\"admonish\", \"links\"]; got:
\n{rendered}",
         );
         assert!(
             rendered.contains("[preprocessor.admonish]"),
@@ -243,13 +352,16 @@
     }

     #[test]
-    fn
book_config_register_listings_preprocessor_skips_before_when_admonish_absent() {
+    fn
book_config_register_listings_preprocessor_orders_before_links_when_admonish_absent()
{
+        // The include splicer requires `before = ["links"]` so it sees raw
+        // `{{#include listings/...}}` before mdbook's built-in `links`
+        // expands them. Without this, the splicer silently no-ops.
         let mut cfg = BookConfig::parse("[book]\ntitle = \"Test\"\n").unwrap();
         cfg.register_listings_preprocessor();
         let rendered = cfg.render();
         assert!(
-            !rendered.contains("before"),
-            "listings should not declare a before field when admonish is
absent; got:\n{rendered}",
+            rendered.contains(r#"before = ["links"]"#),
+            "listings should declare before = [\"links\"] when admonish is
absent; got:\n{rendered}",
         );
     }
```

The new helpers carry a single `// CALLOUT:` marker each — the detail that earns the WHY comment is the {{#callout assets-on-build}} note, which lives in `main.rs` next to the preprocessor call:

```
--- main-v9
+++ main-v10
@@ -7,7 +7,7 @@
 use mdbook_listings::diff::splice_chapter as splice_diffs;
 use mdbook_listings::freeze::{FreezeOptions, FreezeOutcome, freeze};
 use mdbook_listings::include::splice_chapter as splice_includes;
-use mdbook_listings::install::{InstallOutcome, install};
+use mdbook_listings::install::{InstallOutcome, ensure_assets_fresh, install};
 use mdbook_listings::manifest::Manifest;
 use mdbook_preprocessor::book::BookItem;

@@ -127,6 +127,8 @@
 /// payload on stdout.
 fn preprocess() -> Result<()> {
     let (ctx, mut book) = mdbook_preprocessor::parse_input(std::io::stdin())?;
+    // CALLOUT: assets-on-build Refresh the bundled CSS/JS on every build so
the rendered HTML always uses assets matching the binary version. No-op when
bytes already match. Solves the asset-version-skew bug surfaced by t2t Pass 3 —
bumping the binary forward without re-running `install` no longer leaves stale
assets in place.
+    ensure_assets_fresh(&ctx.root).context("refreshing bundled CSS/JS
assets")?;
     let manifest = Manifest::load(&ctx.root)?;
     let src_dir = ctx.root.join(&ctx.config.book.src);
     let renderer = SupportedRenderer::from_renderer_name(&ctx.renderer)
```

> **[1] assets-on-build** — Refresh the bundled CSS/JS on every build so the rendered HTML always uses assets matching the binary version. No-op when bytes already match. Solves the asset-version-skew bug surfaced by t2t Pass 3 — bumping the binary forward without re-running `install` no longer leaves stale assets in place.

Tests added in this slice (all in `tests/install.rs`):

- `install_writes_gitignore_entries_for_both_assets` — end-to-end

install run produces a `.gitignore` listing both assets.

- `ensure_assets_fresh_writes_when_missing` — the bundled bytes land

on first call.

- `ensure_assets_fresh_is_noop_when_bytes_match` — preprocessor calls

this on every build; mtime churn would force unnecessary rebuilds.

- `ensure_assets_fresh_overwrites_stale_bytes` — proves the t2t

Pass 3 fix: stale on-disk copies are refreshed automatically.

- `ensure_gitignore_creates_file_when_missing` — bare-tempdir case.
- `ensure_gitignore_appends_only_missing_entries` — preserves

existing author entries; never duplicates.

- `ensure_gitignore_is_noop_when_complete` — required for AC 6

idempotency (the future slice that adds the “already installed” message depends on this).

```
--- install-tests-v4
+++ install-tests-v5
@@ -3,6 +3,10 @@
 use std::fs;
 use std::path::{Path, PathBuf};

+use mdbook_listings::install::{
+    CSS_ASSET, CSS_ASSET_FILENAME, JS_ASSET, JS_ASSET_FILENAME,
ensure_assets_fresh,
+    ensure_gitignore,
+};
 use predicates::str::contains;
 use tempfile::TempDir;

@@ -84,8 +88,8 @@

     let book_toml = fs::read_to_string(book_root.join("book.toml")).unwrap();
     assert!(
-        book_toml.contains(r#"before = ["admonish"]"#),
-        "listings should be ordered before admonish; got:\n{book_toml}",
+        book_toml.contains(r#"before = ["admonish", "links"]"#),
+        "listings should be ordered before both admonish and links; got:
\n{book_toml}",
     );
     assert!(
         book_toml.contains("[preprocessor.listings]"),
@@ -110,3 +114,148 @@
         .failure()
         .stderr(contains("book.toml not found"));
 }
+
+/// `install` writes both asset paths into `.gitignore` (creating the file
+/// if missing) so downstream books treat them as build artifacts (ch.6
+/// slice 2 / AC 2).
+#[test]
+fn install_writes_gitignore_entries_for_both_assets() {
+    let book = MinimalFixtureBook::new();
+
+    mdbook_listings()
+        .args(["install", "--book-root"])
+        .arg(book.root())
+        .assert()
+        .success();
+
+    let gitignore =
fs::read_to_string(book.root().join(".gitignore")).expect(".gitignore");
+    assert!(
+        gitignore.lines().any(|l| l.trim() == CSS_ASSET_FILENAME),
+        "`.gitignore` should list the CSS asset; got:\n{gitignore}",
+    );
+    assert!(
+        gitignore.lines().any(|l| l.trim() == JS_ASSET_FILENAME),
+        "`.gitignore` should list the JS asset; got:\n{gitignore}",
+    );
+}
```

```
+
+/// `ensure_assets_fresh` writes the bundled bytes when the on-disk copies
+/// are missing, returning `true` (something was written).
+#[test]
+fn ensure_assets_fresh_writes_when_missing() {
+    let tmp = TempDir::new().expect("tempdir");
+
+    let wrote = ensure_assets_fresh(tmp.path()).expect("ensure_assets_fresh");
+
+    assert!(wrote, "should report a write when assets were missing");
+    assert_eq!(
+        fs::read(tmp.path().join(CSS_ASSET_FILENAME)).expect("css written"),
+        CSS_ASSET,
+    );
+    assert_eq!(
+        fs::read(tmp.path().join(JS_ASSET_FILENAME)).expect("js written"),
+        JS_ASSET,
+    );
+}
+
+/// `ensure_assets_fresh` is a no-op when both files already match the
+/// bundled bytes — the preprocessor calls this on every build, so it must
+/// not churn mtimes when nothing has changed.
+#[test]
+fn ensure_assets_fresh_is_noop_when_bytes_match() {
+    let tmp = TempDir::new().expect("tempdir");
+    fs::write(tmp.path().join(CSS_ASSET_FILENAME), CSS_ASSET).unwrap();
+    fs::write(tmp.path().join(JS_ASSET_FILENAME), JS_ASSET).unwrap();
+
+    let wrote = ensure_assets_fresh(tmp.path()).expect("ensure_assets_fresh");
+
+    assert!(!wrote, "should report no-op when bytes already match");
+}
+
+/// `ensure_assets_fresh` overwrites stale on-disk bytes — this is what
+/// keeps the rendered HTML in sync with the upgraded binary even when an
+/// author skips re-running `install`.
+#[test]
+fn ensure_assets_fresh_overwrites_stale_bytes() {
+    let tmp = TempDir::new().expect("tempdir");
+    fs::write(tmp.path().join(CSS_ASSET_FILENAME), b"/* stale */").unwrap();
+    fs::write(tmp.path().join(JS_ASSET_FILENAME), b"// stale\n").unwrap();
+
+    let wrote = ensure_assets_fresh(tmp.path()).expect("ensure_assets_fresh");
+
+    assert!(wrote, "should report a write when bytes were stale");
+    assert_eq!(
+        fs::read(tmp.path().join(CSS_ASSET_FILENAME)).expect("css refreshed"),
+        CSS_ASSET,
+    );
+    assert_eq!(
+        fs::read(tmp.path().join(JS_ASSET_FILENAME)).expect("js refreshed"),
+        JS_ASSET,
+    );
+}
```

```
+
+/// `ensure_gitignore` creates `.gitignore` with both entries when no file
+/// exists.
+#[test]
+fn ensure_gitignore_creates_file_when_missing() {
+    let tmp = TempDir::new().expect("tempdir");
+
+    let wrote = ensure_gitignore(tmp.path()).expect("ensure_gitignore");
+
+    assert!(wrote, "should report a write when .gitignore was missing");
+    let gitignore =
fs::read_to_string(tmp.path().join(".gitignore")).expect(".gitignore");
+    assert!(gitignore.lines().any(|l| l.trim() == CSS_ASSET_FILENAME));
+    assert!(gitignore.lines().any(|l| l.trim() == JS_ASSET_FILENAME));
+}
+
+/// `ensure_gitignore` appends only the missing entry, leaving any existing
+/// author entries (and the entry that's already there) untouched.
+#[test]
+fn ensure_gitignore_appends_only_missing_entries() {
+    let tmp = TempDir::new().expect("tempdir");
+    let existing = "build/\nmdbook-listings.css\n";
+    fs::write(tmp.path().join(".gitignore"), existing).unwrap();
+
+    let wrote = ensure_gitignore(tmp.path()).expect("ensure_gitignore");
+
+    assert!(
+        wrote,
+        "JS entry was missing, so .gitignore should be written"
+    );
+    let gitignore =
fs::read_to_string(tmp.path().join(".gitignore")).expect(".gitignore");
+    assert!(
+        gitignore.contains("build/\n"),
+        "existing author entries must survive; got:\n{gitignore}",
+    );
+    assert_eq!(
+        gitignore
+            .lines()
+            .filter(|l| l.trim() == CSS_ASSET_FILENAME)
+            .count(),
+        1,
+        "CSS entry must not be duplicated; got:\n{gitignore}",
+    );
+    assert!(
+        gitignore.lines().any(|l| l.trim() == JS_ASSET_FILENAME),
+        "JS entry must be appended; got:\n{gitignore}",
+    );
+}
+
+/// `ensure_gitignore` is a no-op when both entries are already present —
+/// matters because re-running `install` on a configured book must not
+/// churn the file (AC 6 idempotency).
+#[test]
+fn ensure_gitignore_is_noop_when_complete() {
```

```
+    let tmp = TempDir::new().expect("tempdir");
+    let existing = format!("target/
\n{CSS_ASSET_FILENAME}\n{JS_ASSET_FILENAME}\n");
+    fs::write(tmp.path().join(".gitignore"), &existing).unwrap();
+
+    let wrote = ensure_gitignore(tmp.path()).expect("ensure_gitignore");
+
+    assert!(
+        !wrote,
+        "should report no-op when both entries already present"
+    );
+    let gitignore =
fs::read_to_string(tmp.path().join(".gitignore")).expect(".gitignore");
+    assert_eq!(gitignore, existing, ".gitignore must be byte-identical");
+}
```

Migration for an existing book (this book did exactly this in the slice-2 commit):

1. Re-run `mdbook-listings install --book-root <book>` — writes

`.gitignore` and refreshes the asset bytes.
1. `git rm --cached <book>/mdbook-listings.css <book>/mdbook-listings.js`

to untrack the old committed copies.
1. `mdbook build` regenerates the assets via the preprocessor.

After migration, `cargo install --force ... mdbook-listings` is the only step needed to upgrade — the next build picks up the new bytes automatically.

Verify Sync with Source

Placeholder — this chapter's story has not been shipped yet.

Recipes

Placeholder — recipes will crystallise as shipped stories reveal workflows worth writing down.

Future Work

The project's canonical "what's planned" reference is the top-level `ROADMAP.md` file. This chapter holds detailed design sketches for features the project plans to add — depth that doesn't fit in a roadmap's one-line bullets, written down while the design is fresh so a future implementer doesn't have to rediscover it.

When one of these features ships, its sketch leaves this chapter and reappears as a slice in its parent story chapter or as its own new chapter.

Sidecar TOML callouts

Some listings can't carry inline `// CALLOUT:` markers — code the author doesn't own (third-party crates, vendored snippets, generated code), or languages without a recognised single-line comment syntax (CSS, plain Markdown). For those cases, callouts can live in a sibling TOML file alongside the frozen listing:

```
# book/src/listings/<tag>.callouts.toml
[[callout]]
line = 47
label = "upsert-order"
body = "Preserves insertion order on replacement."

[[callout]]
line = 62
label = "empty-manifest"
# no body field → bare-anchor (label-only) form
```

The splicer loads the sidecar (when present) at the same time it parses inline markers, merges the two sets, and emits one combined overlay per fenced block. Label collisions across the two sources (the same label inline AND in the sidecar) fail the build with a diagnostic naming the duplicate label and both source locations.

This adds two acceptance criteria to ch.5's primitive: callouts can be attached without modifying the listing's bytes, and inline

- sidecar callouts compose cleanly.

PDF inline-badge rendering

HTML callouts render as interactive inline badges on the line that previously held the marker comment (ch.5 slice 7). PDF renders the same callouts in a complementary shape: marker comment visible in the listing + a styled blockquote below (ch.5 slice 6). A future iteration could match the HTML form in PDF — strip the marker comment from the PDF listing too, and render a typst inline-superscript marker on the source line instead. Bodies stay in the blockquote (no hover popover in print), each entry keyed by the same ordinal that appears on the listing-side badge.

The `pdf_callouts` integration test grows assertions for the inline marker; the existing assertions for blockquote bodies stay.

Retrospective application of callouts to earlier chapters

Once sidecar callouts are available, a chore-level pass walks back through the listings frozen by ch.2 (Install), ch.3 (Freeze), and ch.4 (Show Diffs) and adds callouts to them via the sidecar form. The point is to demonstrate, in place, how callouts replace the conventional inline-

comment style of code documentation: the prose lives in the chapter, the labels make the prose addressable from the source position, and the source stays comment-light.

This depends on the sidecar form above, since modifying the already-frozen source listings would defeat the back-catalogue concept.