zenfmt

One representation, every document

THE PREMISE

Every format a converter reads is a lossy projection onto one shared representation. Every format it writes is a lossy projection back out. The engineering is in those two projections, and so is the honesty.

Zen Contributors

Contents

Preface

A document is a promise. Someone wrote something down, and someone else will read it. Between the writing and the reading sits an accident of history: the file format. The promise survives only as well as the software that carries it across.

zenfmt is a document converter written in Zig. It reads 19 input formats and writes GitHub-Flavored Markdown. The readers cover Word documents old and new, spreadsheets in four encodings, presentations in three, EPUB books, PDF files, and the lightweight markup family. The converter is also a worked answer to a harder question. What does it take to build a converter you can trust with a file you did not create? One whose losses you can enumerate, and whose output you can reproduce byte for byte?

This book describes the system that answers that question. The design decisions themselves live in the Zen Discussion records under `docs/zds/`. Those are 12 numbered documents that argued each choice before code existed. The book is the narrative companion. It walks the same territory in the order a reader learns it, not the order it was decided.

Three commitments shape every chapter that follows.

- **Lossiness is reported, not hidden.** Every conversion discards something. zenfmt names what it dropped, in diagnostics designed after Elm's error messages, and records them in a manifest beside every artifact.
- **The engine knows no formats.** No identifier from any file specification appears in the core. A format is a library. The conversion matrix is assembled by the compiler from declarative descriptors.
- **Documents are untrusted input.** Every limit is named. Every loop is bounded. Every archive is treated as a potential bomb. A refusal is a diagnostic a person can act on, never a crash and never silence.

> *Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.*
>
> - Edsger W. Dijkstra, "On the nature of Computing Science" (1984)

How to read this book

The chapters are ordered for a reader who has never seen zenfmt. They are written to be entered wherever your problem starts.

If you have a document to convert, start with Chapter 1. It follows one DOCX file from the command line to its Markdown artifact and manifest. Keep Chapter 8, the reference, under your thumb.

If you are embedding the library or writing filters, read Chapters 1 and 2 for the shared vocabulary, then go straight to Chapter 7. The filter system is the library's proudest surface. The worked example there compiles a project of your own against the zenfmt package.

If you are writing a format plugin, Chapters 2 and 3 are your contract. They give the document representation you must build and the reader obligations the engine will hold you to: tokens, reports, and limits. Chapters 4 through 6 then show every trick the shipped readers use, from CFB sector chains to PDF width metrics, and what they refuse to do.

If you are here to judge the claims, Chapter 9 is the benchmark. It states the method, the corpus, and the measurements against pandoc and anydoc. Its dashboard renders from the same machine-readable results file the build step writes.

> **Conventions** Code and element names are set in `monospace`. Transcripts show real command output, captured from the build this book compiled against. Boxed asides like this one carry definitions, predictions, and checkpoints. The book expects you to argue with it before it reveals an answer. Every chapter closes with a teach-back. If you cannot explain the chapter to a colleague in five sentences, the chapter has not finished with you.

PART I

One command

We begin with a document you did not write, in a format you did not choose, and one command. We end knowing everything that command left behind: an artifact, a manifest, a set of honest reports, and an exit code a script can branch on.

1 A Conversion, End to End

Learning contract By the end of this chapter you should be able to convert any supported document to Markdown from the command line. You should be able to read the adjacent manifest and say what each field certifies. You should be able to dissect a diagnostic into its four questions, predict the exit code of a failed run, and explain how zenfmt decides what format a file is without trusting its name.

Checkpoint: prior knowledge You need a terminal and one document: a `.docx`, a `.pdf`, an `.epub`, anything from the roster in this chapter. You do not need Zig yet. The library and its internals begin in Part II. If you already use pandoc, watch for the places where zenfmt deliberately behaves differently: the manifest, the reports, and the refusal rules.

1.1 A file arrives

Download the archive for your platform from GitHub Releases and unpack the single executable. On macOS, the repository also provides a Homebrew cask that downloads the Apple Silicon or Intel archive directly from the same release:

```
$ brew install --cask https://raw.githubusercontent.com/insanai/zenfmt/main/packaging/
homebrew/Casks/zenfmt.rb
```

The executable contains both the CLI and server. It does not need a Java, Python, Node, npm, OCR, VLM, or model runtime.

A report lands in your inbox. It is `report.docx`. It was written in Microsoft Word by someone who cared about headings and tables. You need it as Markdown, perhaps for a static site, a search index, or a language model that eats plain text. This is the whole job:

```
$ zenfmt report.docx
```

Two files appear beside the input, and two notes appear on stderr:

```
-- MERGED CELLS DEGRADED ------------------------------------ zenfmt

This document merges table cells across rows or columns.

Merged content sits in its first cell; the covered positions are empty.

What you can do:

  Keep the source:
    Keep the source document if the merged-cell layout matters.
```

The first file is `report.md`, the artifact. It is CommonMark with GFM tables. There is one blank line between blocks, a single trailing newline, and no trailing spaces. These determinism rules are part of the writer's contract. Convert the same input twice and you get identical bytes. The second file is `report.md.zenfmt.json`, the manifest. It is the part a pipeline learns to lean on.

Artifact The converted document itself. It is written atomically: zenfmt streams to a temporary file in the destination directory and renames it into place only when the conversion succeeds. A crash or a refusal never leaves a half-written output.

1.2 The manifest, field by field

Every path output gets an adjacent `<name>.zenfmt.json`. It is canonical JSON: sorted keys, no white-space, shortest number spellings. Two identical conversions produce byte-identical manifests. Here is the one from `report.docx`, pretty-printed and shortened:


```
{
  "artifact": {
    "digest": {
      "algorithm": "blake3-256",
      "value": "9739657198a2364b...c349d7254f5da96"
    },
    "format": "markdown",
    "name": "report.md",
    "plugin": { "id": "ai.insan.zenfmt.markdown" }
  },
  "ast": { "schema": "ai.insan.zenfmt.ast", "version": 2 },
  "document_metadata": {},
  "facets": {
    "layout": { "count": 1, "digest": { ... }, "unused": true }
  },
  "plugins": {},
  "reports": [ ... ],
  "schema": "ai.insan.zenfmt.artifact-manifest",
  "schema_version": 2,
  "source": {
    "digest": {
      "algorithm": "blake3-256",
      "value": "e736f49b9806935b...02fc36c8f6f658c"
    },
    "format": "docx",
    "name": "report.docx",
    "plugin": { "id": "ai.insan.zenfmt.docx" }
  }
}
```


Read it as a chain of custody.

- **source** names the input, the format zenfmt decided it was, the exact plugin that read it, and a BLAKE3-256 digest of the input bytes. Change one byte of the input and the digest says so.
- **artifact** makes the same statement about the output. The digest is computed while streaming. The bytes on disk are the bytes that were hashed.
- **reports** carries every diagnostic the conversion produced, as structured data rather than wrapped terminal text. The two stderr notes are here with their stable codes, `docx.merged-cells-degraded` and `docx.page-breaks-dropped`, their loss tier, and their directions.
- **document_metadata** is the document's own metadata (title, authors) in canonical form. **plugins** holds versioned, namespaced preservation data that readers stash for a future round trip. Chapter 3 returns to it.
- **facets** summarizes the rich-document annotations the reader attached and this writer did not consume: page geometry here, spreadsheet formulas or tracked revisions elsewhere. Each kind carries a count

and a digest; `--preserve-facets` serializes the full rows. Carried but unused is not lost, and the manifest says which it was.
- **schema** and **schema_version** let a tool refuse a manifest it does not understand instead of misreading it.

When zenfmt later converts a file that has an adjacent manifest, it loads the manifest and verifies the digest. If they match, the plugin data is carried forward. If they do not, the manifest is reported as stale and ignored. It is never trusted.

> ■ **Checkpoint: the manifest** Cover the listing above and answer three questions. Which two fields change if you edit one word of `report.docx` and convert again? Which field proves which plugin produced the Markdown? Where did the two stderr notes go?

1.3 Anatomy of a diagnostic

Conversions lose information. A page break has no Markdown spelling. A merged cell has no GFM equivalent. The position of zenfmt, argued in ZDS 0002, is simple: lossiness is reported, not hidden. Every diagnostic answers four questions, in order, every time. Mistype a format name and the structure is easy to see:

```
$ zenfmt --from docs report.docx
```

```
-- UNKNOWN INPUT FORMAT ---------------------------------------- zenfmt

I do not recognize `docs` as an input format. Did you mean `docx`?

No output file was created. These are the input formats I know: text
markdown csv docx rtf xlsx odt pptx html asciidoc rst ods odp epub pdf
doc xls ppt xlsb

What you can do:

  Select the intended format explicitly:

        zenfmt --from docx report.docx
```

Question	Where it appears	In this report
What is the problem?	The banner and the first paragraph.	`docs` is not a format zenfmt knows.
Where is it?	The banner's right side: a file, or `zenfmt` itself when the problem is the invocation.	The invocation.
What did zenfmt do about it?	The consequence line.	It refused. No output file was created.
What can you do?	The final section. Always concrete, often a complete command.	Run again with `--from docx`.

The style is borrowed deliberately from the Elm compiler. A diagnostic is a small piece of technical writing addressed to a person. It is not a log line addressed to a grep. Each report also carries a stable code, so scripts can match on meaning rather than on prose that may improve between releases.

The four questions are a minimum structure, not the goal by themselves. The final answer must be usable: it names the exact limit, path, format, or construct involved and gives a concrete correction, often a

command that can be copied. “Try again” or “check the input” does not qualify as a zenfmt direction unless the report also says what to check and why.

> **Loss tier** Every loss report classifies itself. **Degraded** means the content survived in a simpler form, like a merged cell's text sitting in its first cell. **Dropped** means the content is absent, like a page break. The tier lives in the manifest, so a pipeline can accept degradations while rejecting drops.

1.4 Exit codes

The process exit code is the report system folded down to one integer:

Code	Class	Meaning
0	success	An artifact and manifest were committed. Notes and warnings may have been printed. Success does not mean lossless.
1	conversion	The input could not be converted: a malformed container, invalid XML, a contradiction the reader detected.
2	usage	The invocation is wrong: an unknown format, a missing `--from` on stdin, a refusal to overwrite an existing output.
3	limit	A resource limit refused the input: a zip bomb, an absurd nesting depth, an oversized entry. Chapter 6 shows every limit.

A bulk script wants exactly this split: fix the invocation on 2, quarantine on 3, log and move on for 1.

1.5 The commands you will actually run

```
$ zenfmt report.docx                  # report.md + report.md.zenfmt.json
$ zenfmt report.docx -o notes.md      # choose the output path
$ zenfmt report.docx --stdout         # document bytes only, to stdout
$ zenfmt - --from html --stdout < page.html   # stdin needs --from
$ zenfmt report.docx --overwrite      # replace an existing artifact
$ zenfmt report.docx --strict         # refuse dropped content, pre-commit
$ zenfmt report.docx --strict=exact   # refuse any declared loss at all
$ zenfmt report.docx --preserve-facets   # full facet rows in the manifest
$ zenfmt report.docx --reports json   # structured diagnostics on stderr
$ zenfmt big.xlsx --limit max_input_bytes=2147483648
```

Three behaviors deserve emphasis. First, overwriting is a refusal by default. If `report.md` exists, zenfmt reports it and exits 2 rather than clobbering it; `--overwrite` replaces each staged file and publishes the new manifest last. Second, strict mode fails early and is graded. Bare `--strict` refuses when the conversion would drop semantic content; `--strict=structure` also refuses structural degradation; and `--strict=exact` refuses any declared loss at all, styling included. In every grade the refusal happens before anything is committed, so a pipeline that cannot tolerate loss gets no partial output. Third, `--stdout` writes only document bytes. Reports go to stderr, the manifest is dropped unless you ask for it with `--metadata-out PATH`, and an embedding application can additionally ask the result whether the stream received nothing, a partial prefix, or the complete artifact.
Here is the roster. This transcript is real, and Part III walks every row:

```
$ zenfmt --list-formats
Readers:
  text .txt .text
  markdown .md .markdown
  csv .csv .tsv
```

```
  docx .docx .docm
  rtf .rtf
  xlsx .xlsx .xlsm
  odt .odt
  pptx .pptx .pptm .ppsx .ppsm
  html .html .htm
  asciidoc .adoc .asciidoc
  rst .rst
  ods .ods
  odp .odp
  epub .epub
  pdf .pdf
  doc .doc
  xls .xls
  ppt .ppt .pps .pot
  xlsb .xlsb
Writers:
  markdown .md .markdown
```

That is 19 readers and 1 writer. The single writer is a design position, not a gap. The intermediate representation is easiest to judge, and to keep honest, while many readers feed one consumer.

1.6 What is this file, really?

Predict before reading on You rename `slides.pptx` to `slides.zip` and run `zenfmt slides.zip`. Write down what you expect before reading on: an error, a PowerPoint conversion, or something else?

The extension is only the first hint. When it matches a registered reader, that reader is chosen, and for in-memory input the bytes are still checked: if the content signature names a different format, the content wins and a `core.extension-mismatch` note says so, which is how `report.docx` holding RTF parses as RTF instead of failing confusingly. When nothing matches, or there is no extension at all, zenfmt reads the bytes and looks for content signatures:

The signatures are worth knowing, because they explain behavior that otherwise looks spooky.

- `PK\x03\x04` opens a ZIP. The specific format comes from part names that are visible in the central directory without extracting anything. `word/document.xml` means DOCX. `xl/workbook.bin` means XLSB, and it is checked before `xl/workbook.xml`, which means XLSX. `ppt/presentation.xml`

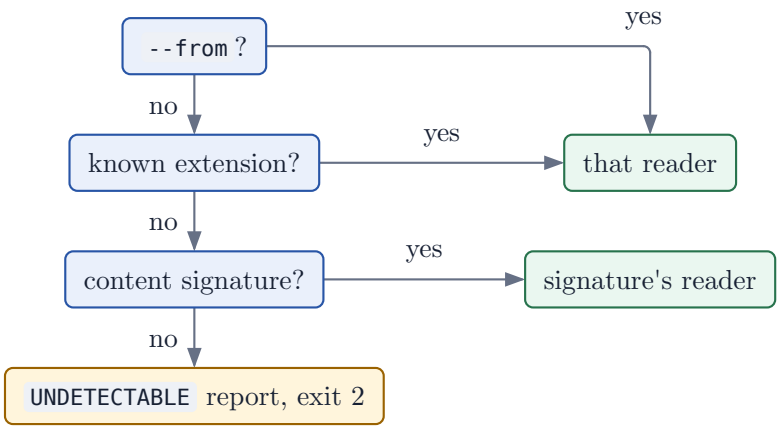

Figure 1: Format detection: explicit flag, then extension, then content signatures. No signature ends in a report, never in a guess.

Figure 2: The conversion pipeline. One representation in the middle. The validator guards both sides of the filter stage, and the lowering plan prices every loss before the writer runs.

means PPTX. An OpenDocument or EPUB file declares itself in its `mimetype` entry: `application/vnd.oasis.opendocument.spreadsheet` is ODS, and `application/epub+zip` is EPUB.

- `\xd0\xcf\x11\xe0\xa1\xb1\x1a\xe1` opens a legacy Compound File. Its directory names its streams in UTF-16LE. `WordDocument` means `.doc`. `Workbook` or `Book` means `.xls`. `PowerPoint Document` means `.ppt`.
- `%PDF` is a PDF. `{\rtf` is RTF.

So consider the renamed `slides.zip`. The extension matches no reader. The sniff finds `ppt/presentation.xml`. The PPTX reader converts it as if nothing happened. The name was wrong; the bytes were not.

■ **Encrypted inputs are refused, always** A password-protected DOCX, an RC4 `.doc`, a DRM'd EPUB, an encrypted PDF: each is detected and refused with its own report code, such as `epub.drm-refused` or `pdf.encryption-refused`. No override flag exists. A converter that emitted gibberish from bytes it could not actually read would be lying about what it did.

1.7 The road a document travels

Every conversion in this book, every format, every chapter, makes the same seven stops:
The reader parses one format and emits nodes, attaching typed facets for what Markdown alone cannot hold. The tree is the single representation of Part II. The validator proves structural invariants before anything else may touch the tree, and proves them again after every filter stage. Filters are Zig transforms compiled into your binary (Chapter 7). The lowering plan records and prices every degradation the writer declares, which is where the graded `--strict` refusal happens. The writer renders Markdown deterministically. Commit publishes the artifact, then any extracted media, then the manifest. Each rename is atomic; the manifest appears last and certifies the complete ensemble. Nothing in the middle knows what format the bytes came from. That ignorance is enforced at compile time, and it is the subject of Chapter 3.

Teach it back. Explain to a colleague, in four sentences, what `zenfmt report.docx` leaves on disk. Then explain how a script they write can verify the output was not tampered with, detect that content was dropped, and distinguish a zip bomb from a mistyped flag.

Exercise 1.1. Convert any office document you have with `--reports json` and find the stable code of each report. Then convert it again and diff both the artifact and the manifest. What changed?
Hint: Nothing should change. Determinism is a stated contract of the writer and the manifest encoder.

Exercise 1.2. Create `notes.md` with some content. Then run a conversion whose output path collides with it, without `--overwrite`. What exit code do you get, and what does the report tell you to do? Verify that `notes.md` was not touched.
Hint: Refusals happen before the rename. The temporary file is discarded.

Exercise 1.3. Take a `.xlsx` file, strip its extension (`cp sheet.xlsx mystery`), and convert it. Which of the three detection stages fired? How would you prove it from the manifest alone?
Hint: The manifest's `source.format` records the decision.

PART II

One representation

There are 19 readers, 1 writer, and between them a single document tree. This part builds that tree twice: first as the idea a filter author holds in their head, then as the flat arrays the machine actually walks.

2 One Representation

Learning contract By the end of this chapter you should be able to name the block and inline node sets, lay out a small document as flat preorder columns, and hop between siblings using `subtree_len`. You should be able to explain the schema table that every rule about a tag derives from, say what an entity is and why most nodes never get one, name the five facet kinds and the erasure rule that keeps them optional, and state the invariants the validator proves after reading and after every filter stage.

Checkpoint: prior knowledge You should have converted at least one document (Chapter 1). Zig familiarity helps from here on: `enum(u32)`, tagged unions, and slices appear on every page. If `std.MultiArrayList` is new to you, this chapter is also a worked introduction to it.

2.1 A tree you already believe in

Ask anyone to sketch a document and they draw the same thing: a heading, then a list, one item holding a paragraph, another holding two. A tree. The question a converter must answer is not whether the middle representation is a tree. The question is which tree, and how it is stored.

The answer to "which tree" is a Zig-native node set. It is structurally compatible with pandoc's document model but owned by this project. Blocks nest. Inlines nest. Nodes carry identifiers, classes, and key-value attributes. The full roster, from `core/src/ast.zig`:

Block tags	Role
`plain`, `paragraph`, `line_block`, `heading`, `code_block`, `raw_block`, `quote`, `list`, `definition_list`, `thematic_break`, `table`, `figure`, `container`	Public content: what `Document.block` returns for body positions.
`extension`	A namespaced plugin construct. Its children are a mandatory source-neutral fallback, so a writer that does not understand the extension still has something honest to emit.
`line`, `list_item`, `definition_entry`, `definition_term`, `definition_body`, `caption`, `table_head`, `table_body`, `table_foot`, `table_row`, `table_cell`	Normalized structure. Each appears only under its governing parent. A `table_row` exists only inside a table section. A `list_item` exists only inside a `list`.

Inline tags	Role
`text`, `space`, `soft_break`, `hard_break`	The prose layer. Text nodes never contain whitespace. Whitespace is always one of the three break or space nodes, so writers control wrapping exactly.
`emphasis`, `strong`, `strikethrough`, `superscript`, `subscript`, `small_caps`, `underline`, `quote`, `span`	Styling containers, nested in one canonical order (Chapter 3).
`code`, `math`, `raw`	Verbatim leaves with payloads.

Inline tags	Role
`link`, `image`, `note`, `citation`	Containers with side-table payloads: a target, a resource reference, a deferred note body, a citation record.
`extension`	The inline spelling of the namespaced construct, with fallback children.

Two tags keep this set closed for ordinary structure: `container` (a block with attributes) and `span` (an inline with attributes). An AsciiDoc sidebar, an HTML `<section>`, a DOCX content control, an ODP speaker-notes page: each becomes a `container` with a class, not a new node type. If a format ever truly cannot be expressed this way, the node set is wrong. That is a design alarm, not an extension point.
The `extension` tags are the one sanctioned escape hatch, and they pay for the privilege (ZDS 0013). An extension names its owner in reverse-DNS form, names itself, and carries a version. Its children are not optional decoration; they are the fallback every other writer renders. The validator refuses an extension with no fallback, and it refuses an extension nested inside another extension of the same owner. The first producer is the HTML reader, which maps `<details>` to an extension whose fallback is the summary line and the disclosed content.

■ **Preorder** Parent first, then each child subtree in order, recursively. Writing a tree down in preorder produces exactly the order your eye reads a document in. That is why a preorder array can be the document, with no pointers at all.

2.2 The shape of the storage

The obvious implementation allocates a node object per element and links the objects with pointers. zenfmt instead does what Zig's own compiler does for `std.zig.Ast`. Every block lives in one `std.MultiArrayList`, a struct-of-arrays, as parallel columns. The tree structure is encoded in a single integer per node.

```
// core/src/ast.zig (storage row, private)
pub const Block = struct {
    tag: BlockTag,          // 1 byte
    payload: u32,           // meaning depends on tag
    attrs: OptionalAttrsIndex,
    inlines: InlineRange,   // for leaf blocks: its prose
    subtree_len: u32,       // this node plus all descendants
};
```

A test asserts that the columns of one block row sum to 21 bytes. A million-block document costs 21 MB of block storage. It is contiguous, and it is allocated out of one arena that is freed in one call when the conversion ends. Note what the row does not contain: no entity field, no facet pointer, no style handle. Rich-document identity lives outside the row, in side tables that cost nothing when unused. That design is this chapter's second half.

■ **Arena** One allocator per conversion. Every node, string, and side-table row is allocated from it, and none is individually freed. `Conversion.deinit` releases the whole document at once. There is no per-node `free`, because there are no per-node allocations.

2.3 The worked example

ZDS 0002 fixes the layout with a small document. It is reproduced here because every walker in the codebase is a loop over these columns.

```
## The *quick* [brown](http://x) fox

- one
- two

  still two
```

i	block	payload	subtree len	inlines
0	`heading`	level 2	1	0..9
1	`list`	unordered	6	none
2	`  list_item`	none	2	none
3	`    plain`	none	1	9..10
4	`  list_item`	none	3	none
5	`    paragraph`	none	1	10..11
6	`    paragraph`	none	1	11..12

The indentation in the table is commentary. It is not stored. The structure lives entirely in `subtree_len`. Node 1's subtree is 6 rows long, so the document's next top-level block, if there were one, would sit at index $1 + 6 = 7$. Node 2's subtree is 2, so its sibling item sits at $2 + 2 = 4$. Walking children is a short loop. There is no recursion and no call stack:

```
var child = parent + 1;
const end = parent + subtree_len[parent];
while (child < end) : (child += subtree_len[child]) {
    // visit `child`
}
```

API anchor: `Document.blockChildren(index)` Returns the child iterator that packages this hop. The same arithmetic drives inline children, the validator, the Markdown writer's frame stack, and the filter rebuild pass. in `core/src/ast.zig`

The heading's prose is the inline range 0..9 in the parallel inline table. The encoding is the same, with 13-byte rows:

j	inline	payload	subtree len
0	`text`	`"The"`	1
1	`space`	none	1
2	`emphasis`	none	2
3	`  text`	`"quick"`	1
4	`space`	none	1
5	`link`	`targets[0]`	2
6	`  text`	`"brown"`	1
7	`space`	none	1

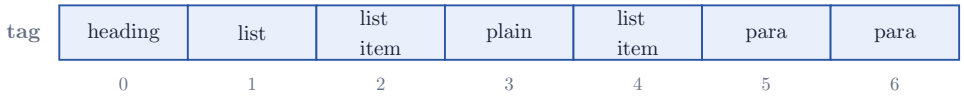

Figure 3: The tag column of the block table for the worked example. The document is this array. There is nothing else to point to.

j	inline	payload	subtree len
8	`text`	`"fox"`	1

Hop from 0 and you visit exactly the heading's 7 top-level children. The hop skips the insides of `emphasis` and `link` without ever testing what they are.

> **Predict before reading on** A filter deletes block 3, the `plain` inside the first item. Which `subtree_len` values must change, and which must not? Write the affected indices down before reading Chapter 7's answer.

2.4 Payloads live in side tables

The `payload` column is a bare `u32`. Its meaning depends on the tag. For `heading` it is a level. For `list` it is an index into a table of list properties. For `link` and `image` it is an index into `targets`. For `code` it is a range in the text pool. For `extension` it is an index into the extensions table holding the owner, the name, and the version. These side tables (targets, headings, lists, table properties, extensions, citations, metadata) are append-only arrays in the same `Store`.

Append-only is a load-bearing property. Side-table indices are never invalidated, so a filter that rebuilds the node arrays can leave every payload index untouched and share the tables wholesale. An identity filter costs one `memcpy` per column and zero side-table work. The test suite measures this claim rather than asserting it.

Strings follow the same discipline. All prose lives in one UTF-8 text pool. A `text` node's payload is a range into that pool. Extracted binary content, such as image data a reader pulled out of a PDF or a DOCX, lives in a separate resource pool, described below.

2.5 One schema, one row per tag

Every fact stated so far raises the same maintenance question: where do the rules about a tag live? What children may a `table` hold? Which side table does a `citation` payload index? Before ZDS 0013 the answers lived in a set of coordinated switch statements. The compiler enforced that every switch covered every tag, but nothing enforced that the switches agreed with one another.

The rules now live in one comptime table in `core/src/schema.zig`, one row per tag:

```
pub const BlockRow = struct {
    tag: BlockTag,
    payload: PayloadKind,   // which side table, or none
    children: ChildRule,    // flow, only: {...}, inline_content, none
};
pub const block_schema = [_]BlockRow{
    .{ .tag = .paragraph, .payload = .none, .children = .inline_content },
    .{ .tag = .list, .payload = .list, .children = .{ .only = &.{.list_item} } },
    // ... one row per tag, checked complete at compile time
};
```

Content kind, placement rules, payload validation, debug names, and the writer capability checks of Chapter 5 are all derived from this table by comptime iteration. A property that used to be five agreeing

switches is now one row read five ways. Drift between the rules is no longer expressible, and a new tag fails compilation everywhere at once until its row exists.

Why a table beats discipline The old switches could not drift in coverage, because Zig demands exhaustive switches. They could drift in content: the placement switch and the validator could assert different child kinds for one tag, and only a test would notice. The table makes that disagreement a type error. This is the same trade the whole chapter makes: move a promise from review time to compile time.

2.6 Storage is private; views are typed

No format library reads those columns directly. The public face of a node is a tagged union built in `core/src/payload.zig` from the schema table's payload kinds:

```
switch (doc.block(index).content) {
    .heading => |h| // h.level, h.inlines: InlineRange
    .list => |l|    // l.kind, l.start, l.items: BlockRange
    .extension => |x| // x.owner, x.name, x.version, x.fallback
    else => {},
}
```

API anchor: `BlockView` / `InlineView` Tagged-union views over the flat storage, driven by the schema table in `core/src/schema.zig`. A new tag cannot be half-added: the compiler demands every switch arm, everywhere, at once, in `core/src/payload.zig`

This split is the chapter's central trade. Storage is chosen for the machine: cache-friendly columns, one arena, integer links. Views are chosen for the person: exhaustive switches, named fields, no way to read a payload as the wrong type. Neither leaks into the other.

Attributes follow the pattern. A node's `attrs` is an optional index into an attrs table holding an identifier, classes, and key-value pairs, with interned keys. Metadata is a small value model: null, bool, int, float, string, inlines, blocks, list, map. Its maps keep bytewise-sorted keys, so the manifest encoding is canonical without a sort at write time. Footnotes are the one deliberate indirection. A `note` inline carries only a reference. Note bodies are collected separately by the builder (`declareNote`, then `beginNoteBody` after the main flow), because every target format renders note bodies out of line.

2.7 Entities and facets

Markdown consumes none of what this section describes, and that is the point. A DOCX paragraph has a named style. A PDF line has a page and a position. A spreadsheet cell has coordinates and a formula. Future writers want that information; the flow representation must not pay for it. ZDS 0013's answer is stand-off annotation: typed facet rows stored beside the tree, joined to nodes by identity, ignored freely.

Entity A stable logical identity, `EntityId`, shared by a kernel node and its facets. Identity is lazy: a node gets an entity the first time a facet attaches to it, and never otherwise. The binding lives in a side table of (node index, entity) rows, ordered by node, so a document with no facets carries no entity storage at all and the 21-byte block row never grows.

Five facet kinds ship, each an append-only table sorted by entity:

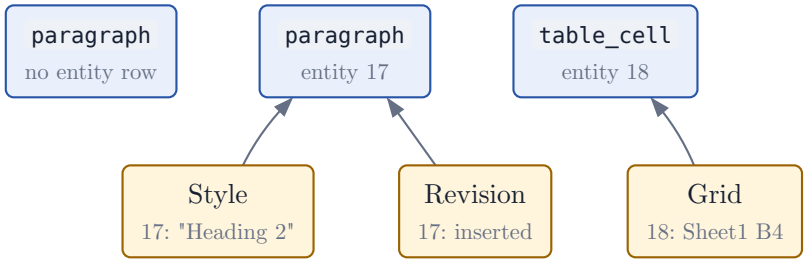

Figure 4: Stand-off binding. Two of three nodes carry entities; facet rows reference the entities, never the node rows. The first paragraph costs nothing.

Facet	Carries
`Provenance`	Producing plugin, source member (an archive part, a page label, a spine item), byte position, and a confidence grade: DOCX paragraphs are facts, PDF paragraphs are projections.
`Style`	Named style, semantic role, language tag, text direction.
`Layout`	Page, slide, or canvas identity plus a box: x, y, width, height, and z-order, in EMU with a top-left origin.
`Grid`	Sheet name, row and column, value type, formula source text, the cached value as the source spelled it, and merge extents.
`Revision`	Insertions, deletions, comments, and bookmarks, with author and timestamp.

One rule keeps the whole arrangement sane, and the validator enforces its checkable consequences:

> ■ **Facet erasure** Erasing every facet table leaves a document that renders meaningfully. Facets refine kernel semantics; they never carry primary content. Text lives in the text pool, never in a facet. A link target or a list structure is never facet-only. This is why the Markdown writer is provably untouched by any facet any future reader invents: the kernel it reads means the same thing with the facet tables ignored.

Layout coordinates deserve their own sentence, because the unit was chosen by arithmetic rather than taste. Every `Layout` box is stored in EMU, English Metric Units, 1/914400 of an inch, with a top-left origin. One point is exactly 12700 EMU. One CSS pixel is exactly 9525. One millimeter is exactly 36000. DOCX and PPTX geometry is EMU natively, so office coordinates transfer without rounding, and PDF points round at a scale of nanometers. A reader normalizes once, at read time, in the one place that knows its source geometry.

Facets survive filtering. The rebuild transform of Chapter 7 records which node ranges it copied, rebases the entity bindings across those ranges in one ordered merge, and hands the new snapshot its own slice of binding rows. A node that a filter drops takes its bindings with it, and its facets become unreachable, exactly as a dropped paragraph's prose does. This is Lemma 2 of ZDS 0013, running as code.

2.8 The resource store

Extracted binary content generalizes the same way. When a reader pulls image bytes out of an archive or a PDF stream, it registers them in the resource store: source name, MIME type, the bytes in a binary pool, and a BLAKE3 digest computed once at registration. On path output the engine writes each resource into a `_media` directory beside the artifact, rewrites matching image URLs, and lists every file with its digest in the manifest, reusing the digest it already has. Stream output extracts nothing and keeps source URLs, so piping stays pure.

2.9 The validator

Everything above is an invariant that something else depends on. The validator in `core/src/ast.zig` proves all of it, with fixed-size explicit stacks and no recursion:

No.	Invariant (abridged; the code is the record)
1	Every `subtree_len` is at least 1, and subtrees nest exactly: children tile their parent's extent with no gap and no overlap.
2	Structural tags appear only under their governing parent: `table_row` under a table section, `list_item` under `list`, and so on, exactly as the schema table's placement rules state.
3	Leaf blocks own disjoint, in-bounds inline ranges. Container blocks own none.
4	Every payload index is in bounds for its tag's side table, and payload values are legal: heading levels 1 to 6, sane spans, and so on.
5	Inline containers that must hold children do. `text` nodes are non-empty and never adjacent to another `text`.
6	Whitespace appears only as `space`, `soft_break`, or `hard_break` nodes, never inside `text`.
7	Attribute rows are in bounds, and interned strings resolve.
8	Note references point at declared notes, and every declared note has exactly one body.
9	Metadata maps keep bytewise-sorted, unique keys.
10	Ranges into the text pool are in bounds, and the whole pool is valid UTF-8.
11	Nesting depth stays within the configured `max_depth`.
12	An `extension` has a non-empty fallback, and no extension nests inside an open extension of the same owner.
13	A snapshot's entity rows are in bounds and strictly increasing by node index, which makes the binding injective for free.
14	Facet tables are sorted by entity, single-valued kinds hold one row per entity, and every facet string range lands inside the text pool.

The validator runs after every reader, and again after every filter stage. That placement is the architecture's honesty mechanism. A reader bug cannot smuggle an impossible tree into the writer. A filter bug is caught at the stage that introduced it, not three stages later. When the validator rejects, the conversion fails with `core.invalid-document-tree`, and the report names the plugin at fault. The tree is evidence, and the report says whose.

Side table An append-only array in the `Store` holding one payload family: targets, list properties, facets, resources, and so on. Node columns index into it. Rebuilds share it. Indices are stable for the life of the conversion.

The pool invariant bites While the media pipeline was built, image bytes were first appended to the text pool. Invariant 10 failed the whole conversion, exactly as designed. Binary content now has its own resource pool. When the validator stops you, the correct response is rarely to weaken the invariant.

Teach it back. Using only the worked-example tables, explain to a colleague how to find the second list item without visiting the first item's children, and how to find the text of the link. Then explain

why a spreadsheet cell's formula lives in a facet rather than in the cell's payload, and what the Markdown writer has to do about that facet. The correct answer to the last part is one word.

Exercise 2.1. Write the block table (tags and `subtree_len`) for a block quote containing one line, followed by a thematic break, followed by one paragraph. Check your `subtree_len` values by verifying the sibling hop lands on each top-level block.

Hint: Does a lone line inside a quote produce `paragraph` or `plain`? Convert one and count.

Exercise 2.2. The inline row is 13 bytes and the block row is 21. Estimate the node storage for a 1,000-page report, and compare it with the size of the DOCX it came from. What dominates the arena in practice?

Hint: The text pool holds the prose exactly once.

Exercise 2.3. Invariant 5 forbids adjacent `text` siblings. Find the builder feature (Chapter 3) that makes it impossible for a well-behaved reader to violate this, even when the reader appends text one character at a time.

Exercise 2.4. Convert a spreadsheet with `--preserve-facets` and read the `facets` object in the manifest. For one grid row, verify by hand that the `entity` it names resolves to a `table_cell` holding the cached value as its text.

Hint: Chapter 7 documents the manifest layout; the rows are sorted by entity.

3 Readers, Writers, Bundles

Learning contract By the end of this chapter you should be able to write a working reader for a format of your own invention. You should be able to explain every field of its descriptor and every obligation the engine holds it to. You should be able to compose readers and writers into a `Bundle`, and describe how a plugin's preservation data survives a round trip through Markdown.

Checkpoint: prior knowledge Chapter 2's tree and Chapter 1's reports. You will write real Zig here. Have `zig build test` working in a checkout, because this chapter's toy reader is meant to be typed in, not admired.

3.1 The engine knows no formats

Grep `core/` for `docx`. You will find nothing: no format name, no element name, no magic number from any file specification. The absence is enforced by construction. The engine learns formats only as descriptor tables handed to it at compile time, and each descriptor is one comptime-validated value exported by a format library.

```
pub const reader = core.Reader(.{
    .id = "ai.insan.zenfmt.docx",
    .format = "docx",
    .extensions = &.{ "docx", "docm" },
    .input = .seekable,
    .data_version = 1,
    .read = @import("package.zig").read,
});
```

Field	Contract
`id`	Reverse-DNS, lowercase ASCII, at least one dot. It names the plugin in manifests and reports, and it keys the plugin's preservation-data namespace.
`format`	The name users type after `--from`. Lowercase letters, digits, hyphens.
`extensions`	Detection hints and derived output names. Primary first, no dots.
`input`	`.bytes`, the default, delivers the whole input as one slice. `.seekable` marks container formats: they receive a file-backed input, the ZIP layer reads it through bounded windows, and piped input spills to a temporary file first.
`data_version`	Versions the plugin's preservation-data namespace. Bump it, and old carried data is dropped rather than misread.
`read`	The one function the engine will call.

Get a field wrong and the compiler files the report, in the same four-question voice the runtime uses. From `core/src/plugin.zig`, lightly abridged:

```
plugin id `Minutes` is not valid. A plugin id is reverse-DNS ASCII:
lowercase letters, digits, hyphens, and at least one dot, as in
`ai.insan.zenfmt.docx`. Change the `.id` field of this descriptor.
```

A descriptor mistake is a compile error. A malformed document is a runtime report. Nothing is a crash.

3.2 Inventing a format

To see the whole reader contract in one sitting, we invent a format small enough to fit on a page. Minutes is a line-based meeting-notes format:

```
!! Sprint 41 review
@ 2026-08-07
- decided: ship the media pipeline
- open: PDF table thresholds
Ana: the validator caught the pool bug immediately.
```

`!!` opens a heading. `@` is metadata. `-` items form a list. A `Name:` prefix marks a speaker paragraph. Here is the complete reader, and it compiles against the real API:

```
const std = @import("std");
const core = @import("zenfmt_core");

pub const reader = core.Reader(.{
    .id = "com.example.minutes",
    .format = "minutes",
    .extensions = &.{"minutes"},
    .data_version = 1,
    .read = read,
});

fn read(ctx: *core.ReadContext) core.ReadError!void {
    var lines = std.mem.splitScalar(u8, try ctx.inputBytes(), '\n');
    var list: ?core.builder.BlockToken = null;
    while (lines.next()) |line| {
        if (line.len == 0) continue;
        if (std.mem.startsWith(u8, line, "!! ")) {
            try closeList(ctx, &list);
            const heading = try ctx.out.beginHeading(1);
            try ctx.out.text(line["!! ".len..]);
            ctx.out.endBlock(heading);
        } else if (std.mem.startsWith(u8, line, "@ ")) {
            try ctx.out.metaString("date", line["@ ".len..]);
        } else if (std.mem.startsWith(u8, line, "- ")) {
            if (list == null) list = try ctx.out.beginList(.unordered);
            const item = try ctx.out.beginBlock(.list_item);
            const body = try ctx.out.beginPlain();
            try ctx.out.text(line["- ".len..]);
            ctx.out.endBlock(body);
            ctx.out.endBlock(item);
        } else {
            try closeList(ctx, &list);
            const paragraph = try ctx.out.beginParagraph();
            if (std.mem.indexOf(u8, line, ": ")) |colon| {
                const strong = try ctx.out.beginInline(.strong);
                try ctx.out.text(line[0..colon]);
                ctx.out.endInline(strong);
                try ctx.out.text(line[colon + 1 ..]);
            } else {
                try ctx.out.text(line);
            }
            ctx.out.endBlock(paragraph);
        }
```

```
    }
    try closeList(ctx, &list);
}

fn closeList(ctx: *core.ReadContext, list: *?core.builder.BlockToken) !void {
    if (list.*) |token| {
        ctx.out.endBlock(token);
        list.* = null;
    }
    return;
}
```

Four properties of the API are doing quiet work here.

First, tokens balance the tree. `beginHeading` returns a `BlockToken` that `endBlock` must receive. Close tokens out of order and an assertion names the exact mismatch during development. The builder maintains `subtree_len`; your reader never touches it.

Second, `text()` canonicalizes. We passed whole lines, whitespace included. The builder splits on whitespace into `text` and `space` nodes and coalesces adjacent text. This is the feature Exercise 2.3 was fishing for. A reader cannot produce the adjacent-`text` trees the validator forbids, even if it appends one byte at a time.

Third, metadata is one call. `metaString` lands in the document's metadata map, which lands in the manifest, sorted and canonical.

Fourth, everything allocates from `ctx.gpa`, the conversion's arena. Note what is absent: no `free`, no ownership transfer, no lifetime puzzle. The conversion ends, the arena goes, and the reader never cleans up.

Beyond this toy, five more Emitter families matter in real readers. Staged attributes: `ctx.out.attrs(.{ .classes = &.{"notes"} })` applies to the next opened node. This is how ODP marks its speaker-notes container. Deferred notes: `declareNote` mid-flow, then `beginNoteBody` and `endNoteBody` after the main flow. This is the shape footnotes take in DOCX, RTF, and ODT. Resources: `ctx.out.resource(source, bytes, mime)` registers extracted image bytes under a `ResourceId`, and the engine commits them beside the artifact. Facets: every token can carry typed stand-off annotation, which is how rich-document meaning survives a Markdown-shaped kernel. One real call from the XLSX reader:

```
try ctx.out.attachGrid(cell, .{
    .sheet = sheet_name,
    .row = row_index,
    .col = column,
    .value_type = .number,
    .formula = formula_text,
    .cached = cached_text,
});
```

The facet rule is ZDS 0013's erasure axiom: a facet refines, it never carries primary content. The kernel must render meaningfully with every facet table empty, and the validator enforces the checkable half of that promise. Extensions, finally, are the escape hatch for constructs the kernel does not model: `beginExtension(owner, name, version)` opens a namespaced node whose children are a mandatory source-neutral fallback. A writer that understands the namespace uses the extension; a writer that does not renders the fallback and reports the loss. The validator rejects an empty fallback and a same-owner extension nested inside another.

> **API anchor: `core.ReadContext`** Everything a reader receives: `gpa` (the arena), `input` (bytes, or a file handle for `.seekable` readers, with `inputBytes()` as the whole-slice shim), `input_name`, `out`

(the Emitter), `reports`, `limits`, and `preservation()` for its version-compatible carried namespace. A reader windows what it needs; it never opens paths of its own. in `core/src/plugin.zig`

3.3 The obligations

The engine holds every reader, shipped or third-party, to the same four rules. They are worth stating plainly, because Part III is largely the story of applying them to hostile real-world formats.

1. Every loss is a report. Drop a construct silently and the format record review will find it. The merged-cells note from Chapter 1 is the pattern. Reports carry stable codes, and the codes appear in tests.
2. Facets refine; they never carry content. Anything a writer must see to render correctly belongs in the kernel. Anything that enriches it, a style name, a cell formula, a page position, belongs in a facet, where it costs nothing to the conversions that ignore it.
3. Limits are honored, not interpreted. `ctx.limits` caps input size, nesting depth, archive expansion, node and facet counts, and resource bytes. Hitting a limit is a refusal with exit class 3. It is never an invitation to be clever.
4. No recursion. Every walker is a loop over an explicit, bounded stack. A document's nesting depth must not become your call depth, because the attacker chooses the document.
5. The validator has the last word. Whatever your reader emits is proven against Chapter 2's invariants before the pipeline continues. You cannot opt out. After a while you will not want to: it is the best free test harness a reader author gets.

● **Predict before reading on** Two readers both claim the extension `md`. When does the mistake surface: when the second library is written, when both are imported, when a bundle lists both, or at runtime on the first `.md` file?

3.4 Bundles: composition at compile time

A format library exports descriptors. A bundle composes them into an engine:

```
pub const Default = core.Bundle(.{
    .readers = .{
        text.reader,     markdown.reader, csv.reader,  docx.reader,
        rtf.reader,      xlsx.reader,     odt.reader,  pptx.reader,
        html.reader,     asciidoc.reader, rst.reader,  ods.reader,
        odp.reader,      epub.reader,     pdf.reader,  doc.reader,
        xls.reader,      ppt.reader,      xlsb.reader,
    },
    .writers = .{markdown.writer},
});
```

`Bundle` validates the table at compile time. Duplicate format names, two readers claiming one extension, a reader and writer of the same format with different ids: each is a named `@compileError`. That answers the prediction above. The mistake surfaces when a bundle lists both, and never later. From the validated table the compiler generates the routing (an `inline for` that turns a runtime format string into a comptime-dispatched call), the `--list-formats` output, the did-you-mean suggestions, and the detection tables. The compiler writes the registry, so no runtime registration code can drift from it.

The umbrella library's `Default` bundle is what the CLI ships. Your application can assemble a smaller one, with 3 formats and your own filter pipeline, and get a proportionally smaller binary. Nothing about the default is privileged.

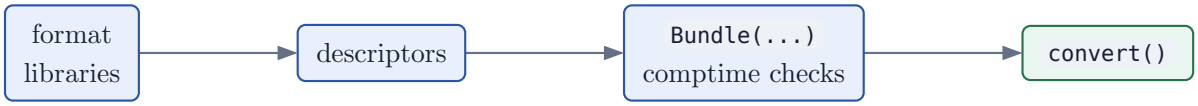

Figure 5: A bundle at build time: descriptors in, one engine out. The engine's format knowledge is this table and nothing else.

3.5 The writer side

One writer ships: Markdown. Its descriptor mirrors the reader's, plus one field the reader has no analogue for: the capability declaration of ZDS 0013.

```
pub const writer = core.Writer(.{
    .id = "ai.insan.zenfmt.markdown",
    .format = "markdown",
    .extensions = &.{ "md", "markdown" },
    .write = write,
    .capabilities = &capabilities,
});

pub const capabilities: core.lowering.Capabilities = .{
    .exact_blocks = &.{ .paragraph, .heading, .quote, .list, ... },
    .lowered_blocks = &.{ .definition_list, .table_cell, .extension, ... },
    .exact_inlines = &.{ .text, .emphasis, .strong, .code, ... },
    .lowered_inlines = &.{ .underline, .citation, ... },
    .rules = &rules, // each with a name, a priced LossCost, a note
};
```

The declaration is validated for totality at compile time: every kernel tag must be exact, lowered, or refused, so a new tag fails this writer's build until someone decides its disposition. At run time the writer's emission sites record rule hits on `ctx.plan`; the engine prices the plan, gates the graded `--strict` predicate before anything is committed, and flushes one aggregated loss report per fired rule. A writer can also recover what its own format library preserved: `ctx.preservation()` returns only that writer descriptor's namespace from the input's verified manifest, and only when `data_version` matches. `ctx.preservationAs(T, decode)` is the typed decoding path. The engine never gives a plugin a runtime namespace selector or another plugin's data.

A writer walks views with the same sibling hop the validator uses, and it owns its output discipline completely. The Markdown writer's escaping rules, tight-list handling, and note placement fill Chapter 5. What a writer cannot do is reach back. The document is `*const`, and the type system means it.

3.6 Preservation data: the round trip's memory

Markdown cannot represent everything a reader saw. That is what the reports are for. But some of what Markdown cannot hold is worth carrying rather than merely mourning. A reader may attach one namespaced, versioned JSON value to the conversion:

```
ctx.own_plugin_data = .{
    .version = 1,
    .data = "{\"paragraph_style_ids\":[\"Quote\",\"Intense\"]}",
};
```

The DOCX reader does exactly this with the paragraph style ids it had to flatten. In the manifest, the value appears under the plugin's id:

```
"plugins": {
  "ai.insan.zenfmt.docx": {
```

```
    "data": { "paragraph_style_ids": ["Quote", "Intense"] },
    "version": 1
  }
}
```

When a later conversion reads that Markdown file, the adjacent manifest is loaded and digest-checked, and every namespace is carried forward value-for-value. That includes namespaces from plugins this binary has never heard of. Your plugin's data is yours alone: the engine hands a reader or writer only its own namespace, matched by descriptor `id` and gated by `data_version`, and re-encodes all namespaces canonically on the way out. A DOCX writer in the same format library can decode those style ids through `ctx.preservationAs`; no runtime plugin lookup is involved.

Version your data like a wire format `data_version` is compared before your data is returned to a reader or writer. Bump it whenever the JSON shape changes. Incompatible versions remain opaque and are carried forward, but they are not handed to the plugin for decoding. Guessing at old shapes is neither necessary nor safe.

Teach it back. Without looking at the listing, write the Minutes descriptor from memory. For each field, name the failure it prevents, and say whether that failure would otherwise appear at compile time or at runtime.

Exercise 3.1. Give Minutes a second extension, `mins`, and prove detection works by converting `standup.mins` without `--from`.
Hint: One array literal changes. The bundle regenerates the rest.

Exercise 3.2. Minutes ignores empty lines, which is fine. But it also accepts a `Name:` speaker prefix containing a URL, and mangles it. Add a report: code `minutes.ambiguous-speaker`, severity note, one direction. Follow a constructor in `formats/csv/src/reader.zig` as your model, and assert the code in a test.

Exercise 3.3. Build a bundle containing only Minutes and the Markdown writer, and a `main` that converts stdin. Measure the binary against the full CLI. Where did the difference go?
Hint: All 19 readers, their parsers, and the entity table are comptime-reachable code. Compare with `-Doptimize=ReleaseSmall` to make it starker.

PART III

Every format

Nineteen readers feed one writer. We open each container the way its authors intended. We take what maps. We file a report for what does not.

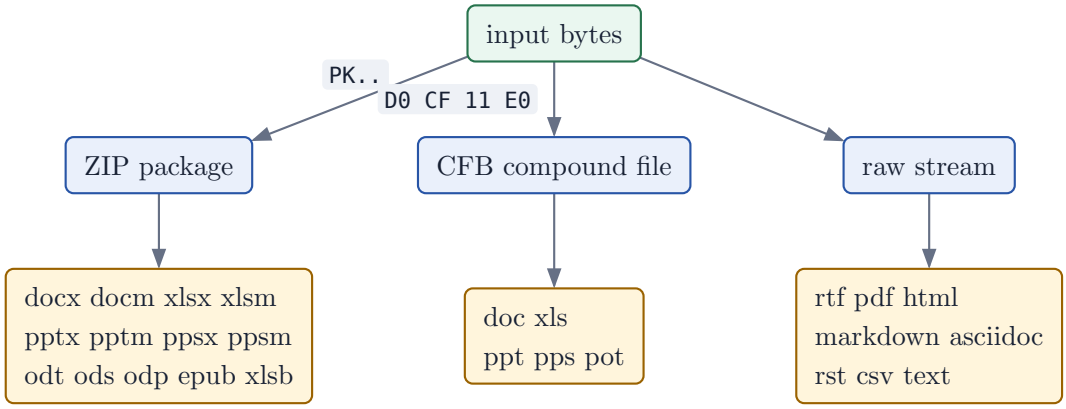

Figure 6: The container taxonomy. The format libraries sit on two shared container libraries. `zenfmt_ooxml` opens ZIP packages and `zenfmt_cfb` opens compound files. Stream formats parse their bytes directly.

4 Every Format Is a Container

Learning contract By the end of this chapter you should be able to name the three container families and place any input format in one of them. You should be able to explain why the ZIP reader trusts only the central directory, and trace a CFB stream through its FAT chain. For any office format, you should be able to state one thing zenfmt maps faithfully and one thing it reports as lost.

Checkpoint: representation Recall from Part II that a reader never returns a tree. It narrates one through the `Emitter`: `beginBlock`, `text`, `endBlock`. The validator checks the result. Every reader in this chapter is such a narration. They differ only in what they read.

4.1 Three families

Every input zenfmt accepts arrives in one of three shapes. The shape decides which support library opens it, which limits guard it, and which attacks it must survive.

The dispatch is content sniffing. It is not faith in file extensions. A ZIP container names its own subtype. The characteristic part names appear verbatim in the central directory: `word/document.xml`, `xl/workbook.bin`, `ppt/presentation.xml`. The OpenDocument and EPUB families carry a `mimetype` entry, and its value is stored uncompressed for exactly this reason. Detectors can find `application/vnd.oasis.opendocument.spreadsheet` or `application/epub+zip` without inflating anything.

A compound file names its subtype one level down. Its directory entries are UTF-16LE stream names. Finding `WordDocument`, `Workbook`, or `PowerPoint Document` in the directory sectors settles doc versus xls versus ppt. Nothing needs to be parsed first.

Content signature A byte pattern that a format's own tooling is obliged to write: magic numbers, mandatory part names, the `mimetype` entry. zenfmt routes on signatures first and treats the extension as a hint. A `.docx` that is really RTF converts as RTF.

4.2 The ZIP discipline

The ZIP reader in `zenfmt_ooxml` reads the central directory only. Local file headers are advisory copies. A hostile archive makes them disagree with the directory, so zenfmt never walks them. From the directory

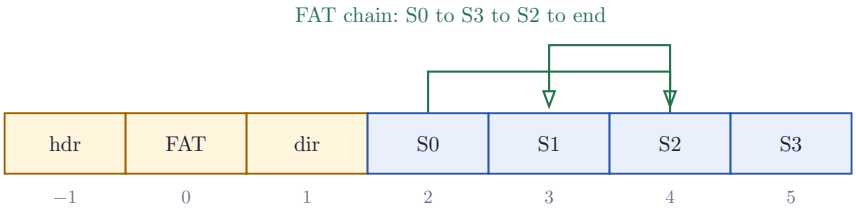

Figure 7: A CFB stream is a chain of sectors threaded through the FAT. Sentinel values terminate chains. A crafted FAT that loops back is detected, because every walk is bounded by the sector count, and the file is refused as malformed.

it accepts exactly two compression methods, stored and deflate, and it enforces the archive limits of chapter 6 while streaming, not after.

Three rules have no override flag, because no legitimate document needs one:

1. A hostile entry name rejects the whole archive before any entry is read. Hostile means absolute, drive-lettered, or containing `..`.
2. An encrypted entry is a refusal, never a skip.
3. Any compression method other than stored or deflate is a refusal.

Decompression is budgeted per entry. An entry may expand to at most `max(compressed_size, 64)` times `max_compression_ratio` bytes. The check runs as the inflate stream produces output. A zip bomb therefore dies in the middle of the stream with `docx.archive-limit`, or the sibling code of whichever format opened the archive. It costs zenfmt the budget, not the bomb's full yield.

4.3 The compound-file discipline

The legacy binary formats predate ZIP-in-Office by a decade. Their container, MS-CFB, is a small FAT filesystem inside a file. It has 512-byte sectors, a File Allocation Table mapping each sector to the next, and a directory of 128-byte entries with UTF-16LE names. Streams smaller than 4,096 bytes live in a mini stream of 64-byte mini-sectors with its own mini-FAT.

`zenfmt_cfb` gives format libraries the same API shape the ZIP reader gives: `Cfb.open(arena, bytes, limits)`, then `find(name)`, then `readStream(arena, entry, limits)`. Every chain walk is bounded by the sector count. The bounded walks cover the FAT, the DIFAT, the mini-FAT, and the directory. A cyclic FAT produces `doc.not-a-compound-file` rather than a hang. The library also owns the two text helpers every CFB tenant needs, cp1252 and UTF-16LE decoding, because the formats inside predate UTF-8.

4.4 XML as events

Between the container and the reader sits one more shared machine: the pull parser in `zenfmt_xml`. It produces events, never a tree. The events are `element_start`, `element_end`, `text`, and `done`. A hundred-megabyte `document.xml` therefore costs bounded memory. Three decisions matter more than its size:

- No DTD, ever. A `DOCTYPE` declaration is refused outright, with `docx.doctype-refused` and its siblings. Office software never writes one. Entity-expansion attacks require one. The only entities that decode are the five predefined names and numeric references.
- Namespaces match by URI. Elements match on the pair of namespace URI and local name, never on prefix. A document that renames `w:` to `x:` parses identically.
- Depth is bounded. An explicit stack capped by `max_xml_depth` makes deep nesting a limit report, not a stack overflow.

Predict before reading on A DOCX part starts with `<!DOCTYPE w:document []>` and otherwise parses cleanly. Will zenfmt convert it with a warning, or refuse? Decide which principle from chapter 6 applies, then check the transcript there.

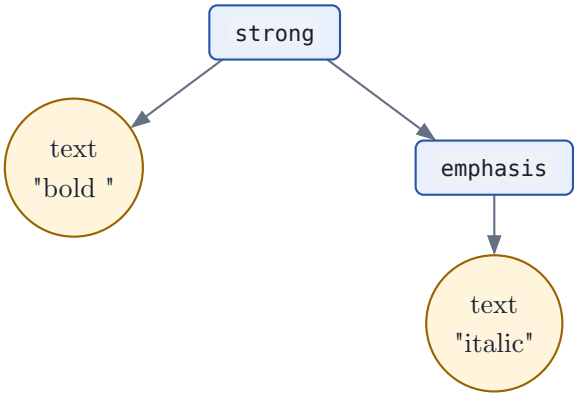

Figure 8: Two adjacent runs, bold then bold-italic. Closing everything and reopening would emit two separate bold spans. Sharing the common prefix keeps one `strong` node. This is the tree a person would draw.

4.5 DOCX: resolution before mapping

WordprocessingML separates what a paragraph is from what it references. The reader (ZDS 0003) therefore resolves before it maps.

- Styles. A paragraph names a style id. The file `word/styles.xml` chains styles through `basedOn`, bounded at 16 hops. The chain decides whether the paragraph is `Heading1` through `Heading9`. A heading arrives as `heading` with its level. Every other named style is preserved as plugin data in the manifest, so a future DOCX writer can restore it.
- Numbering. A list item names a pair, `numId` and `ilvl`. The file `word/numbering.xml` maps concrete to abstract numbering, which decides bullet versus ordered. Word stores no list structure at all, so zenfmt synthesizes it with a stack keyed on level. A level jump opens intervening empty items. A level decrease closes lists. Each table cell gets its own list base, so a table never leaks list state.
- Fields. `HYPERLINK` lives in a field machine. `fldChar begin` starts instruction capture. `separate` parses the URL and opens a link. `end` closes it. Readers keep the cached field result. The instruction text never leaks into output.

Character formatting is a set of flags on runs: bold, italic, strike, superscript, subscript, small caps, underline. Markdown wants nesting instead. Every flag-based reader (DOCX, RTF, ODT) converts flags to the canonical nesting order and shares the common prefix between consecutive runs.

> ■ **Canonical inline order** `link`, `strong`, `emphasis`, `strikethrough`, `superscript`, `subscript`, `small_caps`, `underline`, outermost first. The list lives in `core/src/payload.zig`. Flag sets always nest in this order, so equal formatting produces byte-equal Markdown from every source format.

Tables read their column count from `tblGrid` before the first row. Leading `tblHeader` rows route into `table_head`. A `gridSpan` becomes the cell's `col_span`. A `vMerge` continuation cell folds into its originating cell. These degradations file one `docx.merged-cells-degraded` note per table, not one per cell.

4.6 Spreadsheets: one projection, three encodings

All four spreadsheet formats project identically. Each sheet becomes a level-2 heading carrying the sheet name, then one GFM table whose first row is the header. The reason is simple: a user comparing `.xlsx` output with `.xls` output should see the same document. What differs is only the cell encoding.

Dates deserve a sentence. A spreadsheet date is a float counting days from an epoch. zenfmt renders it as ISO `YYYY-MM-DD` using the civil-from-days algorithm, never a locale. A formula converts as its cached value. When the cache is absent, the reader files one `formula-without-cached-value` note and moves on. zenfmt evaluates nothing.

Format	Encoding	The characteristic trap
xlsx / xlsm	XML: `<c r="B2" t="s"><v>7</v></c>` with shared strings. Styles map `numFmtId` to date or percent.	Self-closing `<row/>` and `<c/>` elements produce no end event. Both must still materialize, or the table loses its shape.
xlsb	Binary records in a ZIP: a record id of 1 or 2 bytes, a length varint of 1 to 4 bytes, then records such as `BrtCellRk`, `BrtCellIsst`, and cached `BrtFmlaNum`.	`BrtBundleSh` in real files carries a 40-byte preamble where the spec example shows 36. The reader accepts both by requiring exact record consumption.
xls	BIFF8 records in a CFB stream: `BOUNDSHEET`, `SST` with `CONTINUE` splits, `RK` packed numbers, `FORMULA` plus a cached `STRING`.	An `SST` string may switch encodings at a `CONTINUE` boundary. The 1904 `DATEMODE` flag shifts every serial date.

Table 1: The same cell, three ways. The projection is shared. Only the decoding differs, which is why the three readers agree byte-for-byte.

4.7 Presentations: the honest projection

A slide deck is a spatial, animated medium. A Markdown file is neither. Every presentation reader emits an unconditional warning before its first slide: `pptx.presentation-projection`, `ppt.presentation-projection`, or `odp.presentation-projection`. The warning states that positioning, animation, and non-text shapes do not survive. This is the report system used exactly as designed. The loss is structural and guaranteed, so the diagnostic is unconditional. `--strict` turns it into a refusal for pipelines that must not accept it.

What does survive: slide order from `sldIdLst` resolved through relationships, title placeholders as level-2 headings, body text with bullet levels rebuilt as nested lists, `a:tbl` tables, external hyperlinks resolved through each slide's own relationship part, images with their alt text, and speaker notes as a `container` classed `notes`. The binary `.ppt` walks its record tree instead. Every record is a header of version, instance, type, and length. The reader harvests `TextHeaderAtom` classifications and the `TextCharsAtom` or `TextBytesAtom` payloads under them. ODP walks `draw:page` frames classed `title`, `outline`, and `notes`, and reads slide tables from `table:table` elements inside frames.

4.8 OpenDocument text: styles and a re-parse trick

ODT keeps heading levels in the open: `text:h` carries `text:outline-level`. Lists are explicit. Those two facts remove DOCX's two hardest inferences. The difficulty that remains is style resolution. A `text:span` names an automatic style. That style may chain by `style:parent-style-name` into `styles.xml`. Only at the end of the chain does the reader learn that `fo:font-weight` is bold.

One implementation detail is worth retelling. Footnote bodies appear inline, inside the paragraph that references them. The AST wants note bodies deferred to the end. So the reader captures the note body's raw byte range during the main pass and re-parses it afterwards. The captured fragment floated free of its namespace declarations, so the reader wraps it in a synthetic `<office:wrap>` element that re-binds the `office:`, `text:`, `table:`, `draw:`, and `svg:` prefixes. The pull parser neither knows nor cares that the document is synthetic.

4.9 EPUB: a website in a box

EPUB is ZIP holding XHTML. `META-INF/container.xml` names the OPF package. The package's `manifest` maps ids to files. Its `spine` fixes reading order. Dublin Core metadata (`dc:title`, `dc:creator`, `dc:language`) becomes document metadata in the manifest. Each spine chapter then flows through the same HTML machinery the standalone HTML reader uses: `parseFragment` with the chapter's directory

as a base, so relative image and link targets rebase onto archive-entry paths. Chapters concatenate without synthetic headings. The book's own headings are trusted. A `META-INF/encryption.xml` file means `epub.drm-refused`, with no override.

4.10 PDF: geometry, not structure

PDF is the one input with no document structure at all. It is placed glyphs. The reader (ZDS 0011) parses the object graph: classic xref tables and cross-reference streams, object streams, FlateDecode with PNG predictors. It decodes text through `ToUnicode` CMaps or the standard encodings. It tracks the text matrix and a glyph-width pen, so words join and split where the geometry says they do.

Structure is then recovered, conservatively. A font-size histogram per document sets heading tiers: a line at 1.6 times the body median or more becomes `h1`, at 1.35 times `h2`, at 1.12 times with bold `h3`. Vertical gaps split paragraphs. Hyphenated line-wraps rejoin. Tables are claimed only when the geometry insists. That means a drawn lattice of rules, or start-x positions clustering into the same columns across at least three consecutive baselines. Paragraphs win every doubt. Embedded images are saved as-is where the encoding permits: JPEG and JPEG 2000 streams verbatim, Flate bitmaps wrapped losslessly as PNG, through the media pipeline of chapter 7. Where it does not permit (CCITT, JBIG2), they are counted in `pdf.images-omitted`. The unconditional `pdf.layout-projection` warning tells the user exactly what kind of recovery they are reading.

4.11 Legacy Word: the piece table

The `.doc` WordDocument stream does not store text in order. The File Information Block points into a table stream, named `0Table` or `1Table` (a FIB flag says which). The table stream holds the piece table: a sorted array of character positions (cp) paired with descriptors that map each cp range to a file offset (fc) range. Bit 30 of the fc marks 8-bit cp1252 pieces; without it, the piece is 16-bit UTF-16LE.

Paragraph marks arrive as `\r` characters in the decoded stream. Heading styles resolve through the stylesheet (STSH) by built-in style identifier. The paragraph's style index is found through the PAPX formatted-disk-page chain. Hyperlink fields hide between the field characters `0x13`, `0x14`, and `0x15`. The reader keeps the cached result and parses the instruction for its URL. Cell marks (`0x07`) flatten into paragraphs with a `doc.tables-flattened` note. That is the honest reading of a structure this reader does not yet rebuild.

4.12 What each format loses, and what the facets carry

Every reader's ZDS record carries the full table. This is the shape of it. The point is not the losses themselves; every converter loses these. The point is that each loss has a name you can grep for in the manifest, and that since IR v2 a third column exists: information Markdown cannot hold that now rides in typed facets, carried for a richer writer instead of merely mourned.

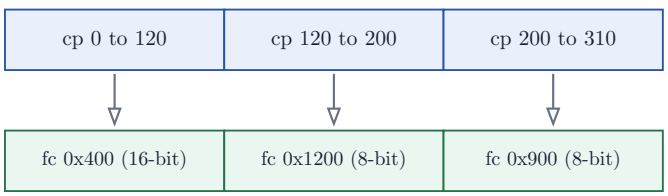

Figure 9: The piece table maps logical character positions to physical file ranges. Reading order is cp order, not file order. An edit in 1997 Word appended a piece, not the text.

Format	Converts faithfully	Deliberate, reported losses	Carried in facets
docx	headings, styled runs, links, lists, tables, footnotes, images	comments (`docx.comment-dropped`), text boxes, merged cells degrade (`docx.merged-cells-degraded`)	style names, tracked insertions and deletions, page size, image bytes as resources
rtf	styled text, tables, lists, links, footnotes, headings	image bytes (`rtf.images-dropped`), OLE objects (`rtf.objects-dropped`), nested tables flatten	stylesheet names on styled paragraphs
xlsx/xlsb/xls/ods	typed cells, dates, cached formula values, sheet structure	charts, cell formatting	grid facets: sheet, exact row and column, value type, formula source where the format spells it, cached value, merges
pptx/ppt/odp	titles, body text, tables, links, images, notes	animation, charts (`presentation-projection`)	slide geometry in EMU (pptx, odp), per-slide provenance (ppt), picture bytes as resources (pptx)
odt	headings, styles, lists, tables, images, footnotes	sourceless frames (`odt.frame-dropped`)	common style names, annotations as comment revisions, image bytes as resources
epub	chapters in spine order, metadata, images, links	CSS entirely, fonts; DRM'd books refuse (`epub.drm-refused`)	per-chapter provenance naming the spine member
pdf	text, headings by size, paragraphs, geometric tables, embedded images	layout itself (`pdf.layout-projection`), unmappable glyphs (`pdf.unmappable-text`), CCITT and JBIG2 images	page positions in EMU with projected confidence, per-page provenance
doc	full text in both encodings, heading styles, hyperlinks	table grids flatten (`doc.tables-flattened`), embedded objects	STSH style names, heading provenance with stream positions

Carried is not converted. The Markdown artifact is the same either way; the facets ride in the manifest as counted, digested summaries, and in full under `--preserve-facets`. What changed is that the loss ledger now distinguishes three fates instead of two: rendered, reported as lost, or carried for a writer that can use it.

Teach it back. Explain to a colleague why the ZIP reader ignores local file headers, why the DOCX reader needs a numbering stack when Word stores no list structure, and why all three spreadsheet

readers emit byte-identical Markdown for equal content. If the last one stalls, re-read the canonical-order definition.

Exercise 4.1. Take any `.docx` you own and run `zenfmt file.docx --reports=json`. Match every report code you receive to a row of the omissions table in ZDS 0003. Which losses did your document actually exercise?

Hint: Most documents trip two or three codes, not the whole table.

Exercise 4.2. Rename a real `.xlsx` to `evil.docx` and convert it. Explain the output using the content-signature rule. Then find the sniffing order in `core/src/root.zig` that made it work.

5 The One Writer

Learning contract By the end of this chapter you should be able to state the writer's determinism rules from memory and predict which characters get escaped at a given position. You should be able to explain how a trailing space escapes a bold span without breaking CommonMark, read the writer's capability declaration and price a document's losses from the rule table, choose the right `--strict` grade for a job, and prove, with two commands, that Markdown round-trips to a fixed point.

zenfmt ships nineteen readers and exactly one writer. The asymmetry is deliberate. An intermediate representation is easiest to judge when many producers feed a single consumer. Every reader bug, every mapping decision, every canonicalization rule surfaces as a diff in the same Markdown. The day a second writer lands, it inherits an AST that nineteen formats have already argued into shape.

Checkpoint: the contract The writer receives a validated tree. The invariants of chapter 2 hold before `write` is called. The writer never defends against an impossible tree. It renders the possible ones deterministically.

5.1 Determinism first

Two conversions of the same input must be byte-identical, on every machine, forever. The manifest digests of chapter 7 depend on it. The rules are few and absolute:

- Line endings are `\n`, only.
- Exactly one blank line between blocks. Exactly one `\n` at end of file.
- No trailing whitespace on any line.
- Equal formatting renders equally. The canonical inline order of chapter 4 means the writer never chooses between `**_x_**` and `_**x**_`. The tree already decided.

Determinism The output is a pure function of the document tree. No timestamps, no locale, no hash-map iteration order, no "usually the same". This is what makes output diffable and the round-trip test meaningful.

5.2 Escaping by position

A naive writer escapes every Markdown metacharacter and produces backslash soup. This writer asks where the character sits, because most metacharacters are only meta somewhere.

The `&` rule is the one that pays for itself. Escaping every ampersand turns ordinary prose ugly. So the writer scans forward for the shape of a character reference and escapes only genuine look-alikes.

5.3 The edge-space problem

Word processors love styled runs with edge spaces. A DOCX or RTF run often holds bold text with a trailing space inside the bold. Rendered naively, that is `**bold **`. CommonMark rejects it: a closing `**` cannot follow whitespace. The whole span would silently stop being bold.

The writer relocates edge spaces outside the delimiters as it closes each styled span. It also drops span pairs that end up empty. Here is a real transcript. The RTF run has a trailing space inside `\b`:

Character	Escaped when
`# > - + =`	it is the first non-space character of a line
digits	a digit run at line start would open an ordered list; the writer looks ahead for the `.` or `)` before deciding
`* _`	a delimiter run here could open or close emphasis
`` ` ``	always; code spans are too cheap to trigger
`[ ]`	always; brackets are link syntax anywhere
`&`	only when the following text looks like an entity, such as `&name;` or `
`. Plain `AT&T` passes clean.
`\ < |`	backslash always; `<` where HTML could start; `|` inside table cells

Table 2: The position-aware escaping rules. A character is escaped only where CommonMark would actually reinterpret it.

```
$ printf '{\\rtf1\\ansi Plain {\\b bold }after.\\par}' > edge.rtf
$ zenfmt edge.rtf --stdout --quiet
Plain **bold** after.
```

The space moved from inside the bold to between the words. The output is valid CommonMark and reads identically.

Predict before reading on A DOCX run is bold and contains only spaces. What should the writer emit? Decide, then check: the empty-pair rule drops the delimiters and keeps the spaces. The sequence `** **` appears in no zenfmt output.

5.4 Lists: tight, loose, and the prefix machine

A Markdown list is tight when no blank lines separate its items. The distinction changes rendering in every downstream tool: compact items versus a paragraph per item. The writer computes tightness from the tree and holds it per list frame. A nested list inside a tight list therefore does not force blank lines into its parent.

Continuation lines take a prefix as wide as their marker. This is real output. Note the three-space continuation under the ordered marker, and the absence of blank lines in the tight structure:

```
1. one
   - deep bullet
2. two
```

The mechanics are one machine. Every open container contributes to a prefix stack. Markers are pending: written once, on the first line of their item. Blank lines between blocks print the prefix with no marker. Getting `2.` to line up under `1.` is exactly this machine and nothing else.

5.5 Tables, aligned

GFM tables render with padded columns:

```
| ID  | Months   |
| --- | -------- |
| 1   | January  |
| 2   | February |
```

Padding means measuring every cell before writing the first row. That extra pass has a price, and the benchmark chapter states it honestly. The 25,000-row CSV corpus file is zenfmt's slowest conversion. It is also the one file where anydoc, which emits ragged unpadded pipes, is faster. The scaling is linear: twice the rows costs twice the time. The padded table is a deliberate quality purchase. It is diffable, readable in a terminal, and stable under regeneration.

Cell content that Markdown cannot hold degrades with a name. Block content flattens (`markdown.table-cell-flattened`). Row and column spans degrade (`markdown.cell-span-degraded`). A nested table is dropped with `markdown.nested-table-dropped`.

5.6 Code and raw content

Code blocks fence with backticks and carry their language string unchanged, so syntax highlighting survives the trip. A code block whose content contains a backtick run gets a longer fence. Inline code picks its span delimiters the same way, and pads with spaces when the content starts or ends with a backtick. This is the CommonMark dance, done once, in the writer, so no reader thinks about it.

Raw content is stricter. The AST can carry raw fragments tagged with their format, an HTML island from the HTML reader, for example. The writer emits raw content only when its tag says `markdown` or `html`, the two formats a Markdown file can host. Every other raw fragment is dropped with `markdown.raw-dropped`, because emitting LaTeX into a Markdown file does not become correct by hoping.

5.7 Footnotes, deferred

Note references render as `[^1]` in reading order. Bodies collect and render at the end of the document as `[^1]:` definitions. The AST made this cheap. Readers declare notes and supply bodies whenever their format happens to store them: inline for ODT and RTF, in a separate part for DOCX. The tree always carries them deferred. What Markdown cannot express at all, `underline` and `small_caps`, renders as its plain content with one aggregated `markdown.style-dropped` note.

5.8 What the writer declares

Until ZDS 0013 the writer's losses were correct but implicit: the underline rule lived in one function, the nested-table rule in another, and only the code said which constructs degrade. The writer now states its position once, at compile time, in `formats/markdown/src/capabilities.zig`:

```
pub const capabilities: lowering.Capabilities = .{
    .exact_blocks = &.{ .plain, .paragraph, .heading, .quote, ... },
    .lowered_blocks = &.{ .raw_block, .table, .definition_list, ... },
    .exact_inlines = &.{ .text, .emphasis, .strong, .link, ... },
    .lowered_inlines = &.{ .underline, .small_caps, .raw, ... },
    .rules = &rules,   // the priced degradations below
};
```

The declaration is total, and totality is checked where lies are cheapest to catch. Every kernel tag must appear in exactly one of exact, lowered, or refused, or the bundle does not compile. When a new tag joins the schema table of Chapter 2, this file refuses to build until the writer decides what it will do about it. The mapping-table obligation that ZDS 0001 places on readers now has its mirror on the writer side.

5.9 Pricing the losses

Each degradation is a rule with a stable name, a diagnostic, and a cost on the loss vector of ZDS 0013: component one counts dropped content, component two structural degradation, component three style and metadata loss. The ten rules:

Rule	Costs	Report code
`raw-dropped`	content	`markdown.raw-dropped`
`nested-table`	content	`markdown.nested-table-dropped`
`cell-flattened`	structure	`markdown.table-cell-flattened`
`cell-span`	structure	`markdown.cell-span-degraded`
`definition-list`	structure	`markdown.definition-list-degraded`
`style-dropped`	style	`markdown.style-dropped`
`extension-fallback`	style	`markdown.extension-fallback`
`citation-dropped`	style	`markdown.citation-dropped`
`number-style`	style	`markdown.list-number-style-degraded`
`container-attrs`	style	`markdown.container-attributes-dropped`

The rendering code did not move. Where the writer used to report a loss directly, it now records a hit against the rule, and the engine's plan accumulator prices the hits and flushes one aggregated report per fired rule, in first-hit order, with the count. The refactor was verified byte-for-byte: same artifacts, same reports, same manifests, across the whole benchmark corpus. What changed is that the losses now have prices the engine can act on before writing anything.

> ■ **Loss taxonomy** ZDS 0013 names five outcomes for any construct: equality, normalization (equal meaning, canonical spelling), degradation (some meaning lost, priced and reported), omission (a subtree dropped), and refusal (no output at all). Every rule above is a priced degradation or omission. Nothing is lost silently, and nothing lost is called equal.

5.10 Strict, in three grades

The prices exist so refusal can be a policy instead of a feeling. `--strict` gates the conversion on the aggregated cost, in three grades:

Flag	Refuses when
`--strict` or `--strict=content`	Any semantic content would drop: raw fragments, nested tables, or a reader-side dropped construct.
`--strict=structure`	Content loss, or structural degradation: flattened cells, degraded spans, definition lists.
`--strict=exact`	Any degradation at all. The plan must be pure exact emission.

The gate runs before emission. The engine performs a dry run into a discarding writer, prices the plan it would select, adds the losses the reader already reported, and tests the grade. A refused conversion emits `core.strict-refused` and commits nothing: no artifact, no manifest, not a byte on a stream. A writer may also refuse a construct in every mode by listing it in its capabilities; the engine then fails the conversion with `core.construct-refused` before planning at all. Markdown refuses nothing, degrades much, and prices everything, which is the honest posture for a plain-text target.

5.11 The reader: the writer's sparring partner

The Markdown reader exists first of all to close the loop. It was the phase-2 proof that a second reader needs no engine changes. It is also the instrument that makes round-trip testing possible. It is a full CommonMark plus GFM implementation in two passes. No recursion. No regex.

Block pass. A line-at-a-time machine matches each line against the open container stack: quotes, lists, footnote definitions. Then it opens what the remainder starts: headings, fences, tables, paragraphs. Two rules do the heavy lifting.

- Lazy continuation. A paragraph line may omit its containers' markers, unless the line would instead start a new construct. The interruption rule has one subtlety worth naming. The line `2. four` after `1. three` continues the open list. It does not lazily join the paragraph, because a matching open list claims its own continuation.
- Structural closes. Eight block starts close an open list rather than nest inside its last item: quotes, fences, headings, thematic breaks, tables, HTML, and reference definitions. Lists end where a human says they end.

Inline pass. Emphasis is the delimiter-stack algorithm from the CommonMark spec. Text becomes a doubly-linked list of nodes. Delimiter runs of `*`, `_`, or `~~` join a chain marked can-open and can-close. The processor repeatedly matches the closest closer to its nearest opener.

> **The rule of three** Suppose a delimiter run can both open and close. It may match a run only if the combined length of the two runs is not a multiple of three, or both runs are themselves multiples of three. This single rule is why `a**"foo"**` bolds and `**foo*` does not. It is also why implementing emphasis by counting asterisks always fails.

Code spans resolve before emphasis and hide their contents from it. Brackets resolve after, against inline targets or the reference definitions the block pass collected. Labels are case-folded and whitespace-normalized before lookup.

5.12 The fixed point

Everything above compounds into one testable property. Converting Markdown to Markdown reaches a fixed point after at most one pass. The first pass may canonicalize: setext headings become ATX, `*em*` becomes `_em_`, spacing normalizes. The second pass must be byte-identical to the first, forever after.

```
$ zenfmt rt.md --stdout --quiet --from markdown > rt1.md
$ zenfmt rt1.md --stdout --quiet --from markdown > rt2.md
$ cmp rt1.md rt2.md && echo FIXED-POINT
FIXED-POINT
```

The round-trip suite pins this for headings, emphasis, lists, tables, code, links, and footnotes. The fuzzer feeds the reader garbage with the validator as oracle. When a future change breaks either canonicalization or parsing, this is the test that names it.

Teach it back. Explain why `**bold **text` is not valid CommonMark emphasis, what the writer does about it, and why the fix belongs in the writer rather than in every reader that produces edge spaces.

Exercise 5.1. Predict the output of converting this Markdown to Markdown, then run it: `Heading\n=======\n\nA *b* c** d.` Which characters changed, and which escaping rule fired on `**`?

Hint: Setext becomes ATX. An unmatched delimiter run is plain text, but only until something could match it.

Exercise 5.2. Find `appendClose` in `formats/markdown/src/writer.zig` and list the transformations it applies when closing a styled span. Match each to a sentence in this chapter.

6 Hostile by Default

Learning contract By the end of this chapter you should be able to state the threat model in one sentence and name the limit that stops each classic attack: the zip bomb, the billion laughs, the cyclic FAT, the reference loop. You should be able to compute a decompression budget, explain what `--strict` changes, and say why a failed conversion leaves no file behind.

The threat model is one sentence: every input document is untrusted bytes. A converter is a parser farm, and parser farms are where memory bugs, hangs, and resource exhaustion live. zenfmt's answer is not cleverness. It is arithmetic: every allocation bounded, every loop bounded, every expansion budgeted, and every refusal explained to the person holding the file with the exact bound and a safe next action.

The refusal philosophy When input crosses a limit, zenfmt refuses loudly, names the limit, and says what to do next. It never silently degrades security, never retries with the check disabled, and never publishes a partial path output. A direct API stream reports whether no bytes, a prefix, or the complete artifact reached its caller.

Checkpoint: reports Recall from chapter 1 that every diagnostic answers four questions: what happened, where, what zenfmt did about it, and what you can do. That structure is only the floor: the direction must identify a concrete correction, safe override, or containment action. Every refusal in this chapter is such a report, with a stable code you can match in scripts.

6.1 The limits table

Every named bound lives in one struct, `core/src/limits.zig`. Nothing else in the tree hard-codes a resource number. Each field is overridable from the command line, with one exception explained below.

The exception: `max_depth` and `max_xml_depth` may be raised only to 4,096, the hard cap that sizes every fixed walker stack in the binary. An override above the cap is refused as an invalid value. There is no flag that turns a bounded stack into an unbounded one.

The IR v2 bounds near the bottom come from ZDS 0013. `max_nodes` and `max_decoded_text_bytes` bound the kernel itself, so a small compressed input cannot decode into an unbounded tree or text pool. A facet bomb, one paragraph carrying a million annotations, dies at `max_facet_rows` as a refusal rather than an allocation storm; the erasure axiom makes this safe, because no facet can change what renders. The two lowering limits cap the writer's planning work under hostile rule interactions.

Memory has one more bound worth knowing. Since the InputMode repair, a ZIP-backed format read from a file is never held in memory whole: the reader windows the central directory and each entry separately, so peak memory is the directory plus one expanding entry, not the archive beside its expansion. Piped input for those formats spills to a temporary file first for the same reason.

6.2 Bombs are a budget problem

A zip bomb is a small file that expands enormously. The defense is not detecting bombs. It is refusing to pay for them. Each archive entry gets a budget:

$$\text{allowance} = \max(\text{compressed size}, 64) \times \text{max\_compression\_ratio}$$

With the default ratio of 200, a 1 KiB entry may expand to 200 KiB, and a 486 KiB entry to about 95 MiB. The check runs inside the inflate loop, as output is produced. The moment the stream exceeds its allowance, the conversion stops with an archive-limit report and the remaining gigabytes are never

Limit	Default	What it bounds
`max_input_bytes`	512 MiB	bytes read from one input document
`max_depth`	256	nesting of either node tree, and the size of every explicit walker stack
`max_archive_entries`	4,096	entries admitted from one archive central directory
`max_entry_uncompressed`	256 MiB	expansion of one archive entry
`max_total_uncompressed`	1 GiB	expansion of all read entries together
`max_compression_ratio`	200	expansion ratio, checked during streaming decompression
`max_entry_name_bytes`	1,024	length of one archive entry name
`max_xml_depth`	256	XML element nesting
`max_scan_chunk_bytes`	1 MiB	scanner scratch per chunk
`max_manifest_bytes`	16 MiB	size of an adjacent manifest accepted on input
`max_plugin_data_bytes`	4 MiB	one plugin-data namespace value
`max_manifest_depth`	64	JSON nesting in an accepted manifest
`max_report_samples`	4	locations one aggregated report lists before counting the rest
`max_reports_total`	16 Ki	distinct aggregated report groups
`max_resources`	256	resources a reader may extract
`max_resource_bytes`	128 MiB	total extracted resource bytes
`max_nodes`	16 Mi	kernel nodes per document
`max_facet_rows`	1 Mi	facet rows across all tables
`max_decoded_text_bytes`	256 MiB	decoded text pool bytes
`max_lowering_alternatives`	8	lowering alternatives per construct
`max_lowering_work`	64 Mi	rule applications per conversion

Table 3: Every limit, its default, and what it bounds. Overrides use `--limit NAME=VALUE`.

decompressed. The per-entry and total caps back the ratio up, so many medium entries cannot add up to a surprise either.

The same budget discipline guards PDF streams: FlateDecode output is capped by the same ratio and size rules before predictors run.

6.3 The attacks with names

Billion laughs. The XML entity-expansion attack needs a DTD, and a DTD needs a `DOCTYPE`. zenfmt refuses the `DOCTYPE` itself. This is the real banner, produced by a crafted `.docx`:

```
-- XML WITH A DOCTYPE REFUSED --------------------------------- zenfmt

A part in this document carries a DOCTYPE declaration. No office
application writes one, and DTD processing is how XML entity-expansion
attacks work, so zenfmt never processes them.

The conversion stopped and no output file was created.
```

```
What you can do:

    Treat the file as suspect; a legitimate document re-exported from
    its native application will not carry a DOCTYPE.
```

Note the last line of the middle paragraph. No output file was created. We verify that claim below.

Cyclic chains. A CFB file whose FAT points sector 3 at sector 5 and sector 5 back at sector 3 would hang a naive reader forever. Every chain walk in `zenfmt_cfb` is bounded by the sector count, so the cycle is detected within one pass and refused as malformed.

Reference loops. A PDF object that resolves to itself, directly or through friends, would recurse a naive resolver to death. The resolver bounds indirection at 32 hops and carries an in-progress set, so a loop is `pdf.malformed`, not a hang. The xref chain is bounded at 64 sections; the page tree walk carries a visited set and a page cap.

Traversal names. An archive entry named `../../etc/passwd` rejects the entire archive before any entry is read. There is no per-entry forgiveness, because a name like that means the archive was built by an attacker, and nothing else in it deserves trust.

6.4 Encryption is always a refusal

zenfmt does not decrypt documents, not even with well-known default passwords. An encrypted input is refused with a format-specific code and no override flag:

Code	Trigger
`docx.encrypted`	an OOXML package with encryption
`doc.encryption-refused`	the FIB encryption flag in a `.doc`
`xls.encryption-refused`	a `FILEPASS` record in the workbook
`ppt.encryption-refused`	a crypt-session container in the deck
`pdf.encryption-refused`	an `/Encrypt` dictionary in the trailer, including the empty-password case
`epub.drm-refused`	a `META-INF/encryption.xml` entry

The include directives of the lightweight formats get the same treatment. AsciiDoc `include::` and reStructuredText `.. include::` would let a document read arbitrary files from the machine converting it. Both are refused, with `asciidoc.include-refused` and `rst.include-refused`.

> ■ **Predict before reading on** You run a batch of ten thousand documents overnight and three refuse with `docx.encrypted`. Is that a bug in the batch, a bug in zenfmt, or exactly the designed behavior? What exit class do those three return, and how does your script tell them apart from crashes?

The answer to the last part: refusals carry an exit class. The CLI exits 0 on success, and maps conversion failures, usage errors, and limit refusals to 1, 2, and 3. A bulk script can count zip bombs without reading a single banner.

6.5 Strict mode

`--strict` tests the selected lowering plan before the writer opens. A strict failure therefore sends no stream bytes and leaves no artifact, manifest, or media file. Pipelines that must not accept lossy conversions get a hard gate; interactive users keep the default, which converts and explains.

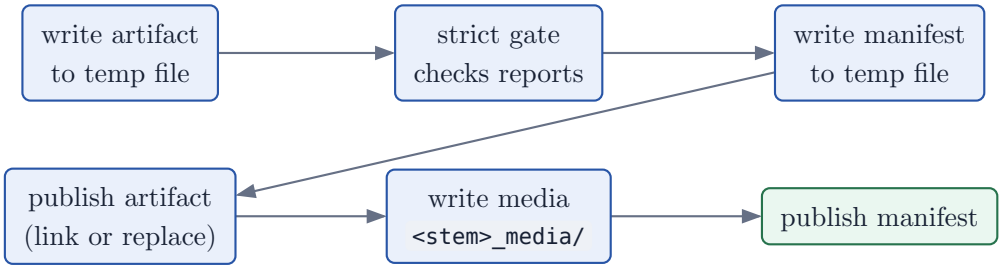

Figure 10: The commit order. Every write goes to an unpredictable temporary name first. The artifact becomes visible before its manifest, so a manifest never describes a file that does not exist.

6.6 Running out of memory is not a crash

`convert` never returns an error union. Its result always carries reports. One failure mode deserves special care: what if the failure is that there is no memory left to build a report? zenfmt reserves a static report, `core.out-of-memory`, compiled into the binary, requiring zero allocation to return. The allocation-failure suite proves it: the test harness injects failure at every single allocation site in a conversion, one site at a time, and asserts that every run returns the reserved report and never a crash or a partial artifact.

The fuzzers complete the picture. Every text reader has a fuzz target that feeds it arbitrary bytes, with the tree validator of chapter 2 as the oracle. A reader may refuse garbage. It may convert garbage into a strange but valid document. It may not produce an invalid tree, and it may not hang, because every loop it runs is bounded by the limits table above.

6.7 Nothing partial is published to a path

The last defense is the commit protocol. A conversion that fails before publication leaves destination files as it found them, and a completion manifest is never observable before its whole artifact ensemble. Direct streams cannot be rolled back; their API result reports `untouched`, `partial`, or `complete` so callers can discard an incomplete prefix.

The mechanics are ordinary and boring, which is the point. Temporary files are created next to their targets with unpredictable names. The artifact is published by link, which fails if the destination exists and `--overwrite` was not given (`core.destination-exists`), or by replace when it was. Media files follow, under the artifact's own `<stem>_media` directory. The manifest publishes last. A crash at any point cannot leave a manifest claiming an incomplete ensemble is complete. A crash after artifact publication may leave new artifact or media files without a manifest—several filesystem renames cannot form one portable atomic transaction. Such an ensemble is deliberately uncertified; rerun with `--overwrite` to complete it. Temporary files are ignored by the next run.

We promised the DOCTYPE banner told the truth. Here is the check:

```
$ zenfmt doctype.docx -o out.md; echo "exit=$?"
exit=3
$ ls out.md
ls: out.md: No such file or directory
```

Refused, explained, exit class 3, and no partial `out.md` anywhere.

Teach it back. Explain the decompression budget to someone who has never seen a zip bomb: what number is fixed, what number the attacker controls, and why checking during the stream rather than after changes everything. Then explain why `--limit max_depth=100000` is refused when every other limit can be raised.

Exercise 6.1. Compute the maximum bytes zenfmt will decompress for an archive with three entries of compressed sizes 100 bytes, 10 KiB, and 400 KiB, under the default limits. Which limit binds first

for each entry?
Hint: Remember the floor of 64 bytes, and check the per-entry cap against the ratio result.

Exercise 6.2. Build the DOCTYPE refusal yourself: any stored ZIP with a `word/document.xml` whose first bytes declare a DOCTYPE will do. Run it with `--reports=json` and find the stable code your script would match on.

PART IV

Engineering

The converter becomes a library you embed and a pipeline you extend. The contract is one function that never fails out, one arena you own, and one filter system checked by the compiler.

7 Embedding and Filtering

Learning contract By the end of this chapter you should be able to call `zenfmt.convert` from Zig or Python and dispose of its result correctly, choose the authority and isolation boundary for untrusted input, assemble a smaller bundle than the CLI ships, write a filter in your own project and compile it into a working binary, predict which parts of a document a sparse edit copies, and read another tool's `.zenfmt.json` manifest with confidence.

Checkpoint: prior knowledge You need the representation from Part II: flat preorder storage, typed indices, side tables. Nothing else. If `subtree_len` does not immediately mean "where my next sibling starts", reread that chapter first. Every promise this chapter makes about filter cost rests on it.

7.1 One call, one result

The whole library is reachable from one function:

```
const zenfmt = @import("zenfmt");

var conversion = zenfmt.convert(gpa, io, .{
    .input = .{ .path = "report.docx" },
    .output = .{ .path = "report.md" },
});
defer conversion.deinit(gpa);
```

Look at what `convert` returns. It is not `!Conversion`. It is `Conversion`, unconditionally:

```
pub const Conversion = struct {
    status: Status,               // .success or .failed
    reports: []const Report,      // every diagnostic, structured
    manifest_json: ?[]const u8,   // canonical JSON; null on failure
    exit_class: report.ExitClass, // .conversion, .usage, or .limit
    stream: StreamState,          // what a .writer output received
    arena_state: std.heap.ArenaAllocator.State,
};
```

`stream` answers a question only streamed output can raise: did the caller's writer receive nothing, a partial prefix, or the complete artifact? A failed streamed conversion is thereby distinguishable from a successful conversion of an empty document. Path output reports `.none`; its transactional commit makes the question moot.

The never-fails-out contract `convert` returns a value in every case: success, malformed input, limit refusal, even allocation failure. An expected failure is data. You get `.status == .failed` plus structured reports. The reason is simple: the moment a conversion fails is exactly the moment an embedding application needs the explanation, and a Zig error code would throw the explanation away.

One failure cannot allocate a report: running out of memory while building reports. For that case the library keeps one reserved, statically allocated report, with code `core.out-of-memory`. It is returned without touching the allocator. Your error path can rely on `conversion.reports` being non-empty and renderable even then.

7.1.1 The arena you are handed

Everything inside the result lives in one arena: the report strings, the manifest bytes, the document that briefly existed. `convert` created the arena, and its state travels inside the result. `deinit` promotes it back and frees everything at once:

```
pub fn deinit(c: *Conversion, gpa: std.mem.Allocator) void {
    var arena = c.arena_state.promote(gpa);
    arena.deinit();
    c.* = undefined;
}
```

Two consequences are worth internalizing. First, the slices in `reports` and `manifest_json` are valid until `deinit` and not one instruction longer. Copy them out if they must outlive the result. Second, one conversion is one arena. There is no per-node allocation to leak and no destructor order to get wrong. The whole memory story of a conversion is this pair of calls.

> ■ **API anchor: `zenfmt.convert(gpa, io, options) Conversion`** One conversion, start to finish: resolve the input, detect the format, read, validate, filter, write, commit. Never an error union. The result owns its arena. in `core/src/root.zig`

7.1.2 Rendering what happened

`Conversion.renderReports` writes the same Elm-style text the CLI prints. An embedding application does not reimplement the diagnostic format:

```
if (conversion.status == .failed) {
    var buffer: [4096]u8 = undefined;
    var stderr = std.Io.File.stderr().writerStreaming(io, &buffer);
    try conversion.renderReports(&stderr.interface, .{});
    try stderr.interface.flush();
}
```

For machines rather than people, iterate `conversion.reports` directly. Each report carries a stable `code`, a severity, a loss tier, and concrete directions. `exit_class` classifies the worst failure: `.usage` for mistakes a flag fixes, `.limit` for resource refusals, `.conversion` for everything else. The CLI maps the classes to exit codes 2, 3, and 1. Your service can map the same classes to HTTP statuses or retry policies.

7.2 The options, one field at a time

```
pub const ConvertOptions = struct {
    input: InputSpec,           // .path, or .bytes with a display name
    output: OutputSpec,         // .path, or .writer (*std.Io.Writer)
    from: ?[]const u8 = null,   // explicit input format
    to: ?[]const u8 = null,     // explicit output format
    limits: Limits = .{},       // every resource bound, overridable
    overwrite: bool = false,    // replace existing artifact + manifest
    strict: Strictness = .off,  // graded refusal of declared loss
    preserve_facets: bool = false, // full facet rows in the manifest
    pipeline: ?*const Pipeline = null,
};
```

`input` as `.bytes` is how a server converts uploads without touching the filesystem. The `name` field exists only for reports and format detection. `output` as `.writer` streams the artifact anywhere and skips every file side effect. No adjacent manifest is written, but `manifest_json` still carries it. Path output stages complete temporary files, then publishes the artifact, extracted media, and finally the manifest. Each file

is individually atomic, and the last manifest certifies the ensemble. A crash between renames can leave a complete artifact or media file without a manifest; that state is deliberately uncertified and a rerun with `--overwrite` repairs it.

> **Strict is graded, and strict before commit** `Strictness` has four values. `.off` converts and reports. `.content` refuses when the priced plan would drop semantic content. `.structure` also refuses structural degradation, and `.exact` refuses any declared loss at all, styling included. Every grade is checked against the writer's lowering plan before the artifact is committed, so a pipeline that must never publish lossy output gets a hard guarantee, not a log line after the fact.

7.3 Python: the same conversion as a value

The `zenfmt` Python distribution packages the default bundle behind a typed, dependency-free API. Its common path is deliberately one call:

```
import zenfmt

conversion = zenfmt.convert("report.docx")
print(conversion.text)
for report in conversion.reports:
    print(report.code, report.problem)
```

With no `output`, the returned `Conversion` owns the artifact bytes, canonical manifest, reports, and every embedded resource. Supplying an output path asks the same native publisher used by the CLI to commit the complete ensemble:

```
conversion = zenfmt.convert(
    uploaded_bytes,
    name="upload.docx",
    output="build/report.md",
    strict="structure",
    limits=zenfmt.Limits(max_input_bytes=64 * 1024 * 1024),
)
```

A Python string is always a path, never inline document text. That explicit path grants read authority for the file and for one adjacent, digest-bound manifest probe. Bytes and binary readers grant no filesystem authority: their `name` is display-only and is never opened. External resource references are returned as metadata and never fetched. The library has no environment configuration, runtime download, plugin discovery, or network access.

Every Python failure follows the same Elm-style diagnostic shape as the CLI. Applications branch on `error.code`, retain structured reports, and show `str(error)` or `error.hint` to a person. Argument mistakes remain ordinary `TypeError` or `ValueError`, but their messages still state the problem, consequence, and a concrete next action under `What you can do:`.

7.4 Browser: the same engine as a WebAssembly package

The tagged browser distribution is also published as the dependency-free npm package `@insanai/zenfmt`. The package contains the audited WebAssembly module, ES module adapter, dedicated worker, TypeScript declarations, and generated capability contract.

```
$ npm install @insanai/zenfmt
```

```
import {
  createWorkerConverter,
  wasmUrl,
  workerUrl,
} from '@insanai/zenfmt';

const converter = await createWorkerConverter({ moduleUrl: wasmUrl, workerUrl });
const result = await converter.convert(file, { timeoutMs: 30_000 });
console.log(result.text);
converter.dispose();
```

The package does not add a network conversion fallback. The module has no host imports, and conversion remains on the caller's machine. npm is only a distribution option for browser applications. The native CLI and server stay self-contained and do not require Node or npm.

A wheel is not a sandbox The bundled native engine executes inside the Python process with that process's authority. Limits bound input, expansion, structure, resources, and output, but they do not create a process boundary. Convert higher-risk workloads in a restricted worker process or container, publish into a per-job quota-controlled directory, and create worker processes before starting conversion threads (or use multiprocessing `spawn`).

7.5 Bundles: bring only the formats you need

The CLI's 19 readers are a choice, not a floor. `core.Bundle` builds a conversion engine from any descriptor tuple, at compile time:

```
const core = @import("zenfmt_core");
const markdown = @import("zenfmt_markdown");
const docx = @import("zenfmt_docx");

const Slim = core.Bundle(.{
    .readers = .{ docx.reader, markdown.reader },
    .writers = .{markdown.writer},
});

// Slim.convert has the same signature and contract as zenfmt.convert.
```

Duplicate format names, colliding extensions, and a reader and writer sharing one plugin id are compile errors. The messages follow the same four-question structure as runtime reports. The engine inside the bundle knows no format names. Its tables are generated from exactly the descriptors you passed. A slim bundle is not a configuration of the big one. It is the engine, smaller.

7.6 Filters, in the manner of `build.zig`

zenfmt has no embedded scripting language. A filter is a Zig type. A pipeline is a value you build in your own program. The compiler checks the whole thing. This is the same trade `build.zig` makes: the extension language is the implementation language. There is no serialization boundary, no interpreter to sandbox, and no API surface that exists only in documentation.

Figure 11: A conversion, as the bundle runs it. Filters sit between the validated tree and the writer. Each stage revalidates.

7.6.1 The five that ship

Filter	Options	What it does
`filters.shift_headings`	`.{ .by = i8 }`	Adds a constant to every heading level, clamped to 1..6.
`filters.promote_first_heading`	`.{}`	Moves a leading level-1 heading into document metadata as the title.
`filters.drop_empty_containers`	`.{}`	Unwraps `container` and `span` nodes carrying no attributes. Genuinely useful after HTML and DOCX.
`filters.flatten_nested_tables`	`.{ .placeholder = "..." }`	Replaces a table nested in a table cell with a placeholder paragraph and files a warning report.
`filters.strip_classes`	`.{ .pattern = "tmp-*" }`	Removes classes matching an exact name or a `prefix*` wildcard.

Each exists twice over. It is a useful transform, and it is a worked example of one part of the contract: a typed payload edit, a metadata edit, `.unwrap`, a subtree replacement with a report, an attribute rewrite.

7.6.2 Writing your own

The `examples/filters/` project in the repository is a complete, compiling answer, and it is short enough to read whole. Its build depends on the zenfmt package. Its `main.zig` is a program:

```
pub fn filters(p: *zenfmt.Pipeline) void {
    p.add(zenfmt.filters.shift_headings, .{ .by = 1 });
    p.add(InternalLinks, .{ .base = "https://docs.example.com/" });
    p.add(zenfmt.filters.drop_empty_containers, .{});
}
```

`Pipeline.add` takes the filter type and a value of that filter's own `Options` type. Pass the wrong shape and the program does not compile. Order is exactly the order written. The custom stage is a declaration:

```
const InternalLinks = zenfmt.Filter(.{
    .id = "example.internal-links",
    .description = "Rewrite fragment links to absolute URLs",
    .options = InternalLinksOptions,
    .inline_tags = &.{.link},
    .idempotent = true,
    .visit_inline = visitLink,
});
```

Two fields deserve attention. `inline_tags` declares the candidate set. This filter can only ever edit `link` nodes, so the engine scans the tag column (one contiguous `u8` array) and skips the visit machinery entirely for a document containing no links. `idempotent` is a promise the test suite can check: running the stage twice equals running it once.
The visit function is ordinary Zig with a typed context:

```
fn visitLink(
    options: *const InternalLinksOptions,
    ctx: *zenfmt.FilterContext,
```

```
    node: zenfmt.ast.InlineIndex,
) zenfmt.FilterError!zenfmt.FilterAction {
    const link = switch (ctx.inlineView(node).content) {
        .link => |value| value,
        else => unreachable,
    };
    const target = ctx.text(link.url);
    if (!std.mem.startsWith(u8, target, "#")) return .keep;

    const absolute = try ctx.fmt("{s}{s}", .{ options.base, target[1..] });
    try ctx.replaceLinkTarget(node, absolute);
    return .replace;
}
```

Build it and the binary carries the pipeline:

```
$ zig build
$ ./zig-out/bin/zenfmt-filtered --list-filters
Filters, in pipeline order:
  core.shift-headings  Shift every heading level by a constant
  example.internal-links  Rewrite fragment links to absolute URLs
  core.drop-empty-containers  Unwrap containers and spans with no attributes
$ ./zig-out/bin/zenfmt-filtered --filters manual.md --stdout
## Guide

See [the intro](https://docs.example.com/intro) and [other](https://ziglang.org).
```

The fragment link became absolute. The `#` heading became `##`. Both transforms ran inside one binary that is also a complete zenfmt. The last line of the example's `main` hands the pipeline to `zenfmt.cli.main`.

> **Predict before reading on** A filter returns one of `keep`, `drop`, `unwrap`, and `replace`. Before reading on: which of the four can change the number of children the node's parent has, and which can leave the tree byte-for-byte identical to its input?

7.6.3 What a visit may do

`FilterContext` is the entire authority a stage has. For inspection it offers `block` and `inlineView` (typed views), `text` (byte ranges to slices), `hasClass`, `attribute`, and `parents`, the ancestor tag stack. The ancestor stack is how `flatten_nested_tables` knows it is inside a cell. For editing it offers the typed replacements (`replaceHeadingLevel`, `replaceLinkTarget`, `replaceBlockAttrs`, `replaceInlineAttrs`, `makeAttrs`), whole-subtree rebuilding (`beginReplaceBlock` hands you the same `Emitter` readers use; `commitReplaceBlock` seals it), and one document-level edit (`setMetaInlines`). Two utilities round it out: `fmt` allocates into the conversion's arena, and `report` files a diagnostic that lands in the manifest like any reader's.

What a visit cannot do is mutate the tree in place. Every action is recorded as an edit. The engine applies them all at once.

7.6.4 Discovery, then rebuild

A pipeline stage runs in two passes. The discovery pass visits candidate nodes in source order and collects edits. If the list is empty, the stage returns the input document: the same value, no copy of any kind. The identity case is not merely fast. It is measured. The test "an identity pipeline appends no AST storage" in `tests/filters.zig` asserts that the block, inline, and text arrays did not grow.

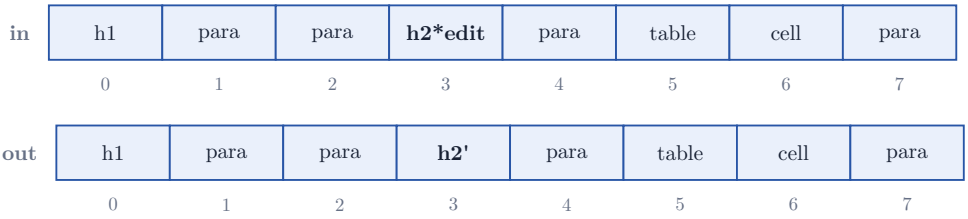

Figure 12: A sparse edit. One heading's payload changes. The runs before and after it move as single column-wise copies. Side tables (text, attrs, targets) are append-only and shared, never copied.

With edits, the rebuild pass copies the spine (the path from the root to each edit) and bulk-copies every untouched subtree, column by column. A subtree is a contiguous slice in preorder storage. So "copy this untouched chapter" is one `memcpy` per column, not a traversal.

Side-table indices survive a rebuild because those tables are append-only. A rebuilt tree's nodes still point at the same text pool, the same attrs, the same targets, plus whatever the stage appended. That is why `ctx.fmt` can hand a filter a new string without invalidating anything.

7.6.5 Failure and the validator

Every stage's output is validated before the next stage sees it: all eleven tree invariants. A stage that fails mid-rebuild rolls the store back by truncating to marks recorded at the stage boundary. A failed pipeline never leaks a half-rebuilt document into your error path. You get `.status == .failed` and either `core.invalid-document-tree` or the stage's own report.

> **Checkpoint: the cost model** Say it precisely. An identity stage costs one scan of one `u8` column per candidate tag. A sparse edit costs the spine plus one column-wise `memcpy` per untouched sibling run. Side tables are never copied. If you cannot justify the third clause from append-only indices, reread the discovery section.

7.7 Reading facets

A filter or an embedding application that wants more than flow can ask the document for its facets. The lookup is two steps, both cheap: node to entity, entity to facet row. A node without facets has no entity and the first step answers null, which is the whole cost of facets to code that ignores them.

```
if (doc.blockEntity(index)) |entity| {
    if (doc.gridOf(entity)) |cell| {
        // cell.sheet, cell.row, cell.col, cell.formula, cell.cached
    }
    for (doc.revisionsOf(entity)) |revision| {
        // revision.kind, revision.author, revision.timestamp
    }
}
```

Five accessors cover the five kinds: `provenanceOf`, `styleOf`, `layoutOf`, and `gridOf` return at most one row; `revisionsOf` returns a slice, because one span can carry an insertion and a comment at once. Layout coordinates are EMU from a top-left origin regardless of source format; the reader that attached them already did the arithmetic. Extracted binary content lives in the resource store, `doc.store.resources`, each entry carrying its source name, MIME type, bytes, and a BLAKE3 digest computed at registration.

7.8 The manifest as an integration surface

Every path conversion commits `<output>.zenfmt.json` beside the artifact. Every streamed conversion returns the same bytes in `manifest_json`. The encoding is the zenfmt canonical JSON profile: UTF-8,

bytewise-sorted keys, exact integers, no whitespace. It is deliberately not RFC 8785, which orders keys by UTF-16 code units and forces every number through a double; the profile is specified in `core/src/json.zig` and pinned by goldens. Two identical conversions are byte-identical, so a digest of the manifest is meaningful.

Key	What a consumer may rely on
`source`, `artifact`	Name, format, plugin id, and BLAKE3-256 digest of each side. The artifact digest is computed while streaming the write. If the file beside the manifest does not hash to it, the pair is stale.
`ast`	The tree schema name and version the conversion used.
`document_metadata`	The document's own metadata (title, authors, and so on) in the canonical metadata encoding.
`facets`	Present only when the document carries facets: one entry per kind with a row count, a digest of the canonical rows, and an `unused` marker saying whether the selected writer consumed the kind. With `preserve_facets` the full rows ride along, so a later same-family writer can recover them without the source.
`media`	Present only when media was extracted: one entry per committed file under `<stem>_media/`, path plus BLAKE3-256 digest.
`reports`	Every diagnostic, structured: code, severity, loss tier, locations, directions.
`plugins`	Namespaced preservation data, versioned per plugin id. A namespace a converter does not understand is carried value for value.
`schema`, `schema_version`	The envelope's own identity. Verify these before trusting anything else.

The `plugins` map is the round-trip channel. The DOCX reader records paragraph style ids under `ai.insan.zenfmt.docx`. A future DOCX writer, or your own tool, reads them back. The carry rule means a manifest can pass through tools that know nothing of a namespace without losing it.

Teach it back. Explain to a colleague why `convert` returning a value rather than an error union is a feature. Use the out-of-memory case as your hardest example. Then explain what an identity filter costs, and why the answer would be different if the tree were stored as heap nodes with pointers.

Exercise 7.1. Write a filter `external-links-nofollow` that appends `?ref=docs` to every `http(s)` link target. Which candidate tag set do you declare? Is your filter idempotent as specified? If not, what test would catch it?

Hint: `ctx.text(link.url)`, `std.mem.startsWith`, `ctx.fmt`, then `replaceLinkTarget`. Idempotence fails if you append unconditionally. Check for the suffix first.

Exercise 7.2. Your service converts untrusted uploads with `.bytes` input and `.writer` output. List every filesystem side effect that configuration can produce, and say where the manifest ends up.

Hint: Count the side effects again. The answer is a short list, and `manifest_json` is the whole story.

8 Reference

This chapter is the desk reference: the CLI, every format, every stable report code, every limit, and the design records, collected beside one another. Everything here is taken from the code's own tables. Where the book and the binary disagree, trust the binary and file the difference as a bug.

8.1 The command line

```
usage: zenfmt [options] INPUT
```

`INPUT` is a file path, or `-` for stdin. Stdin requires `--from` and either `-o PATH` or `--stdout`. The reason is plain: a stream has no extension to detect from and no name to derive the output path from.

Flag	Value	Meaning
`--from`, `-f`	`FORMAT`	Input format. Default: from the extension, then from content signatures.
`--to`, `-t`	`FORMAT`	Output format. Default: `markdown`.
`--output`, `-o`	`PATH`	Output file. Default: `INPUT` with the writer's extension.
`--stdout`		Write the document to stdout instead of a file. No artifact or manifest file is committed.
`--metadata-out`	`PATH`	Persist the manifest produced with `--stdout`. Only meaningful with `--stdout`.
`--overwrite`		Replace existing artifact and manifest paths. Without it an existing destination is a refusal (`core.destination-exists`).
`--preserve-facets`		Serialize full facet rows into the manifest instead of the default digest-and-count summaries.
`--filters`		Run the filter pipeline compiled into this binary.
`--list-formats`		Print the bundle's readers and writers with their extensions.
`--list-filters`		Print the filters compiled into this binary, in pipeline order.
`--strict`		Refuse declared loss before anything commits. Graded: bare `--strict` (content), `--strict=structure`, `--strict=exact`.
`--quiet`		Suppress notes.
`--reports`	`FORM`	`text` (default) or `json`.
`--limit`	`NAME=VALUE`	Override one resource limit. Repeatable. The names are in the limits table below.
`--help`, `-h`		Show the generated help.
`--version`, `-V`		Show the version.

8.1.1 Exit codes

Code	Class	Meaning
0	success	The artifact committed, and on path output the manifest with it. Notes and warnings may still be present. Read the reports.

Code	Class	Meaning
1	conversion	The input could not be converted: malformed content, an invalid tree, an I/O failure, or a graded `--strict` refusal of declared loss.
2	usage	The invocation is wrong: an unknown format, an undetectable input, a flag mistake. Fixable at the command line.
3	limit	A resource limit refused the input: archive bombs, depth bombs, oversized inputs, refused encryption. The security refusals have no override.

8.2 The formats

19 readers, one writer. Every reader's plugin id is `ai.insan.zenfmt.<format>`. Detection weighs both kinds of evidence: for in-memory input the content signature always runs, and when it contradicts the extension, content wins with a `core.extension-mismatch` note. File inputs whose extension routes are read directly. `--from` overrides everything.

Format	Extensions	Content signature	Facets	Record
`docx`	`.docx .docm`	ZIP central directory names `word/document.xml`	style, revision, layout, media	0003
`doc`	`.doc`	CFB magic + `WordDocument` stream	style, provenance	0012
`odt`	`.odt`	ZIP + OpenDocument text mimetype	style, revision, media	0006
`rtf`	`.rtf`	Leading `{\rtf`	style	0004
`xlsx`	`.xlsx .xlsm`	ZIP names `xl/workbook.xml`	grid	0005
`xlsb`	`.xlsb`	ZIP names `xl/workbook.bin`	grid	0012
`xls`	`.xls`	CFB magic + `Workbook` or `Book` stream	grid	0012
`ods`	`.ods`	ZIP + OpenDocument spreadsheet mimetype	grid	0008
`csv`	`.csv .tsv`	extension only	none	0002
`pptx`	`.pptx .pptm .ppsx .ppsm`	ZIP names `ppt/presentation.xml`	layout, media	0007
`ppt`	`.ppt .pps .pot`	CFB magic + `PowerPoint Document` stream	provenance	0012
`odp`	`.odp`	ZIP + OpenDocument presentation mimetype	layout	0009
`epub`	`.epub`	ZIP + `application/epub+zip` mimetype	provenance	0010
`pdf`	`.pdf`	Leading `%PDF`	layout, provenance, media	0011
`html`	`.html .htm`	extension only	extension nodes	0002

Format	Extensions	Content signature	Facets	Record
`markdown`	`.md .markdown`	extension only	none	0002
`asciidoc`	`.adoc .asciidoc`	extension only	none	0002
`rst`	`.rst`	extension only	none	0002
`text`	`.txt .text`	extension only. The fallback is never silent: an undetectable input is a usage refusal.	none	0002

Word processing. DOCX maps styles to headings through `basedOn` chains, synthesizes lists from numbering properties, folds merged table cells, and carries footnotes. Comments and text boxes are counted omissions; tracked insertions and deletions additionally ride as revision facets, and style names, page size, and image bytes are carried too. Legacy DOC resolves Heading 1 through 9 from the binary stylesheet and extracts hyperlink fields; other named styles survive as style facets. ODT resolves automatic styles, images, sections, and inline footnotes. RTF covers tables, lists, fields, footnotes, and outline headings.

Spreadsheets. All five spreadsheet readers project one sheet to one `h2` plus one table, first row as header, with typed cells. Dates render ISO. Percentages multiply out. Formulas keep their cached values, and a note says so.

Presentations. A slide projects to a title heading plus body content: lists, tables, links, images. Speaker notes land in a `container` classed `notes`. Every presentation conversion carries its projection warning. Charts and animation do not survive, and the reader says so rather than pretending; shape geometry, though, now rides in layout facets in EMU.

Publishing. EPUB walks container, then package, then spine, and parses chapters with the HTML machinery. DRM is refused outright. PDF is native Zig text extraction: xref and object streams, FlateDecode, ToUnicode CMaps, width metrics, heading tiers from font sizes. It carries a layout-projection note on every conversion, because a PDF has pages, not structure.

8.3 Report codes

Every diagnostic carries a stable code, asserted by at least one test. The catalog below groups the codes by namespace. Severity is the worst the code is issued at. Class is the exit class when the code is fatal; "none" marks notes and warnings, which fail a conversion only under `--strict`.

8.3.1 Engine (`core.*`)

Code	Severity	Class	Meaning
`core.out-of-memory`	error	conversion	The reserved allocation-failure report. Always available, never allocated.
`core.destination-exists`	error	conversion	The output exists and `--overwrite` was not given.
`core.file-operation-failed`	error	conversion	An open, write, flush, or rename failed. The report names the operation and path.
`core.host-io-unavailable`	error	usage	A file path was given to a build with no filesystem access compiled in, such as the browser module. Convert

Code	Severity	Class	Meaning
			the bytes instead, or use the CLI or Python library.
`core.writer-output-failed`	error	conversion	A writer or direct output sink stopped during emission. The result says whether a streamed prefix had already been delivered.
`core.input-too-large`	error	limit	The input exceeds `max_input_bytes`.
`core.output-too-large`	error	limit	Writer emission reached `max_output_bytes` before finishing, in any output mode. Nothing was published.
`core.invalid-limit-configuration`	error	limit	A library caller supplied a zero limit or exceeded a fixed hard cap.
`core.invalid-document-tree`	error	conversion	A plugin or filter produced a tree the validator rejects.
`core.nested-table-flattened`	warning	none	The `flatten-nested-tables` filter replaced an inner table.
`core.stale-or-invalid-manifest`	warning	none	An adjacent input manifest failed digest or schema checks and was ignored.
`core.strict-refused`	error	conversion	The graded `--strict` predicate refused the priced loss. Nothing was committed.
`core.strict-unavailable`	error	conversion	The selected writer has no total capability declaration, so strictness cannot be proved before emission.
`core.construct-refused`	error	conversion	The document contains a construct the selected writer refuses to degrade in any mode.
`core.invalid-lowering-plan`	error	conversion	A writer plugin failed to provide a bounded, complete lowering plan.
`core.manifest-encoding-failed`	error	conversion	Generated metadata or preservation data violated the canonical JSON contract; path output remains unpublished.
`core.invalid-preservation-data`	error	conversion	A reader's preservation JSON was malformed, oversized, or used a schema version different from its descriptor.
`core.lowering-limit`	error	limit	The bounded lowering planner reached its alternatives, work, or depth limit before opening output.

Code	Severity	Class	Meaning
`core.reports-truncated`	note	none	The report-group budget was reached and additional distinct groups were counted but omitted.
`core.extension-mismatch`	note	none	The file extension and the content signature disagreed; content evidence routed the file.
`cli.usage`	error	usage	The command line itself was invalid; the report highlights the offending argument.
`core.undetectable-input-format`	error	usage	No extension match and no content signature. The report lists every known format.
`core.unknown-input-format`	error	usage	`--from` or `--to` named a format this bundle does not carry. The report suggests the nearest name.

8.3.2 Markdown writer (`markdown.*`)

The writer's codes are loss notes. Markdown cannot represent the construct, the writer degraded it deliberately, and the manifest says so.

Code	Severity	Meaning
`markdown.style-dropped`	note	Underline, small caps, and similar styling have no Markdown syntax. The text is kept plain.
`markdown.extension-fallback`	note	A namespaced plugin extension the writer does not understand rendered as its source-neutral fallback content.
`markdown.container-attributes-dropped`	note	A container's id, classes, or attributes have no plain-Markdown form.
`markdown.cell-span-degraded`	note	GFM tables cannot span. Merged cells flatten into their first cell.
`markdown.table-cell-flattened`	note	Block content inside a cell became one line.
`markdown.table-caption-degraded`	note	A table caption became one flow line after the table because GFM has no caption association.
`markdown.nested-table-dropped`	warning	A table inside a table cell cannot be written. Its text content was kept.
`markdown.definition-list-degraded`	note	Definition lists render as emphasized terms plus indented paragraphs.
`markdown.list-number-style-degraded`	note	Alpha and roman list numbering becomes decimal.
`markdown.citation-dropped`	note	Citations render as their text. Keys are not preserved in Markdown.

Code	Severity	Meaning
`markdown.raw-dropped`	note	A raw block for another format was omitted.
`markdown.invalid-utf8`	error	The tree contained invalid UTF-8. That is an engine invariant violation surfaced at the writer.

8.3.3 Office Open XML (`docx.*`, `xlsx.*`, `pptx.*`)

Code	Severity	Class	Meaning
`docx.not-an-archive`	error	conversion	Not a ZIP container.
`docx.hostile-archive`	error	limit	Path traversal names or other hostile central-directory content. No override exists.
`docx.archive-limit`	error	limit	An entry count, size, ratio, or name-length limit tripped: the zip-bomb family.
`docx.encrypted`	error	conversion	OOXML encryption. zenfmt never attempts decryption.
`docx.unsupported-compression`	error	conversion	An entry uses a method other than stored or deflate.
`docx.doctype-refused`	error	limit	A DOCTYPE in a part. The XML parser refuses DTDs outright.
`docx.xml-too-deep`	error	limit	Element nesting beyond `max_xml_depth`.
`docx.malformed-xml`	error	conversion	A part fails to parse.
`docx.missing-document-part`	error	conversion	The package relationships name no main document.
`docx.unhandled-construct`	warning	none	Recognized WordprocessingML handled by nobody. Named in the report.
`docx.media-limit`	note	none	Image extraction stopped at the resource limits; later images keep their in-archive references.
`docx.merged-cells-degraded`	note	none	`gridSpan` and `vMerge` folded for the Markdown table.
`docx.comment-dropped`	warning	none	Comments are review apparatus, not content.
`xlsx.not-an-archive`	error	conversion	Not a ZIP container.
`xlsx.missing-workbook`	error	conversion	No readable `xl/workbook.xml`.
`xlsx.formula-without-cached-value`	note	none	Formulas are not evaluated. Cells without cached values are empty.
`pptx.not-an-archive`	error	conversion	Not a ZIP container.

Code	Severity	Class	Meaning
`pptx.missing-presentation`	error	conversion	No readable `ppt/presentation.xml`.
`pptx.presentation-projection`	warning	none	The standing loss statement: geometry, charts, and animation are absent.
`pptx.merged-cells`	note	none	Table span folding, as in DOCX.
`pptx.media-limit`	note	none	Embedded picture extraction stopped at the resource limits; the rest keep path references.

8.3.4 OpenDocument (`odt.*`, `ods.*`, `odp.*`)

Code	Severity	Class	Meaning
`odt.not-an-archive`	error	conversion	Not a ZIP container.
`odt.missing-content`	error	conversion	No `content.xml`.
`odt.malformed-xml`	error	conversion	`content.xml` fails to parse.
`odt.annotations-dropped`	warning	none	Review annotations are not content.
`odt.media-limit`	note	none	Image extraction stopped at the resource limits; later images keep their in-archive references.
`odt.frame-dropped`	warning	none	A frame with neither image source nor text.
`ods.not-an-archive`	error	conversion	As above, for spreadsheets.
`ods.missing-content`	error	conversion	No `content.xml`.
`ods.malformed-xml`	error	conversion	Parse failure.
`ods.annotations-dropped`	warning	none	Cell annotations omitted.
`ods.formula-without-cached-value`	note	none	Formulas keep cached values only.
`odp.not-an-archive`	error	conversion	As above, for presentations.
`odp.missing-content`	error	conversion	No `content.xml`.
`odp.malformed-xml`	error	conversion	Parse failure.
`odp.annotations-dropped`	warning	none	Annotations omitted.
`odp.presentation-projection`	warning	none	The projection statement, as in PPTX.

8.3.5 Legacy binary Office (`doc.*`, `xls.*`, `ppt.*`, `xlsb.*`)

Code	Severity	Class	Meaning
`doc.not-a-compound-file`	error	conversion	Not a CFB container.
`doc.missing-word-stream`	error	conversion	No `WordDocument` stream.
`doc.encryption-refused`	error	conversion	`fEncrypted` is set in the FIB. Never decrypted.

Code	Severity	Class	Meaning
`doc.styles-omitted`	note	none	The document carries no readable stylesheet, so headings could not be resolved.
`doc.tables-flattened`	note	none	Binary table structure degrades to paragraphs.
`doc.page-breaks-dropped`	note	none	Page breaks have no Markdown meaning.
`doc.embedded-objects-dropped`	note	none	OLE objects omitted.
`xls.not-a-compound-file`	error	conversion	Not a CFB container.
`xls.missing-workbook`	error	conversion	No `Workbook` stream.
`xls.encryption-refused`	error	conversion	A `FILEPASS` record. Never decrypted.
`xls.unsupported-biff`	error	conversion	Pre-BIFF8 workbooks are refused, not misread.
`xls.formula-without-cached-value`	note	none	Cached values only, as in XLSX.
`ppt.not-a-compound-file`	error	conversion	Not a CFB container.
`ppt.missing-document-stream`	error	conversion	No `PowerPoint Document` stream.
`ppt.encryption-refused`	error	conversion	A crypt-session container, detected before any output.
`ppt.presentation-projection`	warning	none	The projection statement.
`xlsb.not-an-archive`	error	conversion	Not a ZIP container.
`xlsb.missing-workbook`	error	conversion	No `xl/workbook.bin`.
`xlsb.sheets-unreadable`	error	conversion	The workbook lists sheets but none loaded. An empty result is never silent.
`xlsb.sheet-skipped`	warning	none	Some sheets loaded and some did not. The count is in the report.

8.3.6 Publishing (`epub.*`, `pdf.*`, `html.*`)

Code	Severity	Class	Meaning
`epub.drm-refused`	error	limit	`META-INF/encryption.xml` is present. Refused with no override.
`epub.missing-container`	error	limit	No `META-INF/container.xml`.
`epub.missing-package`	error	conversion	No OPF package, or no chapter loaded.
`epub.malformed-package`	error	conversion	The OPF fails to parse.
`epub.bad-archive`	error	limit	The ZIP layer refused the container.
`epub.archive-limit`	error	limit	A ZIP limit tripped.
`epub.missing-chapter`	warning	none	A spine item's file is absent. Counted.

Code	Severity	Class	Meaning
`epub.skipped-spine-item`	note	none	A non-XHTML spine item (cover image, stylesheet) was skipped. Counted.
`pdf.not-pdf`	error	conversion	No `%PDF-` header.
`pdf.malformed`	error	conversion	Unrecoverable file structure: xref, trailer, or object syntax.
`pdf.encryption-refused`	error	conversion	An `/Encrypt` dictionary. Even empty-password encryption is refused.
`pdf.limit`	error	limit	An object-count, size, or indirection bound tripped.
`pdf.unsupported-filter`	warning	none	A stream filter zenfmt does not decode (CCITT, JBIG2, and others), named in the report.
`pdf.unmappable-text`	warning	none	A font without a usable Unicode mapping. Counted per font.
`pdf.layout-projection`	note	none	The standing statement that page layout became linear text with heuristic structure.
`pdf.media-limit`	warning	none	Image extraction stopped at the media limits; the remainder is omitted.
`pdf.images-omitted`	note	none	Images that could not be extracted as-is.
`pdf.links-omitted`	note	none	Link annotations not carried.
`pdf.no-text`	warning	none	No extractable text. Likely a scanned document; the report suggests OCR.
`html.invalid-utf8`	error	conversion	The input is not UTF-8.
`html.too-deep`	error	limit	Element nesting beyond `max_depth`.
`html.skipped-embedded-content`	warning	none	An inline SVG graphic or a framed document was not converted. Script, style, template, and metadata elements are not reported: they are not document content.

8.3.7 Lightweight markup (`text.*`, `csv.*`, `rtf.*`, `asciidoc.*`, `rst.*`)

Code	Severity	Class	Meaning
`text.invalid-utf8`	error	conversion	Plain text must be UTF-8. zenfmt does not guess encodings.
`csv.invalid-utf8`	error	conversion	As above.
`csv.unterminated-quote`	error	conversion	A quoted field never closes. The position is in the report.
`csv.ragged-row`	note	none	Rows of differing width were padded to the widest.
`rtf.not-rtf`	error	conversion	No `{\rtf` group.

Code	Severity	Class	Meaning
`rtf.groups-too-deep`	error	limit	Group nesting beyond the reader's bound.
`rtf.unknown-control-words`	note	none	Control words zenfmt does not know, counted and sampled.
`rtf.images-dropped`	warning	none	`\pict` data was not extracted.
`rtf.objects-dropped`	warning	none	OLE objects omitted.
`rtf.nested-table-flattened`	note	none	Nested tables flatten into the outer cell.
`asciidoc.invalid-utf8`	error	conversion	Not UTF-8.
`asciidoc.include-refused`	warning	none	`include::` never reads other files. The directive is dropped and named.
`rst.invalid-utf8`	error	conversion	Not UTF-8.
`rst.include-refused`	warning	none	As above, for reStructuredText.

8.3.8 Server (`server.*`)

The zenfmt server (ZDS 0016) reuses the engine's report shape for every server-origin failure: same fields, same JSON, the `server.` prefix. The HTTP status travels beside the envelope; the exit class mirrors the engine's mapping.

Code	Severity	Exit class	Meaning
`server.head-too-large`	error	usage	The request head exceeded the connection's fixed 16 KiB buffer. HTTP 431.
`server.body-too-large`	error	limit	The request body exceeds `--max-body`; refused at the declared length when possible, at the cap otherwise. HTTP 413.
`server.unsupported-media`	error	usage	A content type or multipart shape the route does not accept. HTTP 415.
`server.missing-input`	error	usage	An empty body where a document was required. HTTP 400.
`server.invalid-query`	error	usage	A query parameter carries a value the route does not recognize. HTTP 400.
`server.invalid-request`	error	usage	A JSON body is malformed or contains a value the route does not accept. HTTP 400.
`server.unknown-route`	error	usage	No route matches the path; the admin plane in open mode answers identically. HTTP 404.
`server.method-not-allowed`	error	usage	The path exists under another method; `Allow` enumerates them. HTTP 405.
`server.unauthorized`	error	usage	No usable principal on an authenticated route. HTTP 401.

Code	Severity	Exit class	Meaning
`server.invalid-credentials`	error	usage	Login failed; identical timing and body whether the account exists or not. HTTP 401.
`server.forbidden`	error	usage	The principal's role is below the route's requirement, or an ownership rule was violated. HTTP 403.
`server.password-change-required`	error	usage	The account carries a one-time password; only the password-change route is permitted. HTTP 403.
`server.last-administrator`	error	usage	Refused deletion, demotion, or disabling of the final administrator. HTTP 409.
`server.rate-limited`	error	limit	The caller's rate bucket is exhausted; `Retry-After` is set. HTTP 429.
`server.busy`	error	conversion	Connection slots or the conversion cap are exhausted; `Retry-After` is set. HTTP 503.
`server.limit-override-forbidden`	error	usage	`?limit=` from a principal below administrator. HTTP 403.
`server.store-unavailable`	error	conversion	A store read or write failed or timed out; the log carries the detail, the envelope does not. HTTP 503.
`server.shutting-down`	error	conversion	The request arrived after the drain began. HTTP 503.
`server.out-of-memory`	error	conversion	The reserved static report, mirroring `core.out-of-memory`: preallocated, allocation-free to emit. HTTP 500.
`server.open-network-bind`	warning	none	Startup warning, not an HTTP response: open mode bound a non-loopback address.

8.4 Limits

Every bound has a name, a default, and a `--limit NAME=VALUE` override. Depth overrides above the hard cap of 4096 are refused. The cap sizes the walker stacks, and no input is worth an unbounded stack.

Name	Default	Bounds
`max_input_bytes`	512 MiB	Bytes read from one input document.
`max_depth`	256	Nesting of either node tree, and so every explicit walker stack.
`max_archive_entries`	4096	Entries admitted from one ZIP central directory.
`max_entry_uncompressed`	256 MiB	Expanded size of one archive entry.

Name	Default	Bounds
`max_total_uncompressed`	1 GiB	Expanded size across all read entries.
`max_compression_ratio`	200	Expansion ratio, checked during streaming decompression.
`max_entry_name_bytes`	1024	Archive entry name length.
`max_xml_depth`	256	XML element nesting.
`max_scan_chunk_bytes`	1 MiB	Scanner scratch per chunk.
`max_manifest_bytes`	16 MiB	Size of an adjacent manifest accepted on input.
`max_plugin_data_bytes`	4 MiB	One plugin-data namespace value.
`max_manifest_depth`	64	JSON nesting in an accepted manifest.
`max_report_samples`	4	Locations an aggregated report lists before counting the rest.
`max_reports_total`	16 Ki	Distinct aggregated report groups.
`max_resources`	256	Resources a reader may extract from one document.
`max_resource_bytes`	128 MiB	Total extracted resource bytes per document.
`max_nodes`	16 Mi	Kernel nodes, blocks plus inlines, per document.
`max_facet_rows`	1 Mi	Facet rows across all facet tables.
`max_decoded_text_bytes`	256 MiB	Decoded text pool bytes, distinct from `max_input_bytes`.
`max_lowering_alternatives`	8	Lowering alternatives a writer may declare per construct.
`max_lowering_work`	64 Mi	Lowering rule applications per conversion.
`max_output_bytes`	512 MiB	Artifact bytes a writer may emit, checked at the shared sink in every output mode (ZDS 0014).

8.5 Environment

Building and testing zenfmt needs exactly Zig 0.16.0. Building the design records and this book additionally needs Typst 0.15 or later. The benchmark optionally uses pandoc and Node.js (for anydoc). Both are competitors, not dependencies.

```
zig build                # the CLI into zig-out/bin/
zig build test           # the full suite
zig build fmt-check      # formatting
zig build docs           # ZDS records, index, and site
zig build benchmark      # the conversion benchmark (see its chapter)
```

8.6 The design records

The Zen Discussion records under `docs/zds/` carry the decisions this book describes. Each format record states its mapping table, its deliberate omissions with their report codes, and its round-trip expectations. That is the review-time half of the honesty contract. The report system is the runtime half.

Record	Title and scope
0001	The Zen Discussion Process: the lifecycle, numbering, and registry these records live in.
0002	zenfmt: Architecture and Implementation. The AST, the engine, filters, bundles, the manifest, diagnostics, the coding standard, and the delivery plan.

Record	Title and scope
0003	The DOCX Reader.
0004	The RTF Reader.
0005	The XLSX Reader.
0006	The ODT Reader.
0007	The PPTX Reader.
0008	The ODS Reader.
0009	The ODP Reader.
0010	The EPUB Reader.
0011	The PDF Reader.
0012	The Legacy Binary Office Readers: DOC, XLS, PPT, and XLSB, plus the CFB container.
0013	Layered Document IR and Writer Lowering: the IR v2 kernel schema, entities and sparse facets, extension nodes, the resource store, the lowering planner with graded strict, and the core contract repairs. Committed: the system this book describes.

> *The most consulted page of a reference is the one that admits what the system does not do.*
>
> - the omissions tables, in every format record

9 The Measure of the Tool

Learning contract By the end of this chapter you should be able to run the benchmark yourself, explain what each of the three measured quantities captures and what it deliberately includes, read the dashboard below without the prose, and argue about the one corpus file where zenfmt is not the fastest tool.

A converter's claims are cheap until a corpus arrives. This chapter measures zenfmt against the tools a reader would actually reach for instead: pandoc, the universal document converter, firecrawl's anydoc, the Rust converter whose format roster zenfmt set out to match, and Docling, the Python document-understanding toolkit, measured here in its model-free parser-only configuration. Everything below is generated. The tables, the bars, and the headline ratios come from `benchmarks/results/latest.json`, written by the same build step you can run tonight on your own machine.

9.1 Method

The three quantities **Wall latency** is what a person waits for: process start to process exit. **CPU time** is user time plus system time. It is what a server pays. **Peak RSS** is the high-water mark of resident memory. It decides how many conversions fit on a machine. All three come from the operating system's `wait4` rusage accounting of the finished child. There is no in-process self-measurement.

Each tool converts each corpus file to GitHub-Flavored Markdown. The output is discarded. Each tool is invoked the way its documentation recommends. One warm-up run is discarded, and the tables keep the **median of 5** measured runs. A tool runs only on formats its documentation claims. `unsupported` in the tables is itself a result. So is `failed`, which means an exit code other than zero on a format the tool claims.

Fairness is a design decision, not an accident:

- anydoc's Node.js launcher and pandoc's Haskell runtime startup are **in** the measurement, because they are in every real invocation. zenfmt's own process startup is measured identically.
- zenfmt is built with `-Doptimize=ReleaseSafe`. Bounds checks stay on. This is the mode the project ships. The competitors are their released binaries.
- The corpus is nobody's home turf. It holds real files from public sample repositories, Apache POI and LibreOffice test data, Project Gutenberg, and the W3C. It spans 16 files and every format family zenfmt reads.

Every file is real. Nothing in the corpus was authored for this benchmark.

Checkpoint: reproduction `sh benchmarks/fetch_corpus.sh` downloads the corpus, which is not committed. `npm install --prefix benchmarks/.anydoc @firecrawl/anydoc` installs anydoc, and `zig build benchmark-docling-setup` provisions the pinned model-free Docling environment. `zig build benchmark -Doptimize=ReleaseSafe` runs everything and rewrites both `results.md` and the `latest.json` this chapter renders. Numbers in this printing were measured on an Apple-silicon macOS machine. Yours will differ in absolute value and should not differ in shape.

file	family	origin
`report.docx`	DOCX	filesamples.com document sample
`memo.doc`	Word 97 binary	filesamples.com, authored in Word 9.0
`letter.odt`	OpenDocument text	filesamples.com
`notes.rtf`	RTF	filesamples.com
`sheet.xlsx`	XLSX	filesamples.com
`table.xls`	Excel 97 binary	filesamples.com
`grid.xlsb`	Excel binary workbook	Apache POI test corpus
`sheet.ods`	OpenDocument spreadsheet	filesamples.com
`slides.pptx`	PPTX	Apache POI corpus, a real ApacheCon deck
`deck.ppt`	PowerPoint 97 binary	Apache POI corpus, a 2.5 MB thesis defense
`slides.odp`	OpenDocument presentation	the ApacheCon deck, converted by LibreOffice
`book.epub`	EPUB	Project Gutenberg, Pride and Prejudice
`page.html`	HTML	Project Gutenberg, the same novel as one page
`data.csv`	CSV	FSU sample data, 25,000 rows
`article.pdf`	PDF	pdfobject.com sample article
`spec.pdf`	PDF	W3C accessibility test file

Table 4: The corpus: 16 real documents from public sources.

9.2 The headline

6.9x
speed ratio
anydoc wall time / zenfmt over 14 shared files

7.9x
CPU ratio
anydoc user + system CPU time / zenfmt

10.1x
peak memory ratio
anydoc peak resident memory / zenfmt

The three cards use AnyDoc because it shares the most successful corpus files with zenfmt. A ratio divides AnyDoc by zenfmt, so a value above one means AnyDoc used more of that measure in this run. The geometric mean is used because it treats proportional changes symmetrically. These values describe one corpus and one host; they are not quality scores.

9.3 Support is a result

Half of a converter's value is answering at all. Rows are corpus files. A filled cell means the tool converted the file successfully.

file	size	zenfmt	Docling	anydoc	pandoc	zenfmt wheel
`article.pdf`	18.4 KiB	✓	—	✓	—	✓

Comparison tool / zenfmt	Shared files	Speed	CPU use	Peak memory
Docling	5	190.4x	205.5x	47.5x
anydoc	14	6.9x	7.9x	10.1x
pandoc	6	18.2x	16.5x	16.6x

Table 5: Geometric-mean resource ratios over shared successful files. Each value is the comparison tool divided by zenfmt. Larger than one means the comparison used more elapsed time, CPU time, or peak memory in this run.

file	size	zenfmt	Docling	anydoc	pandoc	zenfmt wheel
`book.epub`	545.5 KiB	✓	—	✓	✓	✓
`data.csv`	618.5 KiB	✓	✓	✓	✓	✓
`deck.ppt`	2519 KiB	✓	—	✓	—	✓
`grid.xlsb`	8.9 KiB	✓	—	failed	—	✓
`letter.odt`	4.2 KiB	✓	—	✓	✓	✓
`memo.doc`	32 KiB	✓	—	✓	—	✓
`notes.rtf`	32.4 KiB	✓	—	✓	✓	✓
`page.html`	832.6 KiB	✓	✓	—	✓	✓
`report.docx`	33.6 KiB	✓	✓	✓	✓	✓
`sheet.ods`	6.1 KiB	✓	—	✓	—	✓
`sheet.xlsx`	12.9 KiB	✓	✓	✓	—	✓
`slides.odp`	466 KiB	✓	—	✓	—	✓
`slides.pptx`	633.1 KiB	✓	✓	✓	—	✓
`spec.pdf`	13 KiB	✓	—	✓	—	✓
`table.xls`	13 KiB	✓	—	✓	—	✓

Each column tells its own story. pandoc's is honest minimalism: it never claimed the binary Office formats, the OpenDocument spreadsheet and presentation, or PDF input. Docling here is deliberately narrowed to its model-free parsers — Office Open XML, HTML, and CSV — so its blank cells are a benchmark choice, not a Docling limit; the next section explains why. anydoc's one `failed` is the real-world XLSB workbook. Its sheet directory uses a 40-byte `BrtBundleSh` record where the specification's example shows 36 bytes. Chapter 4 describes how zenfmt detects this variant by exact-consumption parsing.

9.3.1 Docling, parser only

Docling is a document-understanding toolkit, not a plain converter: its strength is layout, OCR, and table models over PDFs and images. Those pipelines load machine-learning weights and are a different workload from the millisecond structural conversion this chapter measures. The benchmark therefore pins Docling to its model-free backends and denies every model download, so the row measures Docling's own parsers converting Office Open XML, HTML, and CSV to Markdown — and nothing of its AI features. The cost it still carries is a real one: a fresh Python interpreter that imports the toolkit's scientific stack before any document work begins, which is why its bars sit whole seconds above the compiled tools even on the files it does convert. The comparison is narrow on purpose; it keeps the workload appropriate for the modest machines zenfmt targets.

9.4 Latency

> **Predict before reading on** Before reading the bars, consider this. The corpus has a 2.5 MB legacy PowerPoint deck and a 6 KiB OpenDocument spreadsheet. Which will cost zenfmt more wall time? Will the ordering be the same for the other tools?

The log axis is doing real work, because the tools live on different decades. Most zenfmt bars sit in the single digits to low tens of milliseconds; the 33 KiB DOCX converts in about four. That time is dominated by actual parsing, which is why the 2.5 MB `deck.ppt` costs no more than a small spreadsheet: the reader touches the text atoms it projects and skips the rest. The compiled competitors' bars start near their runtime startup floor, about 40 ms for anydoc's Node launcher and a similar amount for pandoc's

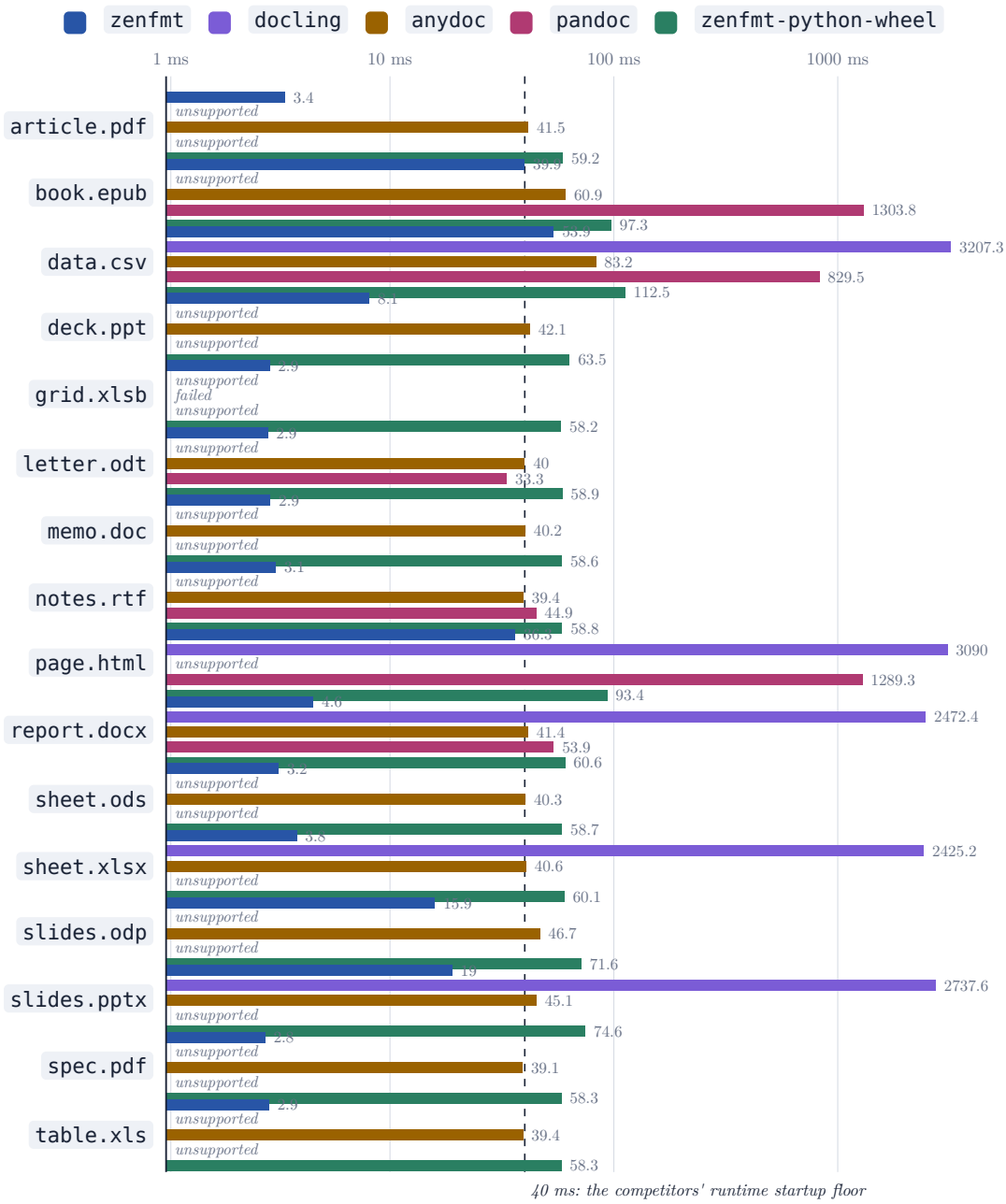

Figure 13: Median wall latency per conversion, log-10 axis. Absent bars carry their reason in italics. The dashed rule marks the startup floor the interpreted runtimes pay before any document work begins.

runtime, before any document work happens. Docling sits a whole decade higher still: even on the files it converts, a fresh interpreter imports its scientific stack before the first byte is read, so its floor is measured in seconds, not milliseconds.

zenfmt's heaviest file is `data.csv`: 25,000 rows at about 54 milliseconds, its slowest by a small margin. The cost is deliberate — zenfmt measures every column across every row so it can emit width-aligned GFM table pipes, an O(rows × columns) pass that anydoc skips by emitting ragged ones — yet even here zenfmt finishes ahead of anydoc. On large structured inputs the ordering is not close: pandoc climbs to about 1.3 seconds on the EPUB book and the 850 KiB HTML page, where zenfmt stays well under a tenth of a second.

9.5 CPU use

CPU time measures processor work rather than what a person waits for. It includes user and operating-system time reported for the child process. On this single-conversion workload it follows wall latency

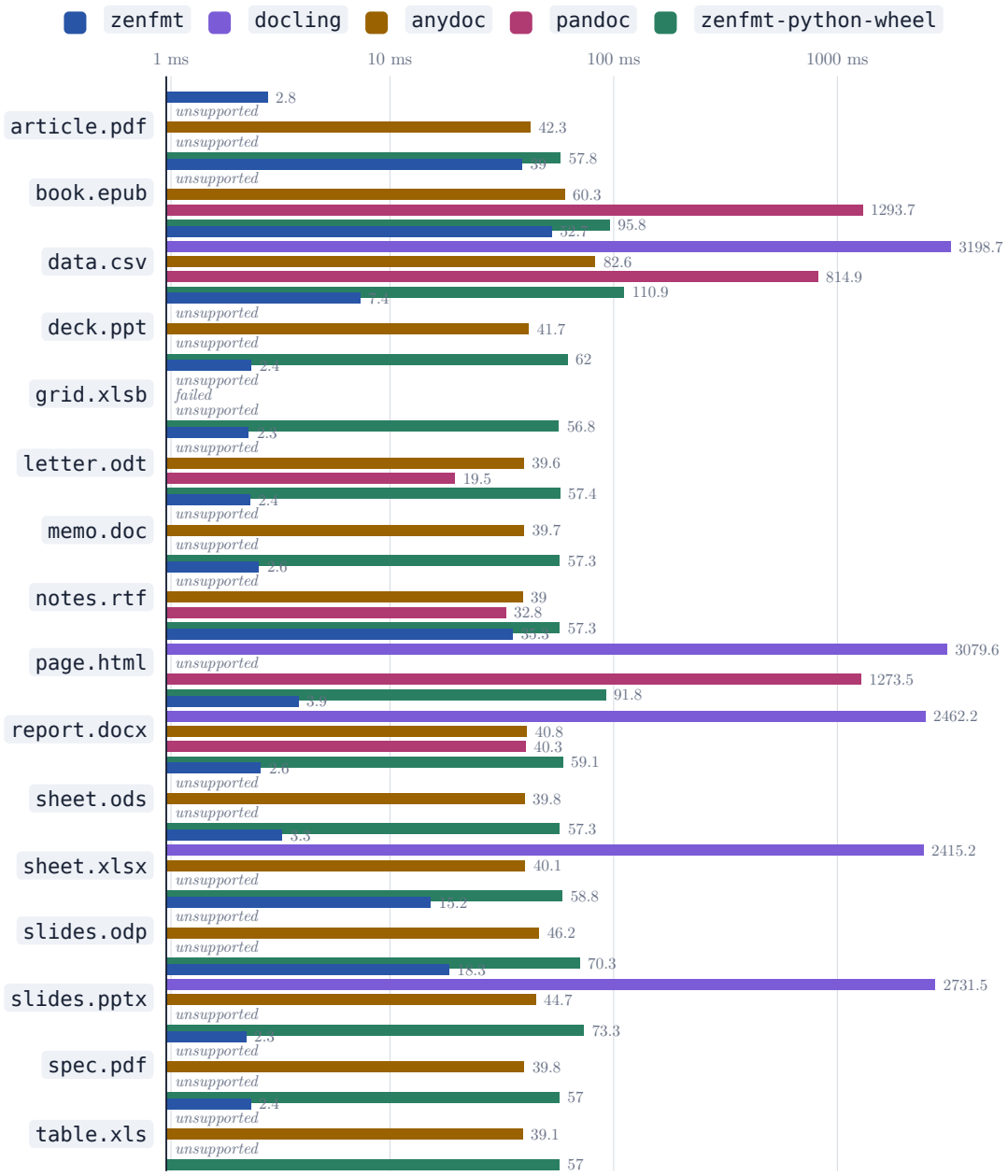

Figure 14: Median user plus system CPU time per conversion, log-10 axis.

closely, but it remains separate because a service pays CPU even when requests overlap. The ratio table above uses the same shared successful files as the speed comparison.

9.6 Memory

Memory tells the architecture story more plainly than latency does. zenfmt's peak sits a small constant above the input size. There is one arena per conversion, flat struct-of-arrays storage, and no DOM. pandoc builds a full tree in a garbage-collected heap: over 100 MB for a 33 KiB DOCX, and more than 400 MB for the HTML page. anydoc pays a flat 48 MB before documents enter the picture. On the shared files the geometric-mean gap is an order of magnitude. It widens exactly on the inputs where memory matters.

9.7 The shape of the win

One more picture makes the distribution visible. Take each file both zenfmt and anydoc convert. Divide anydoc's median wall time by zenfmt's. Sort. The result is not one lucky file carrying an average. All 14

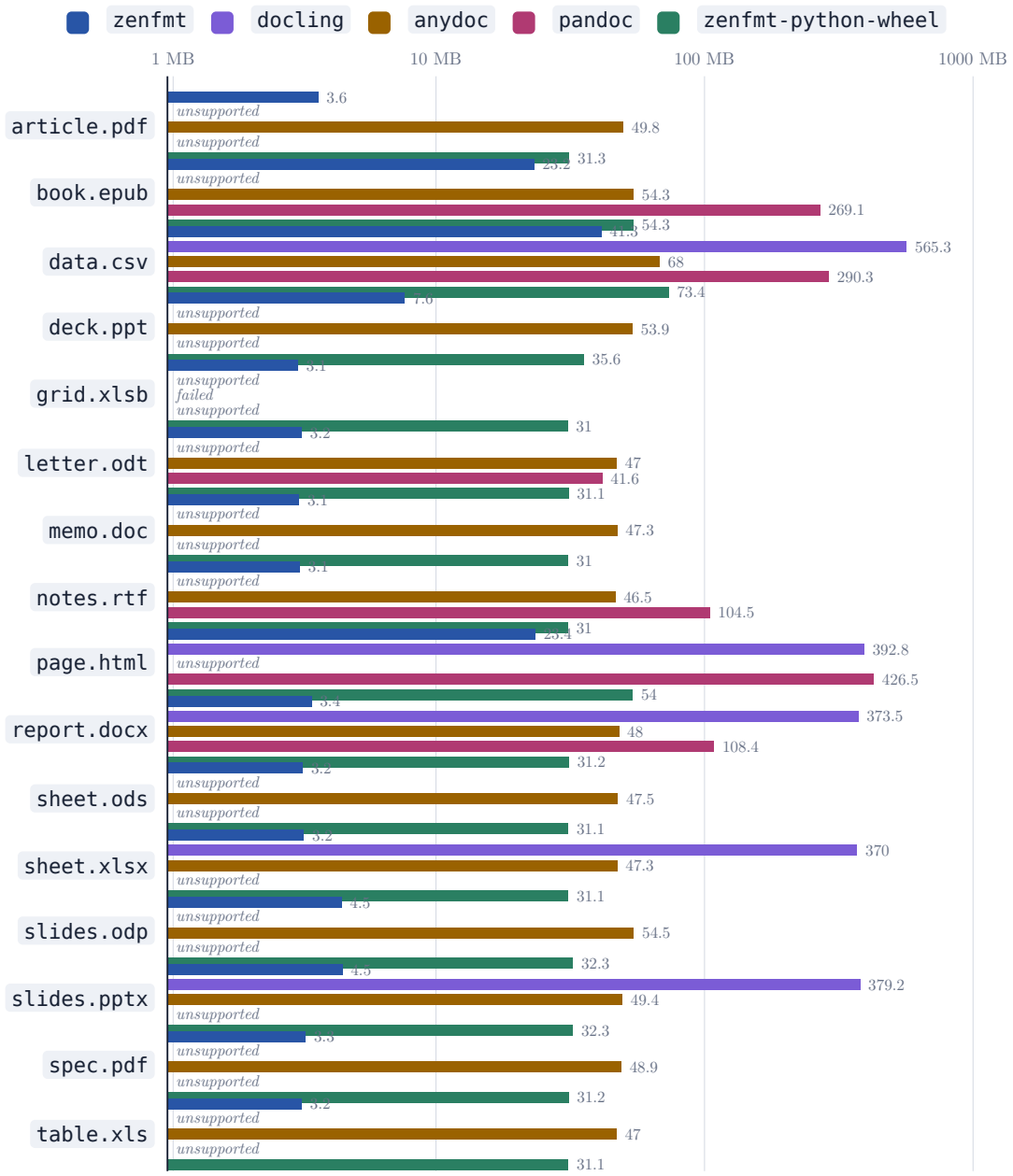

Figure 15: Peak resident set size per conversion, log-10 axis.

shared files land on the winning side of parity; `data.csv` is the closest at 1.5x. The spread tells you the rest: small files are dominated by the competitor's startup, and large files are dominated by parsing.

9.8 Where the time goes

The process benchmark treats each tool as a black box. A second harness, `zig build benchmark-stages`, opens zenfmt's box from inside the library: for each corpus file it times a conversion through a probe writer that emits nothing, which prices reading, tree building, validation, and the manifest. A timed wrapper then measures the ordinary Markdown writer callback directly. The residual after subtracting both from total is the lowering share; it also includes the small writer setup and finalization difference. That residual is derived, not directly measured, and the results file says so in a `derived` field. For a focused profile, append `-- --file data.csv --iterations 25`; the default remains five runs over the full corpus.

Two facts fall out. Parsing dominates the container formats: an ODT or a PPTX spends almost all its time inside the archive and XML, and the Markdown writer is nearly free. The 633 KiB CSV now spends 10.4 ms reading, 6.6 ms in the lowering residual, and 9.9 ms in the writer. Stage separation found the

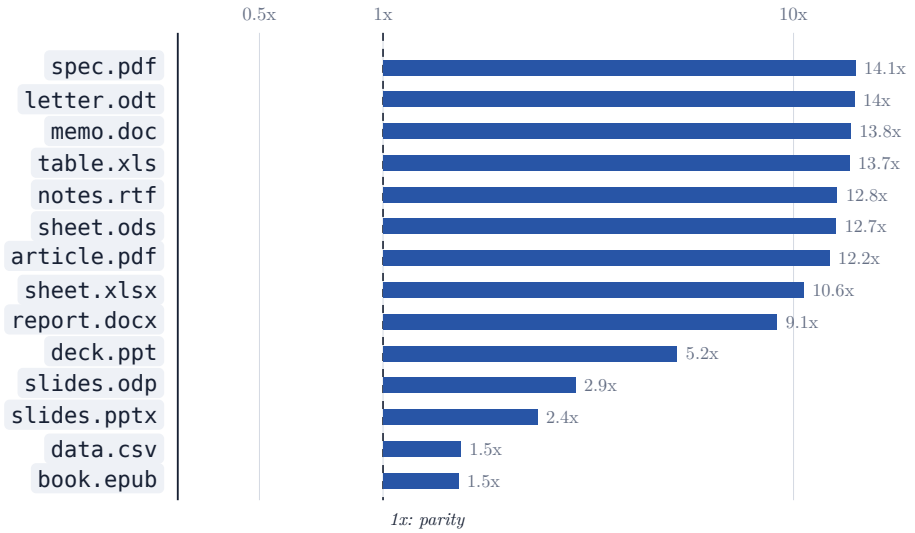

Figure 16: Wall-time speedup of zenfmt over anydoc per shared corpus file, sorted, log-10 axis. The dashed rule is parity.

File	Read (ms)	Lower (ms)	Write (ms)	Total (ms)
`book.epub`	18.31	2.8	5.71	26.82
`data.csv`	11.05	7.28	10.98	29.31
`deck.ppt`	3.81	0.04	0.16	4.01
`page.html`	12.87	2.95	5.91	21.72
`report.docx`	0.47	0.02	0.03	0.52
`sheet.xlsx`	0.36	0.02	0.02	0.4
`slides.odp`	5.9	0.08	0.11	6.08
`slides.pptx`	4.93	0.2	0.11	5.25

Table 6: In-process stage split, median of 5 runs, for the eight files with the most work to split. Read covers parsing, building, and validation; write is measured inside the callback; lower is the explicitly marked residual.

0.3.0 regression: validation and lowering each created hard-cap-sized scratch storage repeatedly as they visited a wide document. The current implementation allocates one scratch area for each validation or lowering plan and reuses it synchronously. That keeps the same bounds and decisions while avoiding work proportional to the node count times the hard cap.

9.9 The Python wheel

The same engine ships to Python as the `zenfmt` wheel (ZDS 0014), and the benchmark measures what users actually install: `zig build benchmark-python` builds the wheel, installs it into a clean isolated environment, verifies artifact parity against the same-revision CLI, and only then times it. The detailed results land in `benchmarks/results/python.json`; the process harness adds a cold `zenfmt-python-wheel` row beside the CLI. Cold and warm numbers answer different questions and are never merged into one headline.

Threads	Documents	Wall (ms)	Docs / s
1	80	855.7	93.49
2	80	441.66	181.14
4	80	237.71	336.54
8	80	173.7	460.56

Table 7: One immutable `Converter` shared across worker threads over independent corpus documents. The GIL is released for every native call, so throughput scales with cores until conversion saturates memory bandwidth.

45.7 ms
cold import
fresh interpreter, `import zenfmt`

49.5 ms
cold first conversion
import, load, verify, convert

1.4802 ms
warm memory call
median corpus file, in-process

0.29 ms
boundary microbenchmark
tiny text; FFI + copies + models

The boundary microbenchmark is reported separately and never used to claim corpus throughput: it exists so native loading, validation, copying, and model construction stay visible when parsing work is negligible. Parity ran before timing: 16 corpus files compared for format ids, artifact digests, resource digests, and report codes — all agreed.
Head to head on the shared corpus, the wheel's cold child-process row — a fresh interpreter per document, directly comparable to the CLI row — runs at 9.9x the CLI's wall time (geometric mean over 16 files; above 1.0 means the CLI is faster). The difference is interpreter start plus one-time bridge verification; the warm rows above are what a long-running service pays.

9.10 The server lens: against Apache Tika

The rows above measure a converter that starts, converts one file, and exits. A different question is how the long-running *service* compares, and the natural comparison there is Apache Tika, the tool teams reach for when they want extraction behind a port. `zig build benchmark-server` starts zenfmt in open mode and a pinned Apache Tika Server (4.0.0-beta-1, its documented Markdown handler) on loopback, and writes `benchmarks/results/server.json`. These numbers are never merged with the native rows above: process startup, HTTP transfer, and service isolation are different costs.

65x
startup ratio
zenfmt is ready in 0.11 s; Tika's JVM and parser pool take 7 s; the ratio is Tika divided by zenfmt

34x
sampled resident ratio
53 MB resident against Tika's 1815 MB parent and direct parser children

31x
shared warm ratio
geometric mean over the 16 files both services convert, each service warmed to steady state first; the ratio is Tika divided by zenfmt

Warm latency here measures steady state: both services convert one discarded warm-up per file before the timed samples, because Tika's per-client parser mode pays several seconds per forked worker on its first requests. zenfmt had lower median latency on all 16 shared files in this run. The closest was `data.csv`, where Tika used 2.3x the wall time. This is one host and corpus, not a claim about every service workload. Concurrent throughput, in documents per second, scaled with cores for zenfmt in this short run:

concurrency	zenfmt	Tika
1	553 /s	4 /s

concurrency	zenfmt	Tika
2	1075 /s	8 /s
4	2016 /s	14 /s
8	2655 /s	15 /s

One caveat belongs to Tika, not against it. Tika 4 parses in forked child processes, so a parser that panics or exhausts memory takes down a worker, not the service. zenfmt converts in-process; the deployment guide requires a supervisor and operating-system limits for exactly this reason. The speed and memory numbers are real, and so is that difference; the record keeps both visible rather than collapsing them into a single verdict.

9.11 Reading it honestly

These measurements need their caveats stated plainly.

- **Startup is part of the story, but not all of it.** Subtract the 40 ms runtime floor from every anydoc bar, and its document work is competitive on small files. zenfmt's advantage there is the sum of native start and arena parsing. It is not evidence that the Rust code is slow. On the large inputs the gap survives the subtraction.
- **Completeness is bounded by the corpus.** Sixteen real files cover every reader once. They do not cover every construct. The per-format fidelity claims rest on the test suites and the ZDS mapping tables, not on this chapter.
- **PDF numbers measure text extraction**, structure heuristics included. They do not measure OCR. Scanned documents are refused with `pdf.no-text` rather than converted into silence.
- **One machine, one printing.** The shapes replicate across machines. The absolute values are yours to re-measure with one build command.

Teach it back. Explain to a colleague why the benchmark keeps the median of 5 runs and not the mean, why it uses the geometric and not the arithmetic mean across files, why a runtime's startup belongs inside the measurement, and what the empty cells in the support matrix are evidence of. Then run the benchmark on your machine and recompile this book. If the argument survives your own bars, it was an argument and not a printing.

Exercise 9.1. Add another corpus file: a Markdown document of at least a megabyte. Predict all three of zenfmt's bars before running. Which tool's ordering changes? pandoc is a Markdown-native tool. Why does it not obviously win this file?

Hint: `benchmarks/fetch_corpus.sh` shows the naming convention; the harness picks up any file in the corpus directory. pandoc must parse **and** re-serialize; zenfmt's round-trip is measured in the fixed-point suite.

10 The Server

zenfmt converts documents in-process through the engine, the CLI, the Python library, and a browser WebAssembly build. The server (ZDS 0016) adds the missing surface: a long-running process that accepts documents over HTTP and answers with Markdown and structured reports, in the shape teams already run Apache Tika for. It starts in one command, and it grows into an administered service only when the deployment warrants it.

10.1 Two postures

Open mode is one command with no state:

```
zenfmt serve
```

It binds `127.0.0.1:8998`, converts whatever arrives, and serves a small converter interface. Every request acts as an anonymous user; nothing touches disk. This is the drop-in Tika-class deployment.
Secure mode is a deliberate upgrade:

```
zenfmt serve --secure --data-dir /var/lib/zenfmt
```

It turns on accounts, sessions, API keys, an audit trail, and the administration interface, all stored in an embedded database under the data directory. On the first start it prints a one-time administrator password to stderr, framed apart from the log stream; that password must be changed at first login and is never shown again.
Binding a non-loopback address in open mode is permitted but loud: the server emits the `server.open-network-bind` warning at startup, because every network peer can then convert documents anonymously. For a service reachable beyond loopback, run secure mode behind a TLS-terminating reverse proxy.

10.2 The conversion API

The request body is the document; metadata rides in the query string and headers, so `curl` stays one line:

```
curl -s -T report.docx \
  "http://127.0.0.1:8998/api/v1/convert?to=markdown" \
  -H "Accept: text/markdown"
```

`?to=` selects the writer, `?from=` overrides detection, and `?strict=` selects the strictness grade with the same spellings as the CLI (`content`, `structure`, `exact`). `PUT` is accepted as an alias for `POST` to ease Tika muscle memory.
With `Accept: text/markdown` the response is the artifact bytes, chunked, with `X-Zenfmt-Report-Count` and `X-Zenfmt-Exit-Class` summary headers. With `Accept: application/json` the response is the envelope: `status`, `artifact`, `artifact_name`, `resources`, `manifest` (the canonical manifest embedded verbatim), `reports`, `exit_class`, `source_format`, and `output_format`. The field names and semantics mirror the Python `Conversion` model, so a client that already reads one reads the other.
A failed conversion returns the failure envelope with `status: "failed"`, the engine's own reports, and no artifact. The HTTP status follows the exit class: usage is `400`, limit is `413`, conversion is `422`. The server never rewrites a report; API callers and the interface render the identical structured diagnostic.

10.3 Discovery, health, and metrics

`GET /api/v1/formats` returns the capability document: the readers, writers, extensions, and the limits this deployment actually enforces. `GET /api/v1/status` returns the version, mode, uptime, and bounded counters.

`GET /openapi.json` returns the embedded OpenAPI 3.1 contract. The web interface presents its practical entry points at `/docs`, which remains available without a session in both modes. Protected account and administration pages still require sign in.

The operational plane is unauthenticated in both modes and exposes no data beyond check names: `GET /healthz` is liveness, `GET /readyz` is readiness (the worker pool and, in secure mode, a store ping), and `GET /metrics` is Prometheus text exposition. Every metric carries the `zenfmt_` prefix and only comptime-enum label values, so a scrape allocates nothing and cardinality is bounded by construction. The first release serves the operational plane on the main listener.

The monitoring runbook maps the shipped signals to metrics. These signals cover error rate through `zenfmt_http_requests_total`, saturation through `zenfmt_http_rejected_total` and `zenfmt_conversions_active`, and authentication pressure through `zenfmt_auth_failures_total`.

10.4 Accounts, sessions, and keys

Secure mode has exactly two roles, `administrator` and `user`, so the authorization matrix stays small enough to test exhaustively. A browser authenticates with a session cookie and a per-session CSRF token, which the interface sends on every state-changing request; a program authenticates with an API key presented as `Authorization: Bearer zfk_<id>.<secret>`. Bearer requests are CSRF-exempt by construction.

Passwords are stored as Argon2id PHC strings (the OWASP interactive profile); tokens and key secrets are stored only as SHA-256 digests and compared in constant time. A key's secret appears exactly once, in the response that creates it. The last administrator cannot be deleted, demoted, or disabled — the API refuses with `server.last-administrator`, and the interface disables the impossible action for guidance while the API stays authoritative.

Administrators manage accounts under `/api/v1/users` and read the audit log under `/api/v1/audit`. The audit table records administrative fact — account changes, logins, key lifecycle, server start and stop — under a closed verb set, with bounded retention. Conversion requests are metered in metrics and logs, not audited per request.

10.5 The interface

The web interface is a Zig program. It follows the architecture Zig uses to serve its own standard-library documentation: a static HTML shell, one fixed JavaScript glue file, and a WebAssembly module compiled from Zig that renders every page. The module owns all interface state and markup; the glue holds no product logic, only the browser bridge. Styling is vendored daisyUI 5 over Tailwind 4 — a committed stylesheet, generated offline, with no CDN, no npm build at CI time, and no third-party runtime code. Material style surfaces, elevation, visible state, and large interaction targets provide the component language. Bold editorial typography and generous spacing keep the page easy to scan. Routine conversion has one clear path from document selection to the Convert button. Detailed controls remain visually secondary, while account changes and destructive administration ask for deliberate confirmation. A System / Light / Dark theme selector follows the operating system by default and remembers an explicit choice.

The interface is the first client of the public REST API: it calls exactly the documented routes, with no private endpoints. It requires JavaScript and WebAssembly; the `<noscript>` page shows the copy-pasteable `curl` line, because the API, not the interface, is the no-script path. The OpenAPI document,

vendor styles, first party styles, JavaScript bridge, HTML shell, and interface WebAssembly are embedded in the same executable.

10.6 Deployment

The server is one binary, one optional directory, and one port. A released `zenfmt` contains the engine, the HTTP service, the embedded database and its migrations, and every interface asset; it needs no adjacent bundle, Java runtime, Python environment, model directory, npm tree, or repository checkout. The `-Dserver=false` build is the smaller pure-converter binary for size-sensitive targets.

The standard library ships no TLS server, so in-process TLS is out of scope. The supported deployments are loopback (the default), a private network with `--secure`, or a TLS-terminating reverse proxy with `--behind-proxy`. That option asserts that a trusted TLS terminating proxy protects the server and marks cookies `Secure`. The server ignores `Forwarded` and `X-Forwarded-For`, so attribution and rate limiting use the immediate socket peer. The deployment guide ships nginx and Caddy fragments and a manual verification checklist.

Upgrades are stop, replace the binary, start; migrations run forward automatically, and a store whose schema is newer than the binary refuses to start rather than risk corruption. Backups are an operator action through the embedded database's own tooling against the data directory.

10.7 Tika migration

The server is comparable to Tika Server as an operational extraction service, but the first release is not a wire-compatible replacement. The accepted `PUT` method helps existing scripts, but zenfmt does not expose Tika paths such as `/tika`, `/rmeta`, `/meta`, or `/unpack`, and it does not copy Tika's response metadata or error bodies. A migration changes the request path to `/api/v1/convert` and chooses either zenfmt Markdown or the JSON envelope explicitly.

Tika request	zenfmt equivalent
`PUT /tika` (text extraction)	`PUT /api/v1/convert?to=markdown` with `Accept: text/markdown`.
`PUT /rmeta/text` (metadata + text)	`PUT /api/v1/convert` with `Accept: application/json`; the manifest carries the structured metadata.
`/meta`, `/unpack`, `/detect`	Not offered; detection is automatic and resources travel inside the JSON envelope.