Faux-Press Pipeline Walkthrough

> A fine-grained, file:line-cited tour of what actually happens when you run `faux--press render input.md -o out.pdf`. Reflects code as of commit `26652a6f` on `main` (2026-05-28).

This document is a **companion** to:

- `docs/design/identity-preserving-renderer-architecture.md` — the thesis (why)
- `docs/parity-audit/pipeline.md` — the eight-stage block diagram (what)

This file describes **how** — concrete functions, concrete IR types, concrete order. Everything cited here can be located by file:line. If a stage is described below and the cited code disagrees, **the code wins** — open a PR correcting this doc.

Table of contents

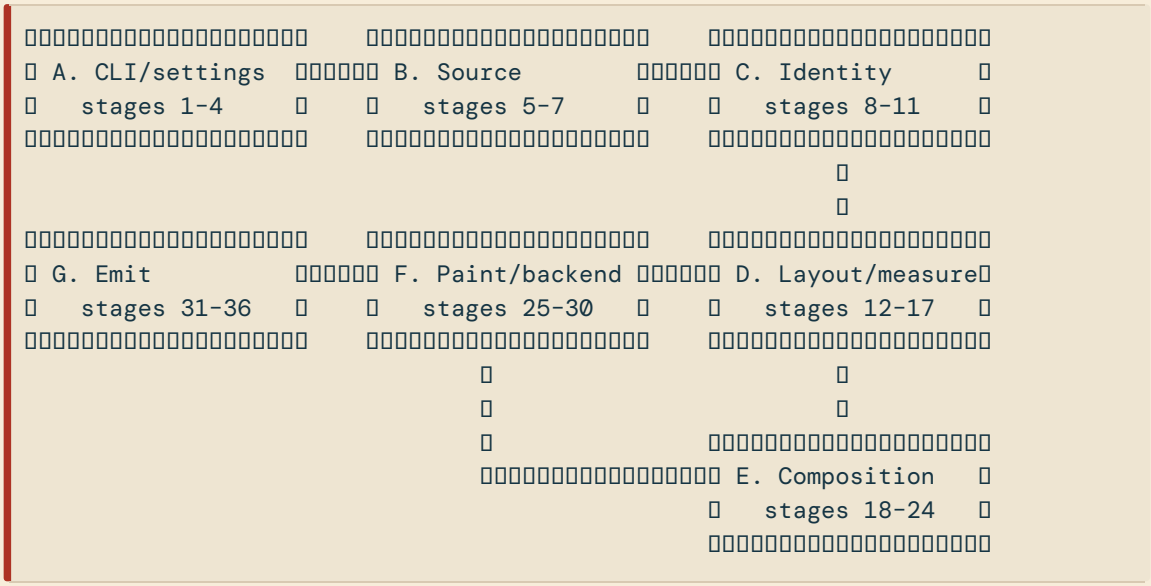
How to read this manual

Each stage follows the same template:

> **N. Stage name**
>
> - **Where:** `crate-name/src/file.rs:LINE` (jump-target)
> - **Function:** the entry point
> - **Input:** the concrete Rust type that comes in
> - **Output:** the concrete Rust type that goes out
> - **What it does:** one sentence of *what*, not *why*
> - **Non-obvious:** anything that bites — re-entrancy, mutation, parallelism, caching, fallback paths, dead code
> - **Smell?** (only when present) a flagged concern, with a number that appears again in §6

Stages are numbered globally so they can be cited as “Stage 14” in code review.

The 30-second view



Mental model:

- **A** prepares config; **B** parses bytes into the IR graph; **C** decorates it; **D** turns components into measurable layouts and shapes their text; **E** chooses page breaks and places fragments; **F** turns placements into paint ops; **G** serializes to PDF/EPUB/-canvas bytes.

- The same `RendererIrGraph` value (mutable, ~60k nodes on a book) flows from **B** through the end of **F**. It is then **dropped** before **G** to free memory before krilla serializes.
- Stages **D**, **E**, **F** can run **chapter-parallel** when `profile.enable_chapter_parallel_spine` is set.

Stage-by-stage walkthrough

A. CLI and settings (1–4)

1. CLI parse

- **Where:** `crates/faux-press/src/main.rs:431`
- **Function:** `main()` → dispatches `Cmd::Render { … }` to `handlers::render()`
- **Input:** argv (clap-parsed)
- **Output:** typed `Cmd::Render` struct
- **What it does:** Parses the subcommand and arguments.
- **Non-obvious:** `--engine` is parsed but every value resolves to identity (see comment "Phase 5.1" near `handlers.rs:13462`). Smell #1.

2. Settings cascade

- **Where:** `crates/faux-press/src/handlers.rs:13475–13582`
- **Function:** `render()` → `identity_pipeline_profile()` + `load_design_layers()` + `cli_override_settings_from_set_args()`
- **Input:** preset name, target name, `--design`, `--set k=v` overrides
- **Output:** `RendererPipelineProfile` (theme tokens, page geometry, behavior flags fully resolved)
- **What it does:** Builds the cascade `preset → project → design → CLI`.
- **Non-obvious:** the cascade lives in `render-settings/`; `handlers.rs` only wires it. The profile is the **single mutable carrier** for every behavior knob downstream — chapter-parallel toggle, fast-pdf, mode flags.

3. Read source bytes

- **Where:** `crates/faux-press/src/handlers.rs:13448`
- **Function:** `std::fs::read_to_string(input)`
- **Input:** filesystem path

- **Output:** `String` (UTF-8 markdown)
- **What it does:** I/O barrier — only place inside `render()` that touches the user's filesystem for the source. (Image/font I/O happens later, in resource resolvers.)

4. Generated-matter pre-pass

- **Where:** `crates/faux-press/src/handlers.rs:17448–17582`
- **Function:** `render_identity_bytes_with_explicit_generated_matter()`
- **Input:** raw markdown + profile + format
- **Output:** `IdentityRenderBytes` (or fall-through to stage 5)
- **What it does:** Scans for `@toc`, `@index`, `@glossary`, `@bibliography`, `@crossref` blocks. If any are present, the renderer runs **twice** — first to compute page numbers, then to inject the materialized matter and re-render.
- **Non-obvious:** this is the only stage where a render can be re-entrant. The second render path goes through stage 5 with a synthesised markdown string, *not* with the original. Smell #2.

B. Source → Identity Graph (5–7)

5. `render_identity_document()` entry

- **Where:** `crates/faux-press/src/handlers.rs:18708–18880`
- **Function:** `render_identity_document()`
- **Input:** markdown `&str`, `RendererPipelineProfile`, trace flag, fast_pdf flag
- **Output:** `IdentityRenderResult { bytes, bookmarks, object_targets, placement_audit }`
- **What it does:** Runs the rest of the pipeline. From here on, "the pipeline" is `renderer-pipeline`'s graph orchestration.

6. Asset preflight

- **Where:** `crates/faux-press/src/handlers.rs:18723–18750`
- **Function:** instantiates `GoogleFontsFetcher`, `HarfrustShaper`, `InMemoryFontInventory`, then `build_identity_media_assets()`
- **Input:** markdown + profile + input path
- **Output:** `(InMemoryMediaResourceRegistry, InMemoryImageInventory)`
- **What it does:** Pre-walks the markdown for image and Mermaid references; resolves bytes; populates the in-memory image registry.

- **Non-obvious:** image bytes are fetched **here**, before the IR graph is built. Font bytes are fetched later in stage 14.

7. Markdown → `RendererIrGraph`

- **Where:** `crates/renderer-pipeline/src/lib.rs:1168–1177`
- **Function:** `run_markdown_pipeline_with_profile_…()` → `parse_renderer_graph_with_metadata()` (lives in `crates/markdown-ingest/src/renderer_graph.rs`)
- **Input:** markdown `&str`, `RendererPipelineProfile`
- **Output:** `RendererIrGraph` — a DAG of `Source` and `Publication` nodes (~60k nodes on a book), with `IdentityRef` on every node.
- **What it does:** Parses CommonMark+GFM via `pulldown_cmark`, builds the `book_model` AST, then projects that AST onto the renderer's IR graph as Source and Publication payloads with `Contains` / `Describes` / `References` / `FootnoteRefersTo` relationship edges.
- **Non-obvious:** This is **the** authoritative producer of identity. Every downstream stage is a *transformation* of this graph; nothing else creates Source or Publication nodes. After this point the markdown string is no longer used — only the graph.

C. Identity Graph enrichment (8–11)

These four stages are pure decorations of the graph; they add nodes and edges but never remove or reorder.

8. Target capability

- **Where:** `crates/renderer-pipeline/src/lib.rs:1433–1451`
- **Input:** `RendererIrGraph`, target list (`["pdf", "canvas", "epub"]` or a subset)
- **Output:** same graph + `TargetCapabilityNodeInstance` nodes
- **What it does:** Declares each target's supported features (geometry, color profile, max page size, reflow constraints) so backend stages know what to promise/refuse.

9. Resource manifest

- **Where:** `crates/renderer-pipeline/src/lib.rs:1452–1473`
- **Output:** graph + `ResourceNodeInstance` nodes (font + image refs, not yet bound)

- **What it does:** Walks the graph and emits one resource node per font family reference and one per image. Resource nodes are placeholders — no bytes yet.

10. Media resource resolution

- **Where:** `crates/renderer-pipeline/src/lib.rs:1474–1477`
- **Function:** `resolve_media_resources()` (crate `media-resource-resolver`)
- **Input:** graph + `InMemoryMediaResourceRegistry` from stage 6
- **Output:** graph with image resources bound to digest, format, dimensions, color profile
- **Non-obvious:** does **not** touch the disk. The registry from stage 6 has already loaded bytes; this stage just attaches identity.

11. Publication → Component

- **Where:** `crates/renderer-pipeline/src/lib.rs:1478–1485`
- **Function:** `realize_components()` (crate `component-realizer`)
- **Input:** graph + `ComponentRealizationProfile` (theme tokens)
- **Output:** graph + `ComponentNodeInstance` nodes — Heading / BodyText / BlockQuote / CodeBlock / Table / Figure / List / ListItem / etc.
- **What it does:** Each Publication role gets one Component instance with variant + slots + properties. This is also where theme tokens (font, size, color, spacing) are resolved per component.
- **Non-obvious:** every component's primary slot is **seeded with the component's own publication identity** (`component-realizer/src/lib.rs:2167`). When that primary slot's name is also a structural-parent slot (`title`, `items`, `item_body`, `media`), it produces a self-loop in `MeasurementGraphIndex.structural_parent_by_occupant`. This was the root cause of a multi-minute hang fixed in `26652a6f`. Smell #3.

D. Layout and text measurement (12–17)

12. Component → Layout

- **Where:** `crates/renderer-pipeline/src/lib.rs:1486–1490`
- **Function:** `lower_components_to_layout_default()` (crate `component-layout-lowerer`)
- **Input:** graph (Components)

- **Output:** graph + `LayoutNodeInstance` (Box / Row / Column / Table / List / Flow / TextFlow / Frame / Stack / Grid)
- **What it does:** Converts each component to a measurable layout primitive with width/height/margin/padding constraints and column grids.

13. Layout → Measurement (Knuth-Plass first pass)

- **Where:** `crates/renderer-pipeline/src/lib.rs:1521–1528`
- **Function:** `lower_layout_to_measurement()` (crate `layout-measurement-lowerer`)
- **Input:** graph (Layouts) + `LayoutMeasurementProfile`
- **Output:** graph + `MeasurementNodeInstance` + Text Composition placeholders
- **What it does:** For each text composition (`Flow`, `InlineFlow`, etc.), runs a **char-advance heuristic** Knuth-Plass to predict line counts, builds intrinsic sizes, computes baselines and table column widths.
- **Non-obvious:** this stage builds `MeasurementGraphIndex` (`from_graph()` at line 2271), an O(N) index over publications, components, structural parents, table cells, and list-body occupants. **Hot path** for measurement reads — performance and correctness here matter. Smell #3 lives here.

14. Font resolution

- **Where:** `crates/renderer-pipeline/src/lib.rs:1529–1533`
- **Function:** `resolve_renderer_graph_fonts()` (crate `font-resolver`)
- **Input:** graph + `FontFetcher` (default: `GoogleFontsFetcher`) + `InMemoryFontInventory`
- **Output:** graph + `FontResource` nodes with bytes loaded; missing-glyph map populated
- **What it does:** Fetches font files for every family the graph references. Hits Google Fonts CDN with on-disk cache at `~/.cache/faux-press/fonts/`.
- **Non-obvious:** this is one of two stages that does network I/O (the other being image fetch in stage 6). A 403 here aborts the render with a typed error (observed for `Times New Roman` on the `academic-paper` preset).

15. Text shaping (Harfbuzz)

- **Where:** `crates/renderer-pipeline/src/lib.rs:1534–1544`

- **Function:** `shape_text_compositions_with_profile_and_cache()` (crate `text-shaping-lowerer`)
- **Input:** graph + font inventory + `HarfrustShaper` + optional `PersistentShapeCache`
- **Output:** graph with `ShapedRunInstance` records replacing Text Composition placeholders
- **What it does:** Runs harfbuzz per text composition; emits glyph IDs, advances, offsets, cluster info; records missing glyphs.
- **Non-obvious:** the shape cache, when wired, dedupes shaping across runs with identical (`font_digest, text, language, axes`). Per project memory this is a major performance lever (large_book-1000 text-shaping went 14.8s → 1.83s with caching + thread-local cache). Smell #4.

16. Table column refinement

- **Where:** `crates/renderer-pipeline/src/lib.rs:1551–1554`
- **Function:** `refine_table_layouts_from_shape()`
- **Input:** graph (post-shape)
- **Output:** graph with table column widths re-solved using ground-truth shaped advances
- **What it does:** Stage 13 estimated column widths from char advance; this re-solves with real glyph advances so short-label columns don't overflow.
- **Non-obvious:** shares the `TextShapingStats` enum with stage 15 — they appear as one stage in stats output but are two distinct passes. Smell #5.

17. Flow exclusion reflow (conditional)

- **Where:** `crates/renderer-pipeline/src/lib.rs:1555–1588`
- **Function:** `reflow_text_compositions_with_page_exclusions()`
- **Input:** graph (shaped) + body region rect
- **Output:** graph reflowed around sidebars / pull-quotes / shape exclusions
- **What it does:** Re-runs Knuth-Plass per composition with width constraints reduced by exclusion rects. **Only runs** when `page_master_for_pipeline()` returns a non-null body region (i.e., the layout has decorations).
- **Non-obvious:** capped at `max_retry_passes`; can fail to converge silently on pathological inputs. Smell #6.

E. Composition, fragmentation, placement (18–24)

This block is where chapter-parallel dispatch happens.

18. Spine block dispatch

- **Where:** `crates/renderer-pipeline/src/lib.rs:2078–2127`
- **Function:** `run_spine_block()`
- **Input:** graph (post-shape) + `profile.enable_chapter_parallel_spine` flag
- **Output:** graph (post-backend) + accumulated stats
- **What it does:** Branches between serial and chapter-parallel execution of stages 20–26.
- **Non-obvious:** stage 19 (composition planning) **always runs serially** even in parallel mode, because the planning cursor cannot be split across chapters. This is the architectural bottleneck the chapter-parallel work could not eliminate.

19. Composition planning (whole-graph, serial)

- **Where:** `crates/renderer-pipeline/src/lib.rs` (composition-planner stage)
- **Function:** `plan_composition()` (crate `composition-planner`)
- **Input:** graph (shaped)
- **Output:** graph + `CompositionNodeInstance` candidates per page/spread/chapter
- **What it does:** Runs `chapter-composition-solver` to generate `PageCandidate` / `SpreadCandidate` lists with quality-cost vectors; threads a `PagePlanningCursor` through document order to track current-page / current-block-fill state. Footnotes and sidebars consume cursor state.
- **Smell #7:** this stage **generates candidates only**; it does not commit. Stage 21 (composition-fragmenter) commits. The two-stage split is currently load-bearing only for non-flow compositions; for flow compositions it could be one stage.

20. Chapter partitioning (parallel mode only)

- **Where:** `crates/renderer-pipeline/src/chapter_parallel.rs:99–144`
- **Function:** `detect_chapters()`, `expand_chapter_membership()`, `subgraph_for_chapter()`
- **Input:** graph (planned) + chapter boundary policy
- **Output:** `Vec<RendererIrGraph>` — one heap-allocated subgraph per chapter

- **What it does:** Walks `Contains` relationships to identify chapter boundaries; deep-copies each chapter's subgraph into its own heap allocation so worker threads can mutate independently.

21. Composition fragmentation

- **Where:** crate `composition-fragmenter`, called per-chapter
- **Input:** chapter subgraph (post-planning)
- **Output:** subgraph + `FragmentNodeInstance` records
- **What it does:** Commits to a chosen alternative; reifies fragments (continuation phases, repeated adornments like table headers, drop-cap carry-over).

22. Fragment placement

- **Where:** crate `fragment-placer`
- **Input:** fragmented subgraph
- **Output:** subgraph + `PlacementNodeInstance` (final RectPtMilli on page)
- **What it does:** Deterministic geometry assignment — no further solver decisions, just coordinate math from the chosen plan + fragments.

23. Accessibility metadata (pre-paint, optional)

- **Where:** `crates/renderer-pipeline/src/lib.rs:2165–2180`
- **Function:** `lower_renderer_metadata(accessibility_only_metadata_profile)`
- **Input:** placed subgraph
- **Output:** subgraph + `AccessibilityNodeInstance` records (PDF/UA-1 tag tree)
- **What it does:** Emits roles, labels, reading order, alt text.
- **Non-obvious:** runs **twice** in the spine — here (pre-paint) and again at stage 25 (post-paint). Skipped entirely if `profile.skip_accessibility_records`. Smell #8.

24. (paint, see F.25)

F. Paint and backend (25–30)

25. Placement → Paint

- **Where:** crate `placement-painter`
- **Input:** placed subgraph
- **Output:** subgraph + `PaintNodeInstance` (GlyphRun / Rect / Path / Image / Group)

- **What it does:** Converts placements into backend-neutral drawing ops.

26. Accessibility metadata (post-paint, optional)

- **Where:** `crates/renderer-pipeline/src/lib.rs:2185–2196`
- **What it does:** Same call as stage 23 but after paint, refining the tag tree with paint-time identity bindings. Smell #8 (continued).

27. Backend output lowering

- **Where:** crate `backend-output-lowerer`
- **Input:** painted subgraph
- **Output:** subgraph + `BackendOutputNodeInstance` (per-target manifest)
- **What it does:** Records *what* each backend should emit — resource counts, loss records, unsupported features — without producing bytes. The actual bytes come at stage 33+.
- **Smell #9:** the manifest is rarely queried after this stage. Most of its value is captured by `EmitIR` (stage 28). Two crates duplicate the "summary of paint output" responsibility.

28. Chapter merge (parallel mode only)

- **Where:** `crates/renderer-pipeline/src/chapter_parallel.rs` (merge logic)
- **Input:** `Vec<RendererIrGraph>` from worker threads + original ordering
- **Output:** single merged `RendererIrGraph`
- **What it does:** Concatenates worker subgraphs, re-sorts nodes by original IngestId so the output graph is byte-identical to the serial run. Stats are summed.

29. Graph → EmitIR

- **Where:** `crates/renderer-pipeline/src/lib.rs:1641–1645`
- **Function:** `lower_graph_to_emit_ir()` (crate `graph-emit-ir`)
- **Input:** painted graph + `GraphEmitIrProfile`
- **Output:** `EmitIR` — immutable per-page list of `EmitOp` (text runs, rects, images, annotations, bookmarks)
- **What it does:** Bridges the mutable identity graph into a flat, serialization-friendly per-page op list.

- **Non-obvious: after this stage the `RendererIrGraph` is dropped** (`handlers.rs` around `18793`) to free 2–3 GB on book-sized inputs before krilla runs. Down-stream stages cannot inspect the graph.

30. Final renderer-metadata stage (optional, cross-cutting)

- **Where:** `crates/renderer-pipeline/src/lib.rs:1654–1677`
- **What it does:** Emits diagnostics, incremental-invalidation, document-envelope nodes.
- **Non-obvious:** the production fast path (`--fast`) skips all three of: invalidation, diagnostic, accessibility records — at which point this stage is a no-op. Smell #10.

G. Emit (31–36)

31. Graph validation

- **Where:** `crates/renderer-pipeline/src/lib.rs:1679–1686`
- **Function:** `graph.validate()`
- **Input:** graph (post-metadata)
- **Output:** `Vec<IrGraphValidationIssue>` (empty on success)
- **What it does:** Asserts edges reference valid nodes, ids unique, etc.

32. Backend selection

- **Where:** `crates/faux-press/src/handlers.rs:18844–18866`
- **What it does:** Match on format string → dispatches to `emit_pdf::emit()` / `emit_canvas::emit()` / `emit_epub::emit()`.

33. PDF metadata resolution (PDF only)

- **Where:** `handlers.rs:18835–18843`
- **Output:** `MetadataSettings { title, authors, language }`
- **What it does:** Pulls document metadata from frontmatter for the PDF / PDF/UA-1 / PDF/A-2 producer.

34. Emit PDF (krilla)

- **Where:** crate `emit-pdf`, called via `emit_pdf_checked()` at `handlers.rs:21282`
- **Input:** `EmitIR` + font inventory + image inventory + `PdfEmitConfig`
- **Output:** `(Vec<u8>, Vec<PdfEmitFailure>)`

- **What it does:** Walks `EmitIR` per-page, emits page content streams, embeds fonts and images, writes tagged PDF tree, applies DEFLATE compression (skipped if `--fast-pdf`).
- **Non-obvious:** krilla parallelizes per-page DEFLATE via its `rayon` feature flag (project memory: 1.3s saved on book-1000).

35. Emit canvas / EPUB

- **Where:** `crates/emit-canvas/`, `crates/emit-epub/`
- **What it does:** Same `EmitIR` input, different serialization. Canvas emits a JSON op stream; EPUB emits a ZIP archive of XHTML+CSS+fonts.

36. Write bytes

- **Where:** `handlers.rs:13575`
- **Function:** `write_render_bytes(output, &bytes)` or stdout
- **What it does:** I/O barrier. Final exit point.

The IR catalogue

Every IR that crosses a stage boundary, in producer order. Cross-cutting IRs listed at the bottom — they don't sit on the linear pipeline.

IR	Concrete type	Crate	File:Line	Producer	Consumer
Document Envelope	`DocumentEnvelopeInstance`	`renderer-contracts`	`ir_instance.rs:1400`	renderer-pipeline	all
Source IR	`SourceNodeInstance` + book-model AST	`renderer-contracts` + `book-model`	`ir_instance.rs:1415`	markdown-ingest	component-realizer
Publication IR	`PublicationNodeInstance`	`renderer-contracts`	`ir_instance.rs:1430`	markdown-ingest (paired with Source)	component-realizer
Inline Content	`InlineNodeInstance`	`renderer-contracts`	`ir_instance.rs:1485`	markdown-ingest	text-shaping-lowerer

IR	Concrete type	Crate	File:Line	Producer	Consumer
Style/Theme	`StyleThemeInstance`	`renderer-contracts`	`ir_instance.rs:1500`	component-realizer	layout-measurement-lowerer
Component IR	`ComponentNodeInstance`	`renderer-contracts`	`ir_instance.rs:1513`	component-realizer	component-layout-lowerer
Layout IR	`LayoutNodeInstance`	`renderer-contracts`	`ir_instance.rs:1528`	component-layout-lowerer	layout-measurement-lowerer
Measurement IR	`MeasurementNodeInstance`	`renderer-contracts`	`ir_instance.rs:1554`	layout-measurement-lowerer	composition-planner
Text Composition IR	`TextCompositionNodeInstance`	`renderer-contracts`	`ir_instance.rs:1597`	layout-measurement-lowerer (placeholders) → text-shaping-lowerer (filled)	composition-planner
Composition IR	`CompositionNodeInstance`	`renderer-contracts`	`ir_instance.rs:1640`	composition-planner	composition-fragmenter
Fragment IR	`FragmentNodeInstance`	`renderer-contracts`	`ir_instance.rs:1679`	composition-fragmenter	fragment-placer
Placement IR	`PlacementNodeInstance`	`renderer-contracts`	`ir_instance.rs:3058`	fragment-placer	placement-painter

IR	Concrete type	Crate	File:Line	Producer	Consumer
Paint IR	`PaintNodeInstance`	`renderer-contracts`	`ir_instance.rs:3110`	placement-pa inter	backend-out p ut-lowerer + graph-emit-i r
Backend Out-put IR	`BackendOutputNodeInstanc e`	`renderer-contracts`	`ir_instance.rs:3196`	backend-out p ut-lowerer	(rarely consumed)
EmitIR	`EmitIR`, `EmitOp`, `EmitGroup`	`emit-ir`	`ir.rs:74`	graph-emit-i r	emit-pdf, emit-canvas, emit-epub

Cross-cutting:

IR	Concrete type	Role
Identity / Provenance Graph	`IdentityRef`, `RendererIrGraph`	Carrier for everything above; every node has stable identity + edges
Target Capability	`TargetCapabilityNodeInstance`	What the backend can promise
Resource	`ResourceNodeInstance`	Fonts / images / SVGs / hyphenation / language
Accessibility	`AccessibilityNodeInstance`	Roles, labels, alt text, reading order
Diagnostics	`DiagnosticsNodeInstance`	Typed failure surface
Loss / Translation	`LossTranslationNodeInstance`	Per-feature losslessness
Incremental Invalidation	`IncrementalInvalidationNodeInstance`	Cache keys + dependency edges

Cross-cutting concerns

Identity graph

`RendererIrGraph` is the carrier from stage 7 to stage 29. Every node has an `IdentityRef` (kind + stable id + precomputed hash). Edges record `LoweringStage` so any stage can ask "where did this node come from?".

The `IdentityRef` precomputed-hash pattern is the universal performance lever (project memory: ~25% pipeline win). Search for `graph.node(&id)` in any new loop — that call is O(nodes); building a kind-by-id index first is required. Project memory flags this as a recurring O(N□) trap.

Font resolver

Two-layer: pure plan core (decides which family + axes to fetch) plus an imperative shell (HTTP + filesystem). Cache lives at `~/.cache/faux-press/fonts/`. Failures abort with typed errors.

Media resource registry

In-memory only — no disk cache. Image bytes are loaded eagerly in stage 6, attached lazily in stage 10.

Diagnostics

Every stage emits typed `Diagnostic` records into the graph (severity + category + identity reference). They are collected only at the end and never gate the render — the pipeline is "best effort with diagnostics," not "refuse on warnings." This is intentional but worth knowing when reviewing.

Trace mode (`--trace`)

Sidecars per-stage stats and lossless-state hashes. Used in CI and the visual-audit script (`scripts/pdf_visual_audit.py`).

Architectural concerns and cleanup opportunities

These are flagged as **smells** — places where the code asks "is this still the right shape?" If you're picking up a refactor, this is the punch list.

Smell #1 — `--engine` is dead

The `--engine` flag is parsed (`legacy` / `identity` / `auto`) but every value resolves to identity (handlers.rs near 13462, comment "Phase 5.1"). Decision: drop the flag, or make `legacy` actually error so users learn it's gone.

Smell #2 — re-entrant render for generated matter

The TOC/Index/Bibliography path in stage 4 re-renders by **mutating the markdown string and going through stage 5 again**. The second pass cannot introspect the first — it just trusts page numbers came back consistent. Cleaner: separate "draft pass" from "final pass" as named entry points so the dataflow is explicit.

Smell #3 — `MeasurementGraphIndex` invariants are implicit

`structural_parent_by_occupant` is supposed to be a forest, but the invariant was undocumented and unenforced until 2026-05-28. Two writers (publication relationships + component slot occupants) populate it without coordinating; the realizer seeds primary slots with the component's own identity, producing a self-loop when the slot name matches a structural-parent slot. Patched (write-side skip + read-side cycle guard + build-time audit), but the underlying issue is that **maps with cross-cutting writers should either be typed (NewType wrapping a forest) or carry an enforced invariant**. See `crates/layout-measurement-lowerer/src/lib.rs:2206` for the documented invariant; consider lifting this pattern to all similar maps in the index.

Smell #4 — text-shaping cache wiring is opt-in

`PersistentShapeCache` is a major performance lever (8× on large_book) but must be threaded through by the caller. The default render path (stage 15) takes `Option<&PersistentShapeCache>`. A first-class "render context" object that owns shape cache + font inventory + image inventory + media registry would simplify call sites and make the production fast path the default.

Smell #5 — text-shaping stage enum hides three sub-passes

Stages 15, 16, 17 (shape, table refine, exclusion reflow) all share the `TextShapingStats` reporting bucket. Stats come back as one number; in practice the three passes are very different work. Split the stats so a trace can localize which sub-pass is hot.

Smell #6 — exclusion reflow can fail silently

Stage 17 caps at `max_retry_passes`; if it doesn't converge it returns the last attempt's geometry without a typed error. Should emit a `Diagnostic`.

Smell #7 — composition-planner / fragmenter overlap

For flow compositions, planner generates one candidate and fragmenter commits to it. The two-stage split is only load-bearing for non-flow compositions where the cursor needs to inspect candidates. Consider a fast path that fuses them when no non-flow compositions exist; or document why the split is universal even when a single candidate exists.

Smell #8 — accessibility runs twice

Stages 23 and 26 both emit `AccessibilityNodeInstances` — pre-paint and post--paint. The post-paint version refines the pre-paint output. Two emission points means two opportunities to disagree. Ideal: emit once, post-paint, after all identity bindings are known. If a pre-paint pass is needed for reading order, name it differently and have the post-paint pass *read* it rather than re-emit.

Smell #9 — `backend-output-lowerer` and `graph-emit-ir` overlap

Both consume Paint IR. `backend-output-lowerer` records a per-target manifest (counts, losses, unsupported features); `graph-emit-ir` produces the actual `EmitIR` op list. The manifest is rarely consumed downstream — when it is, it's for diagnostics, not for emission. Consider folding the manifest into `EmitIR` as metadata, or dropping `backend-output-lowerer` into `graph-emit-ir` as a sub-pass.

Smell #10 — final metadata stage is no-op on fast path

Stage 30 emits diagnostics + invalidation + accessibility records. The production fast path (`--fast`) skips all three. The stage still runs (walks the graph, decides each is skipped, returns). Either make the stage guard return early when all three flags are set, or make the flags compose into a single "no metadata" mode at orchestration time.

Smell #11 — `handlers.rs` is 29,811 lines

`crates/faux-press/src/handlers.rs` is one file. Stages 4, 5, 33, 34 all live there. The shell layer is allowed to be promiscuous (per `layers.toml`), but a single 29k-line file is hard to navigate. Split by command (`render`, `bench`, `report.render`, `inspect`) at minimum.

Smell #12 — `renderer-pipeline/src/lib.rs` is 5,866 lines

The orchestrator. Each stage is a single function call but the wiring between them is deeply nested in one file. Per-stage submodules (`renderer-pipeline::stage::shape`, `::stage::compose`, etc.) would let each stage's input/output type be the only module boundary the next stage sees.

Smell #13 — layer assignments don't match crate names

`.arch/layers.toml` openly admits `layout-measurement-lowerer = "text-traits"` (comment: "today does both traits+composition; will split"). Several other crates are mis-named relative to their layer. Either rename the crates or update the layer comments to stop apologizing — the names should be the truth.

Smell #14 — chapter-parallel only parallelizes part of the spine

Stage 19 (composition-planning) is forced serial because the planning cursor crosses chapters. This caps the speedup. Two paths forward: (a) make the cursor explicit per-chapter and then merge; or (b) make composition-planning per-chapter and accept that cross-chapter widow/orphan decisions degrade slightly.

How to debug a specific stage

Stage X is slow

```
Shell
cargo build -p faux-press --release --locked
sample $(pgrep faux-press) 5 -file /tmp/sample.txt
```

The trace from CLI gives line-level CPU time. Combine with `--trace` flag for per-stage wall time breakdowns. The samply-based programmatic profiler at `scripts/symbolicate-samply.py` is documented in project memory.

Stage X produces wrong output

1. Render with `--trace`.
2. The IR graph is dumped per stage when trace is on.
3. Diff successive dumps; the stage that introduced a bad node is the culprit. Each node carries an `IdentityRef`, so node identity survives stage boundaries — track a single id across the dump set.

Stage X panics or hangs

1. `cargo build -p faux-press --release` with a small repro markdown.

2. If hang: `sample <PID> 5` to capture the inner loop. Look for non-terminating walks (stage 13's cycle audit emits a stderr diagnostic; recurring `graph.node(&id)` in a hot loop is project memory's "third strike" warning).
3. If panic: backtrace + stage stats from the last successful stage tell you which transition broke. Each stage's stats include node counts; a sudden drop or spike between two stages localizes the bug.

Where does invariant X live?

`MeasurementGraphIndex` invariants: `crates/layout-measurement-lowerer/src/lib.rs` around line 2206. Other indexes follow the same pattern — search for `std::collections::BTreeMap<IdentityRef` to find them. Consider promoting the documented-invariant style to every such map.

Maintenance

This document **must** be updated when:

- A pipeline stage is added, removed, reordered, or renamed.
- An IR is added, removed, or its consumer/producer changes.
- One of the smells above is fixed (delete the smell, or move it to an "addressed" archive).
- A function whose file:line is cited above moves more than ~50 lines.

Treat this file like the architecture doc — code is the source of truth, this file is a navigable index. When code disagrees, fix the doc.

Last verified against commit `26652a6f` on `main` (2026-05-28).