Zen Working Group
Request for Discussion and Implementation Record

ZDS 0002
Architectural Specification

zenfmt: Architecture and Implementation

STATE
committed

INTENDED STATUS
Committed

CREATED
2026-08-06

AUTHORS
Zen Contributors <team@insan.ai>

LAST UPDATED
2026-08-08

DISCUSSION

Architecture and first-release plan for the zenfmt document converter

LABELS
architecture, ast, filters, plugins, formats

Status of This Memo

This document is an internal Zen discussion record authored in Typst and tracked in git. It intentionally follows RFC-style structure so the design scope, rationale, trade-offs, and operational constraints remain explicit. Documents that still use the placeholder number XXXXX are provisional drafts. Numbered ZDS documents are part of the project record.

Contents

12. Abstract

zenfmt is a document converter inspired by Pandoc's AST-and-filter pipeline: it parses a document into an abstract syntax tree, optionally transforms that tree, and serializes it in another format. Its AST and APIs are designed for Zig. This record specifies the whole system.

Four commitments shape the design.

The representation is a real AST: blocks nest, inlines nest, metadata is recursive and typed, and every node may carry an identifier, classes, and key-value attributes. It is **stored** as flat preorder struct-of-arrays in one arena — a technique also used by Zig 0.16's `std.zig.Ast` — which builds the AST out of typed `u32` indices rather than pointers. The public model and the storage model are deliberately separate: plugins see typed accessors, not packed-field folklore.

Filters are first-class and written in Zig, in the manner of `build.zig`: a filter is a Zig type with `visitBlock` and `visitInline` methods, registered into a pipeline by a `pub fn filters(p: *Pipeline) void` entry point that a user writes in their own project and compiles against the zenfmt module. There is no embedded scripting language and no serialization boundary. Because a transform rebuilds rather than mutates, unchanged contiguous node ranges can be copied in bulk. The exact cost is measured; the design does not promise one `memcpy` for a mixed block/inline subtree with cross-array references.

The engine knows no formats. Each format is a separate Zig library module exporting reader and/or writer descriptors plus its preservation-data codec. An umbrella module assembles the default comptime registry used by the CLI; embedders can assemble a smaller bundle.

Every file output has an adjacent provenance manifest. For `report.md`, the engine writes `report.md.zenfmt.json` in the same directory. It records the AST schema, source and output digests, document metadata, diagnostics, and versioned plugin-owned preservation data. Unknown plugin namespaces are carried forward unchanged. This lets, for example, a future DOCX writer recover details saved by the DOCX reader without putting DOCX concepts in the AST or engine.

The first release reads the office formats and writes Markdown, with the lightweight markup formats — AsciiDoc, reStructuredText, and the rest — following as plugins against an unchanged core. This record defines the AST, the traversal and rewriting algorithms, the filter contract, the plugin contract, the registry and static router, the memory model, the Elm-style diagnostic reports, OOXML ingestion and its security limits, the Markdown writer's output rules, the adjacent JSON manifest, the coding standard, the CLI, the test strategy, and the phased plan.

13. Introduction

A document converter is a compiler, and is best built like one: a front end per input language, a shared tree in the middle, a back end per output language, and a pass pipeline between them. The comparison predicts where the difficulty lives. It is not in any single parser. It is in the tree, which every front end must reach and every back end must serve, and which cannot be changed later without touching everything.

Pandoc demonstrates that a shared block/inline tree and filters are a durable shape for a converter. zenfmt takes those ideas, not its concrete data model. Pandoc is written in Haskell and naturally exposes recursive algebraic data types. Zig has different strengths: compact explicit storage, tagged unions, typed integer indices, comptime schema generation, and exhaustive switches. The zenfmt AST is designed around those strengths and around the office and markup formats in scope here.

zenfmt separates the tree's **public semantics** from its **physical storage**. Where a conventional recursive AST uses heap objects and pointers, zenfmt stores its own document model in flat arrays of fixed-size records in an arena, with typed `u32` indices for edges. This is a representation choice, and Zig 0.16′s own `std.zig.Ast` likewise uses `std.MultiArrayList`, typed indices, and side data. It buys cache locality and allocation-free node creation. Preorder makes each same-kind subtree contiguous; cross-kind content such as a paragraph's inlines or a note's blocks remains an explicit range and must be copied or shared deliberately.

The second thing a converter accumulates is decisions that are invisible in the code. When a reader emits nothing for a construct, nothing in the source distinguishes "this does not survive the projection" from "we have not written this yet". ZDS 0001 requires format records to carry an explicit omissions section for exactly this reason, and this record establishes the pattern.

13.1. Relationship to other records

ZDS 0001 defines the process and the format-record obligations. This record is the architectural parent of every future format record: a plugin ZDS states its mapping table, its omissions, and its round-trip expectations, and inherits everything here about the AST, the contract, and the limits. A change to the node set itself amends this record and requires its own.

ZDS 0013, *Layered Document IR and Writer Lowering*, supersedes this record's sections on the document AST, the builder, the rebuild transform, and the writer's lowering decisions with the IR v2 design: a semantic kernel with sparse facets, extension nodes, and a deterministic lowering planner. The replacement was a clean break: zenfmt was pre-release with no external consumers, so IR v2 replaced IR v1 in place with no staging or compatibility layer. That rewrite has landed. The superseded sections here are historical; ZDS 0013 is descriptive of the shipped system, and everything outside the superseded sections stays normative.

14. Terminology and Scope

- **AST**: the abstract syntax tree defined under *The Document AST*. Two node arrays, a text pool, and side tables, all in one arena.
- **node**: a `Block` or an `Inline`, identified by a `u32` index into its array.
- **reader**: a plugin that parses one input format into the AST. A front end.
- **writer**: a plugin that serializes the AST into one output format. A back end.
- **filter**: a transform from AST to AST, written in Zig, that a user composes into a pipeline.
- **format library**: one shallow Zig module containing a format's reader, writer when present, helpers, preservation-data schema, and tests.
- **plugin**: a comptime-validated reader, writer, or filter descriptor exported by a format or filter library.
- **artifact manifest**: the adjacent `<output>.zenfmt.json` file containing provenance, semantic document metadata, reports, and namespaced plugin data.
- **plugin data**: versioned JSON owned by a stable plugin id. The engine stores, validates, and carries it but does not interpret it.
- **bundle**: comptime tuples of plugin descriptors. The default bundle powers the umbrella API and CLI; applications may define smaller bundles.
- **registry**: the validated tables generated from a bundle.
- **static router**: the engine's comptime-generated dispatch from a runtime format pair to a monomorphized pipeline.
- **report**: a user-facing diagnostic, rendered in the Elm style described under *Diagnostics and Error Messages*.
- **lossy**: a mapping that discards information present in the source. Every interesting mapping here is lossy in at least one direction.

In scope: the AST, the traversal and rewriting algorithms, the filter system, the plugin contract, the engine, the registry, the memory model, diagnostics, the artifact manifest, ingestion for the first-release formats, the Markdown writer, the coding standard, the CLI, the build, and the test strategy.

Out of scope: templating and standalone-document wrapping, citation processing, PDF output via a typesetter, and any writer other than Markdown in the first release. Each is a candidate for a later record; none is precluded.

15. Problem Statement

A converter has six hard parts, and only the first is a parsing problem.

The tree must serve formats that disagree about structure. DOCX has no list element: a list is a run of paragraphs carrying numbering properties that point into a shared numbering definition. HTML has an explicit list element and permits arbitrary nesting inside items. AsciiDoc and reStructuredText have block-level directives with no HTML analogue. CSV has no inline structure at all. Markdown distinguishes tight and loose lists, a distinction DOCX cannot express. A tree shaped like any one of these makes the others expensive.

Transformation must be possible without forking the program. A converter is substantially more useful when a user can write a filter. Converting a manual usually needs one or two document-specific adjustments — shift every heading down a level, rewrite internal links, drop a class of admonition — and a converter that cannot express those is a converter people work around. This requires a real tree with node identity, and an API for walking and rewriting it.

Fidelity is not achievable and must therefore be specified. Every conversion discards something; the question is whether the user is told. A converter that silently drops the tracked-changes history, the comment threads, and the embedded chart while emitting clean-looking Markdown has produced a plausible lie.

Useful source detail must survive a lossy target when possible. Some data has no honest place in the shared AST but is valuable to the plugin that understands it: a DOCX style id, an OOXML relationship, or an application-specific field. It must travel beside the converted artifact under a versioned plugin namespace, with a digest binding it to the exact output, rather than leaking format fields into the core AST or disappearing forever.

The input is untrusted by construction. The program's entire purpose is to open a file someone else produced. DOCX and ODT are ZIP archives supporting unbounded expansion ratios; XML supports quadratic entity expansion; archive entry names can contain `..`. Resource limits are a design surface, not an implementation detail.

Extension must not touch the core. A converter's value is roughly the product of its input and output counts, so the cost of adding a format sets the project's ceiling. If adding DOCX means editing the engine, the engine accumulates format knowledge until it cannot be reasoned about.

16. Goals and Non-Goals

16.1. Goals

1. **A real, Zig-native AST.** Blocks nest, inlines nest, metadata supports JSON scalars plus rich inline and block values, and every node may carry `Attrs`. Public node views are `union(enum)` values; storage remains compact SoA. Comptime schema functions generate typed payload access, validation, and exhaustive visitor dispatch.
2. **Filters in Zig, in the `build.zig` manner.** A user writes a Zig file declaring a pipeline, compiles it against the zenfmt module, and gets a binary with their transforms in it. No embedded interpreter, no serialization boundary, no dynamic loading.
3. **Flat, data-oriented storage.** Struct-of-arrays, typed `u32` indices, one arena per conversion, no allocation per node, and contiguous same-kind node ranges.
4. **Algorithmic efficiency that follows from the storage**, not from micro-optimization: `O(1)` subtree skip, vectorizable tag scans, bulk range copies during rewrites, and a single linear pass for the common traversal.
5. **A core with no format knowledge.** No identifier from any file specification appears in the engine or the AST. Adding a format creates one library and adds one import to whichever bundle should ship it.

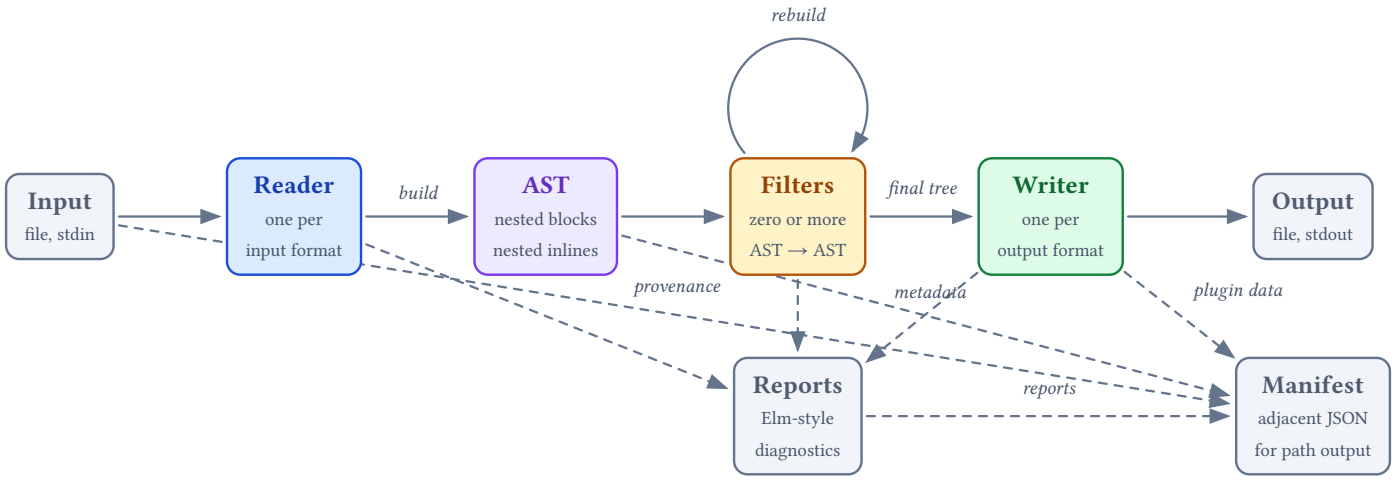
6. **Explicit lossiness.** Every discarded construct is recorded in a report at runtime and in an omissions table in the plugin's ZDS.
7. **Durable conversion context.** Every path output receives an adjacent, deterministic JSON manifest. A verified manifest is available to later readers and writers; unknown plugin namespaces survive unchanged.
8. **A small public surface.** The common CLI is one command and the common Zig API is one function call. Storage, routing, manifests, and atomic commit are automatic. Plugin authors use typed views and emitters; raw arrays and internal indices stay private.
9. **Error messages in the Elm tradition.** A diagnostic names what happened, shows where it happened, states what zenfmt did as a result, and gives at least one concrete next action — never a bare code or unactionable hint.
10. **Bounded resource use on hostile input.** Every limit has a name, a default, a CLI override, and a reason recorded here.
11. **Deterministic output.** The same input bytes produce the same output bytes, on every platform and in every build mode.
12. **Maintainability as a stated constraint**, enforced by the coding standard: bounded everything, no recursion, files under 1,000 lines, and flat source within each library boundary.

16.2. Non-Goals

- **Byte-level round-trip fidelity.** DOCX to Markdown and back will not reproduce the original. This follows from having one shared tree.
- **Maximizing format count.** zenfmt aims at a well-understood set done thoroughly.
- **An embedded scripting language.** Filters are Zig. A zenfmt AST protocol for external filters is a possible later feature, not a first-release goal.
- **Layout preservation.** Pagination, columns, absolute positioning, and typography are not represented.
- **Rendering.** No output format requires a typesetter or a font stack.

17. Design Overview

A conversion is a pipeline with a tree in the middle and an arbitrary number of transforms on it.



The engine owns the arena, the report sink, and the limits. It hands the reader an `Emitter` and the input; the private builder freezes the result into a `Document`; it runs the filter pipeline, each stage producing a new `Document` in the same arena; it hands the last one to the writer. For a path output it then commits the output and its manifest in the same directory. It never interprets

a format-specific node or plugin-data namespace; generic orchestration necessarily knows the selected registry entries, hashes, reports, and semantic metadata needed to produce the manifest.

Two properties follow from the AST sitting between the halves. Each reader and each writer is compiled once regardless of how many formats exist, so the conversion matrix does not multiply code size — only the small dispatch glue is quadratic. And a reader can be tested against the AST directly, with no writer involved, which is what the tree-test suite does.

18. The Document AST

18.1. Shape: a Zig-native document model

Prior converter designs supply two useful ideas: separate block and inline trees, and filters over a shared representation. The node taxonomy below belongs to zenfmt. It is smaller, uses names natural to this API, permits attributes uniformly, and normalizes repeated container structures for efficient walking.

```
pub const BlockTag = enum(u8) {
    plain,
    paragraph,
    line_block,
    heading,
    code_block,
    raw_block,
    quote,
    list,
    definition_list,
    thematic_break,
    table,
    figure,
    container,

    // Private normalized structure; never returned as a top-level API kind.
    line,
    list_item,
    definition_entry,
    definition_term,
    definition_body,
    caption,
    table_head,
    table_body,
    table_foot,
    table_row,
    table_cell,
};

pub const InlineTag = enum(u8) {
    text,
    space,
    soft_break,
    hard_break,
    emphasis,
    underline,
    strong,
    strikethrough,
    superscript,
```

```
    subscript,
    small_caps,
    quote,
    code,
    math,
    raw,
    link,
    image,
    note,
    span,
    citation,
};

pub const MetaValueTag = enum(u8) {
    null,
    boolean,
    integer,
    float,
    string,
    inlines,
    blocks,
    map,
    list,
};
```

Public inspection uses a tagged union, the Zig equivalent of a closed sum type:

```
pub const BlockView = struct {
    attrs: AttrsIndex,
    content: union(BlockTag) {
        plain: InlineRange,
        paragraph: InlineRange,
        line_block: LineRange,
        heading: Heading,
        code_block: Literal,
        raw_block: Raw,
        quote: BlockRange,
        list: List,
        definition_list: DefinitionRange,
        thematic_break: void,
        table: Table,
        figure: Figure,
        container: Container,
        // Private structural cases are omitted from the public iterator.
        line: InlineRange,
        list_item: BlockRange,
        definition_entry: DefinitionEntry,
        definition_term: InlineRange,
        definition_body: BlockRange,
        caption: Caption,
        table_head: TableSection,
        table_body: TableBody,
        table_foot: TableSection,
        table_row: TableRow,
        table_cell: TableCell,
    },
};
```

```zig
pub fn BlockPayload(comptime tag: BlockTag) type {
    const Content = @FieldType(BlockView, "content");
    return @FieldType(Content, @tagName(tag));
}
```

`Document.block(id)` returns `BlockView`; a normal exhaustive `switch` handles it. Code that already knows the tag calls `document.blockAs(id, comptime tag)` and receives `?BlockPayload(tag)`. Inline views follow the same pattern. Raw storage fields and payload-table indices are not public API.

One comptime schema module maps each tag to its payload type, child kind, and allowed placement. As implemented, it is a set of coordinated exhaustive switch functions in `core/src/payload.zig` rather than a single generating table: adding a tag produces compile errors in every switch and in every exhaustive writer, so **coverage** cannot drift, but **agreement** between the functions (that the content rule and the validator assert the same child kind for a tag) is maintained by hand. ZDS 0013 replaces the coordinated switches with one comptime table from which accessors, placement predicates, validator cases, visitor dispatch, and debug names are all derived.

Three points about this list are deliberate.

`space`, `soft_break`, and `hard_break` are separate leaf nodes rather than characters inside a text run. This costs nodes and buys the ability for a filter to distinguish a line wrap in the source from a deliberate break, and for a writer to re-wrap text without guessing.

`container` and `span` exist so a format's structure that has no direct node — an AsciiDoc admonition, an HTML `section`, a DOCX content control, an RST directive — survives as an attributed container rather than being flattened. They are the extension point that keeps the node set closed, they are what most filters match on, and they are in the set from the start rather than added when the first such format arrives.

`emphasis` and `strong` are containers, not flags. This reverses an earlier draft; the reasoning is under *Why the tree is a tree*. `citation` carries typed references, prefixes, suffixes, and modes rather than only an id. `line_block` preserves its list-of-inline-lists shape through storage-only `line` nodes.

The internal structural tags do not leak through the semantic API. For example, `BlockView.list` returns list properties and an iterator of item block ranges; it does not expose storage indices. The same rule applies to definition entries and table sections.

18.2. Storage: flat, preorder, struct-of-arrays

The conversion owns one append-only `Store`. Its block and inline `std.MultiArrayList` values hold nodes in preorder. Each node records its subtree **length**, rather than an absolute end index, so a same-kind subtree can be bulk-copied without rebasing internal edges. A `Document` is a cheap snapshot: ranges and metadata roots into that store. Earlier snapshots remain valid while filters append later ones.

```zig
pub const BlockIndex = enum(u32) { _ };
pub const InlineIndex = enum(u32) { _ };
pub const AttrsIndex = enum(u32) { _ };
pub const NodeIndex = union(enum) {
    block: BlockIndex,
    // `inline` is a Zig keyword; the escaped identifier is the field name.
    @"inline": InlineIndex,
```

```
};

pub const OptionalAttrsIndex = enum(u32) {
    none = std.math.maxInt(u32),
    _,
};

pub fn Range(comptime Index: type) type {
    return struct {
        start: Index,
        len: u32,

        pub const empty: @This() = .{ .start = @enumFromInt(0), .len = 0 };
    };
}

pub const BlockRange = Range(BlockIndex);
pub const InlineRange = Range(InlineIndex);
pub const ByteRange = struct {
    start: u32,
    len: u32,
};

pub const Block = struct {
    tag: BlockTag,
    /// Index into the tag's typed payload table. Accessed only through a
    /// tag-specific view such as `heading()` or `table()`.
    payload: u32,
    attrs: OptionalAttrsIndex,
    inlines: InlineRange,
    /// Includes this node. A leaf has length 1.
    subtree_len: u32,
};

pub const Inline = struct {
    tag: InlineTag,
    payload: u32,
    attrs: OptionalAttrsIndex,
    subtree_len: u32,
};

pub const Store = struct {
    blocks: std.MultiArrayList(Block) = .empty,
    inlines: std.MultiArrayList(Inline) = .empty,
    text: std.ArrayList(u8) = .empty,
    attrs: std.ArrayList(Attrs) = .empty,
    // Typed payload tables, metadata values, citations, strings, pairs,
    // column specs, and plugin data follow the same append-only rule.
};

pub const Document = struct {
    store: *const Store,
    body: BlockRange,
    meta: MetaMapIndex,
    plugin_data: PluginDataRange,
};
```

Children of node `i` are found by hopping: start at `i + 1`, and from each child `j` add `subtree_len[j]`, stopping at `i + subtree_len[i]`. There is no child list to allocate and no parent pointer to maintain.

```
pub const Children = struct {
    lengths: []const u32,
    cursor: u32,
    bound: u32,

    pub fn init(doc: *const Document, parent: u32) Children {
        const lengths = doc.store.blocks.items(.subtree_len);
        assert(parent < lengths.len);
        return .{
            .lengths = lengths,
            .cursor = parent + 1,
            .bound = parent + lengths[parent],
        };
    }

    pub fn next(it: *Children) ?u32 {
        assert(it.cursor <= it.bound);
        if (it.cursor == it.bound) return null;
        const index = it.cursor;
        it.cursor += it.lengths[index];
        assert(it.cursor > index);
        return index;
    }
};
```

The three properties this layout is chosen for:

Property	Consequence
A same-kind subtree is a contiguous range	A changed ancestor can bulk-copy an unchanged block or inline branch one SoA column at a time. Relative `subtree_len` values need no rebasing. Cross-kind ranges and append-only side-table indices remain stable.
`subtree_len` skips a subtree in `O(1)`	A filter that only cares about `link` nodes advances past whole untouched branches without visiting them.
Struct-of-arrays puts every tag in one byte column	`std.mem.indexOfScalar` over `blocks.items(.tag)` is a contiguous byte scan the compiler vectorizes. "Find every table in this document" is a `memchr`, not a tree walk.

18.3. Attributes

Every block and inline may carry `Attrs`: an identifier, a class list, and ordered key-value pairs. Uniform attributes cost one optional index column and remove tag-specific exceptions from filters and format mappings.

```
pub const Attrs = struct {
    id: ByteRange,        // into text; empty when absent
    classes: StringRange,
    pairs: PairRange,
};
```

Attributes carry heading identifiers, the `code_block` language as the first class, `container` and `span` roles, and everything a format has that the node set does not name. They are also how filters select: matching a class is the idiomatic way to write a transform that applies to some nodes and not others.

The `payload` column is not a packed bag of bits. It is an index into a typed side table selected by the tag, reached through an exhaustive accessor. This costs a few small tables and prevents invalid states such as a list-number style being interpreted as a quote type. The private `u32` is an implementation detail; public APIs return these types:

Tag	Typed payload
`heading`	`Heading { level: u8 }`, level 1 to 6.
`list`	`List { kind: ListKind, start: i64, style: NumberStyle, delimiter: NumberDelimiter, items: BlockRange }`. Unordered lists ignore the numeric fields. `plain` versus `paragraph` inside items conveys tight rendering.
`code_block`, `code`	`Literal { text: ByteRange }`; attributes live in the node's `attrs` column.
`raw_block`, `raw`	`Raw { format: StringIndex, text: ByteRange }`.
`table`	`Table { caption: CaptionIndex, columns: ColumnRange }`.
`table_body`	`TableBody { row_head_columns: u32, head_rows: u32 }`.
`table_cell`	`TableCell { alignment: Alignment, row_span: u32, col_span: u32, blocks: BlockRange }`.
`figure`	`Figure { caption: CaptionIndex }`.
`quote`	`QuoteKind`: single or double.
`math`	`Math { kind: MathType, text: ByteRange }`.
`citation`	`CitationRange`; each citation carries id, prefix, suffix, and presentation mode.
`link`, `image`	`Target { url: ByteRange, title: ByteRange }`.
`note`	`BlockRange`.

18.4. The text pool and side tables

All text lives in one growable byte array; a `Range` into it is 8 bytes and never dangles. Text is stored **decoded**: XML character references, RTF hex escapes, and HTML entities are resolved by the reader, so no writer ever sees a source format's escaping. Text is UTF-8, and readers of other encodings transcode on the way in.

Repeated strings — class names, language tags, attribute keys — are interned through a hash map that lives for the conversion and is dropped with the arena. Interning is selective: prose is appended once and never hashed; only the small, frequently repeated vocabulary enters the map.

```
pub const Link = struct {
    target: ByteRange,
    title: ByteRange,
};

pub const Media = struct {
    source: ByteRange,   // path, URL, or archive entry name
    bytes: ByteRange,    // extracted content; empty in the first release
    mime: ByteRange,
};

pub const MetaEntry = struct {
    key: ByteRange,
    value: MetaValueIndex,
};

pub const MetaValue = struct {
    tag: MetaValueTag,
    /// Index into the table selected by `tag`: map entries, value indices,
    /// bools, strings, inline ranges, or block ranges.
    payload: u32,
};
```

Metadata is a zenfmt tagged value model: null, boolean, signed integer, finite float, string, inlines, blocks, list, and map. It deliberately includes JSON's scalar vocabulary rather than forcing counts and numeric properties into text. Maps serialize with keys in bytewise UTF-8 order for deterministic output; their semantic order is insignificant. JSON scalars, lists, and maps use their natural representation in the manifest. Rich inline and block values use a zenfmt-owned tagged object. Validators and filters traverse rich metadata as part of the document rather than treating it as an untyped header bag.

18.5. Invariants

The following hold for every `Document` handed to a filter or a writer. They are checked by `ast.validate`, which runs on every conversion in `Debug` and `ReleaseSafe`, after every filter stage, in every test, and on every fuzz iteration. The validator is the oracle that makes reader and filter fuzzing meaningful: a plugin that produces a structurally impossible tree fails even when nothing crashes.

1. `subtree_len[i] >= 1` and `i + subtree_len[i] <= len`, in both arrays.
2. For any descendant `j`, `j + subtree_len[j] <= i + subtree_len[i]`. Subtrees nest properly.
3. Every `Range` lies within the array it indexes.
4. Container tags hold `inlines == InlineRange.empty`; leaf-content block tags have `subtree_len == 1`.
5. Child tags appear only under their parent tag: `list_item` under a list, `table_row` under a table section, `table_cell` under `table_row`, `definition_term` and `definition_body` under `definition_entry`.
6. Empty inline containers are valid; writers must not assume styled content is non-empty.
7. Every `note` payload names a valid block forest. Metadata block and inline roots are valid, acyclic, and subject to the same depth limit as the body.
8. Depth does not exceed `limits.max_depth` in either tree.
9. `attrs` is either `.none` or a valid `AttrsIndex` for every node; the payload accessor selected by every tag succeeds.

10. The text pool is valid UTF-8, and no `text` node contains a whitespace character — whitespace is a `space`, `soft_break`, or `hard_break` node.
11. No two adjacent siblings are both `text`; readers coalesce.

18.6. A worked example

The Markdown fragment

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

- one
- two

  still two
```

produces the two trees below. Indentation shows the nesting that `subtree_len` encodes; it is not stored.

i	block	payload	subtree len	inlines
0	`heading`	level 2	1	0..9
1	`list`	`unordered`	6	—
2	`list_item`	—	2	—
3	`plain`	—	1	9..10
4	`list_item`	—	3	—
5	`paragraph`	—	1	10..11
6	`paragraph`	—	1	11..12

Block 4′s `subtree_len` is 3 while block 2′s is 2: the second item holds two paragraphs, the first holds one, and both are found by the same hop. Item one's child is `plain` and item two's are `paragraph` — the tight-versus-loose distinction a writer needs in order to reproduce blank lines faithfully.

j	inline	payload	subtree len
0	`text`	`"The"`	1
1	`space`	—	1
2	`emphasis`	—	2
3	`text`	`"quick"`	1
4	`space`	—	1
5	`link`	`targets[0]`	2
6	`text`	`"brown"`	1
7	`space`	—	1
8	`text`	`"fox"`	1

The heading's `inlines` range is 0..9. Its seven top-level children are the ones reachable by hopping from 0, which skips the children of `emphasis` and `link` without a test for what they are.

18.7. Why the tree is a tree

An earlier draft of this record proposed a flat inline layer: a sequence of text runs each carrying a style bitset, with no inline nesting. The argument was that the office formats contain no inline nesting to preserve — DOCX runs carry independent property flags, and so do ODT spans and RTF groups — so building a tree from flags means inventing structure the source never specified.

That argument is right about the readers and wrong about the system, for three reasons.

Filters need node identity. A transform that wraps every `code` inline in a `link` to an API reference has to put a node around another node. A bitset can be set and cleared; it cannot be wrapped. Every interesting filter — unwrap this container, wrap that one, replace a node with a subtree — is a structural edit, and a flat run layer offers nothing to edit. Since filters are a first requirement rather than a later addition, this alone settles it.

Not every inline container is a style. `link`, `image`, `note`, `span`, and `citation` carry payloads and children that no flag set can express. Once the representation must nest for those, making `emphasis` and `strong` different in kind buys nothing and costs a special case in every walker.

One structural model is easier to teach and generate code for. When all inline containers follow the same nesting and visitor rules, the comptime schema can generate one small API instead of accumulating style-specific special cases.

What the flat-run argument does establish is a requirement on readers, and it is kept: **a reader converting a flag set to nesting must use a canonical order**, so bold-italic text produces the same tree no matter which flag the source listed first. The order is `link`, `strong`, `emphasis`, `strikethrough`, `superscript`, `subscript`, `small_caps`, `underline`, outermost first. It is stated once, here, and asserted by a shared test that every flag-based reader runs.

18.8. The Builder

`Builder` is private. The public `Emitter` delegates to it after type and balance checks. The builder borrows the conversion's store and owns the allocator, interning table, and bounded open-node stack.

```
pub const Builder = struct {
    gpa: std.mem.Allocator,
    store: *Store,
    // Snapshot starts, the intern map, and bounded open-node stacks.

    /// Opens a container block; must be matched by `closeBlock`.
    pub fn openBlock(b: *Builder, tag: BlockTag, attrs: ?Attrs) !BlockIndex;
    pub fn closeBlock(b: *Builder) void;
    pub fn leafBlock(b: *Builder, tag: BlockTag, attrs: ?Attrs) !BlockIndex;

    /// Opens a container inline; must be matched by `closeInline`.
    pub fn openInline(b: *Builder, tag: InlineTag, attrs: ?Attrs) !InlineIndex;
    pub fn closeInline(b: *Builder) void;

    /// Appends text, splitting on whitespace into `text` and `space` nodes and
    /// coalescing with a preceding `text`, so a reader appending arbitrary
    /// chunks still produces a canonical tree.
    pub fn text(b: *Builder, bytes: []const u8) !void;

    /// Bulk-copies each SoA column. Relative subtree lengths and append-only
    /// side-table indices require no fix-up.
    pub fn copyBlocks(b: *Builder, range: BlockRange) !BlockRange;
```

```
    pub fn copyInlines(b: *Builder, range: InlineRange) !InlineRange;

    /// Freezes into a `Document`. Asserts the open stack is empty.
    pub fn finish(b: *Builder, meta: MetaMapIndex) Document;
};
```

The concrete implementation uses Zig 0.16's unmanaged `std.ArrayList` and `std.MultiArrayList` forms. Allocation is explicit inside `Builder` through its stored `std.mem.Allocator`.

`text` is the one place the `Builder` is opinionated, and deliberately: the whitespace and coalescing invariants are easier to hold in one function than in nine readers.

`finish` asserts the open stack is empty. A plugin that returns unbalanced is a bug, not a recoverable condition, and it is caught in that plugin's own tests rather than papered over by an auto-close.

19. Traversal and Rewriting Algorithms

The storage was chosen for these. This section states what it buys and what each operation costs.

19.1. Bounded, non-recursive walking

No traversal in zenfmt recurses. Every walker keeps an explicit stack of open node indices bounded by `limits.max_depth`, checked on push. This follows the coding standard's prohibition on recursion, and it is a safety rule before it is a style rule: nesting depth is attacker-controlled, and a bounded stack turns a stack overflow into a report.

The common traversal does not need the stack at all. Emitting a document in document order is a single forward scan over the array, because preorder storage **is** document order. The stack is needed only for writers that must emit something on the way out of a node, and those keep one entry per open container.

19.2. Skipping and scanning

Two operations a pointer-based tree cannot do cheaply:

Subtree skip. A walker that decides a node is uninteresting sets `i += subtree_len[i]` and continues. The entire branch costs nothing — not one cache line is touched.

Tag scan. Because storage is struct-of-arrays, `blocks.items(.tag)` is a contiguous `[]BlockTag`, one byte per node. Finding every table in a document is `std.mem.indexOfScalar` over that column: a vectorized byte scan at memory bandwidth, with no pointer chasing and no node loads. Filters that match a single tag are implemented this way, and the pipeline recognizes them — a filter declaring `pub const matches: []const BlockTag = &.{.table}` gets the scan path rather than the walk path.

19.3. The rebuild transform, with subtree sharing

Filters do not mutate. A stage reads one `Document` and appends a replacement snapshot to the same store only when it finds an edit. A discovery pass records edits in source order; if the edit list is empty, the stage returns the input `Document` unchanged and allocates no AST storage. A rebuild pass then copies only the changed spines. Unchanged sibling subtrees are copied a column at a time, while referenced inline ranges and side-table values are reused because their append-only indices remain stable.

This two-pass form is intentional. A one-pass post-order builder has already copied children by the time it learns that their parent is unchanged. Discovery before emission makes the identity

case zero-copy and keeps rollback trivial: on error, truncate each append-only table to the stage's saved lengths.

The consequences, stated as costs, since "fast" is not a design:

Case	Cost	Note
A filter that matches nothing	`O(n)` tag bytes, zero AST allocation	A declared tag set is scanned before callbacks. If no candidate exists, the original snapshot is returned.
Candidates exist but callbacks keep all	`O(v)`, zero AST allocation	`v` is the visited candidate set plus required ancestors. Discovery records no edits, so rebuild does not run.
A filter that rewrites `k` nodes	`O(v + k · d + m)`	`d` is depth and `m` is the total same-kind node data copied or emitted for changed spines. Unchanged inline ranges and side data are shared.
A tag-scan filter	`O(n)` bytes + the above	The scan reads one byte per node, not twenty-one.
A pipeline of `s` filters	`s` passes	Only stages with edits append a snapshot. Nothing is individually freed; the arena is re-

Case	Cost	Note
		leased once at the end.

Bulk copying is permitted only for ranges whose relative `subtree_len` values are self-contained. Cross-array references are not assumed contiguous: a note may refer to blocks elsewhere, and a block may reuse an inline range. Those indices point into the append-only store and are shared. This corrects the stronger but false claim that every logical subtree is one self-contained slice in both arrays.

19.4. Complexity summary

Operation	Cost	Why
Read a node	`O(1)`	Array index.
Iterate children	`O(children)`	Add `subtree_len`.
Skip a subtree	`O(1)`	Add `subtree_len`.
Find all nodes of a tag	`O(n)` bytes	Vectorized scan of the tag column.
Copy a same-kind subtree	`O(size)` bulk copy	One copy per SoA column.
Full traversal	`O(n)`	Linear scan; preorder is document order.
Filter stage, `k` edits	`O(v + k · d + m)`	Discovery plus spine rebuild.
Depth of any operation	`≤ max_depth`	Checked, not assumed.

19.5. Staged parsing and streaming rendering

The default parser architecture combines ideas that have proved useful in modern high-throughput parsers without requiring every format to pretend it is JSON. It has two stages:

1. A scanner validates encoding and locates format-specific structural bytes in bounded chunks. For XML these include `<`, `>`, `&`, quotes, and line ends; for Markdown they include line ends and delimiter candidates; for CSV they include separators, quotes, and line ends. The scanner carries lexical state across chunks, so a quote or multibyte code point at a boundary is ordinary.
2. A scalar state machine consumes those offsets, validates grammar, resolves names, and emits directly through `Emitter`. It creates no token object per lexeme and never constructs a source-format DOM.

This is the structural-index pattern demonstrated by simdjson and SIMD XML work, adapted to a streaming document converter. The index is a reusable `std.ArrayList(u32)` scratch buffer capped by `limits.scan_chunk_bytes`; it is drained before the next chunk, so worst-case punctuation does not allocate an input-sized second copy. UTF-8 validation is fused with scanning for UTF-8 formats. UTF-16 input is validated and transcoded in bounded chunks before the same scalar stage.

There are scalar and vector scanner implementations with identical differential tests. The vector path uses Zig 0.16 `@Vector` operations and compile-time target feature selection, with no architecture-specific assembly in plugins. It is enabled for a format only when corpus benchmarks show at least a 15% throughput gain at the same peak-memory bound on both a supported x86-64 and AArch64 machine. Small inputs use the scalar path below a measured crossover size. Branchless or SIMD code is not accepted on aesthetics alone.

The grammar stage remains format-specific:

- CommonMark follows the reference two-phase strategy: block parsing first, collecting link-reference definitions, then inline parsing with bounded bracket and delimiter stacks. Regular-expression substitution is not a conforming parser.
- XML is a pull parser over expanded names. Known element and attribute names dispatch through comptime-generated lookup tables; unknown names are skipped or reported according to the format ZDS. Entity decoding writes directly to the text pool.
- `text/html` follows the WHATWG tokenizer and tree-construction states. XML rules are not used for HTML, even when the input looks well formed.
- RTF uses one explicit group stack and CSV uses one quote-aware state machine. Neither receives a structural-index pass unless its own benchmarks justify one.

Writers are the reverse shape: one bounded stack plus a buffered `*std.Io.Writer`. They scan for bytes needing context-sensitive escaping, send ordinary spans with `writeAll`, and take the scalar slow path only at an escape or delimiter. Fence selection is a linear max-run scan. Writers never build the whole output in memory; path outputs stream to a temporary file in the target directory before commit. Parallel parsing or rendering is deferred: it adds ordering, memory, and cancellation costs, and is adopted only if end-to-end benchmarks on representative documents beat the single-threaded staged design.

20. Public API and Developer Experience

The public package has one high-level conversion function. It keeps allocator and I/O authority explicit, as idiomatic Zig 0.16 requires, while making format detection, limits, reports, manifest loading, hashing, and atomic path output defaults rather than chores:

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

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

if (conversion.status == .failed) {
    // Use the standard Elm-style renderer, or inspect conversion.reports.
    try conversion.renderReports(io, .stderr, .auto);
    return error.ConversionFailed;
}
```

The two path fields are enough for the common case. `ConvertOptions` also accepts byte slices and `*std.Io.Reader`/`*std.Io.Writer` for embedding, optional explicit formats, limits, and a filter pipeline. A stream output cannot have an adjacent file, so the returned `Conversion.manifest` gives the caller the same metadata value that path output writes automatically.

The top-level `zenfmt` module is an umbrella library containing the default format bundle. Applications that want a smaller binary import `zenfmt-core` and only the format libraries they need:

```
const Converter = zenfmt_core.Bundle(.{
    .readers = .{ zenfmt_docx.reader },
    .writers = .{ zenfmt_markdown.writer },
});
```

```
var conversion = Converter.convert(gpa, io, options);
defer conversion.deinit(gpa);
```

Both paths use the same `ConvertOptions`, `Conversion`, and plugin contracts. Selecting fewer formats changes linked code, not application semantics. `convert` returns a `Conversion` rather than an error union for every input, format, limit, filesystem, and allocation failure. A reserved inline report slot with static strings lets even `OutOfMemory` return `.status = .failed` without allocating. This prevents Zig's payload-free error values from discarding the explanation and directions at the exact moment an embedding application needs them. A failed result never commits an artifact.

The API follows four rules:

- No global allocator, filesystem handle, environment lookup, or process-wide registry is hidden behind convenience functions.
- Expected conversion failures return `.status = .failed` with one or more structured reports. Callers switch on status and stable report codes; they do not parse error strings.
- `Document`, `BlockView`, `InlineView`, `Attrs`, `Metadata`, `Report`, and `ArtifactManifest` are public. `Store`, `Builder` internals, payload indices, and archive/XML machinery are private.
- Defaults are useful and deterministic. Advanced options are grouped under `.limits`, `.reader`, `.writer`, and `.manifest`; the top-level options struct does not become a flat collection of every plugin knob.

20.1. Plugin authoring surface

A plugin exports one descriptor produced by a comptime constructor. The constructor validates ids, declarations, callback signatures, and option types where the plugin is defined, so registry errors point at the plugin rather than at router internals.

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

pub const plugin = zenfmt.Reader(.{
    .id = "ai.insan.zenfmt.docx",
    .format = "docx",
    .extensions = &.{"docx"},
    .input = .seekable,
    .read = read,
});

fn read(ctx: *zenfmt.ReadContext) zenfmt.ReadError!void {
    const block = try ctx.out.beginBlock(.paragraph, .{});
    defer ctx.out.endBlock(block);
    try ctx.out.text("Hello");
}
```

The emitter returns typed open tokens and `endBlock`/`endInline` require the matching token, making balanced `defer` the natural style. Convenience methods cover text, paragraphs, headings, links, lists, and reports; the generic typed methods cover the rest. Plugins inspect `BlockView` and `InlineView` tagged unions and never read a raw payload number.

Reader, writer, and filter descriptors share the same naming and option conventions. The registry is an array of descriptors, not a parallel set of switches. Public symbols are kept small enough to understand from generated API documentation without first reading the architecture record.

21. Filters

21.1. The `build.zig` analogy, taken literally

Zig's build system is configured by a Zig file in the user's project that imports `std.Build`, declares a graph through a typed API, and is compiled and run by the toolchain. It is not a configuration format and not a scripting language; it is a program, with a compiler checking it.

zenfmt filters work the same way. A user writes a Zig file that imports the `zenfmt` module, declares a pipeline through a typed API, and compiles it into a binary:

```
// filters.zig, in the user's own project
const std = @import("std");
const zenfmt = @import("zenfmt");

pub fn filters(p: *zenfmt.Pipeline) void {
    // One that ships with zenfmt.
    p.add(zenfmt.filters.shift_headings, .{ .by = 1 });

    // One the user wrote, in this file or beside it.
    p.add(InternalLinks, .{ .base = "https://docs.example.com/" });

    // Order matters, and is exactly the order written here.
    p.add(zenfmt.filters.drop_empty_containers, .{});
}

const Options = struct {
    base: []const u8,
};

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

fn visitInline(
    options: *const Options,
    ctx: *zenfmt.FilterContext,
    node: zenfmt.InlineIndex,
) !zenfmt.FilterAction {
    const link = ctx.document.inlineAs(node, .link).?;
    const target = ctx.document.text(link.target);
    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;
}
```

```
zig build           # produces a zenfmt binary carrying these filters
zenfmt --filters manual.docx -o manual.md
```

Everything a filter does is type-checked, there is no serialization boundary, and a filter costs a direct call — the pipeline is an `inline for` over the registered stages. The tradeoff is stated plainly

under *Alternatives Considered*: this buys speed and safety at the cost of requiring a Zig toolchain to write a filter.

21.2. The filter contract

A filter descriptor declares its option type, candidate tags, order, and one or both visit callbacks. `zenfmt.Filter` validates that declaration at comptime.

```
pub const FilterOrder = enum {
    bottom_up,
    top_down,
};

pub const FilterAction = enum {
    keep,
    drop,
    unwrap,
    replace,
};

pub const FilterContext = struct {
    gpa: std.mem.Allocator,
    document: *const Document,
    out: *Emitter,
    reports: *Reports,
    limits: Limits,

    pub fn block(ctx: *FilterContext, node: BlockIndex) BlockView;
    pub fn inlineView(ctx: *FilterContext, node: InlineIndex) InlineView;
    pub fn textOf(ctx: *FilterContext, node: InlineIndex) []const u8;
    pub fn hasClass(ctx: *FilterContext, node: NodeIndex, class: []const u8)
bool;
    pub fn attribute(
        ctx: *FilterContext,
        node: NodeIndex,
        key: []const u8,
    ) ?[]const u8;

    pub fn replaceLinkTarget(
        ctx: *FilterContext,
        node: InlineIndex,
        target: []const u8,
    ) !void;

    pub fn fmt(
        ctx: *FilterContext,
        comptime format: []const u8,
        args: anytype,
    ) ![]const u8;

    pub fn report(ctx: *FilterContext, value: Report) !void;
};
```

The contract:

- **A filter may assume** the input tree satisfies every invariant, that `document` does not change while the stage runs, and that the arena outlives the call.

- **A filter must** leave the emitter balanced, respect `limits`, and be deterministic — the same input tree must produce the same output tree.
- **A filter must not** write to the output stream, read the filesystem or the network, depend on the input or output format, or retain an index from `in` past the end of its stage.
- **The engine guarantees** it validates the tree after every stage, so a filter producing a structurally invalid tree fails at its own stage rather than corrupting a later one.

The prohibition on format knowledge is the important one. A filter that behaves differently for DOCX input is a reader bug wearing a disguise; the whole point of the tree is that a transform written against it works for every format that can reach it.

21.3. Filters that ship

A small set, chosen because they are the transforms people actually reach for and because each exercises a different part of the contract:

Filter	What it does, and what it demonstrates
`shift-headings`	Adds a constant to every heading level, clamping at 1 and 6. The simplest possible stage: one tag, a typed heading-payload edit, `.replace`.
`promote-first-heading`	Moves a leading level-1 heading into document metadata as the title. Shows a filter that edits the document as a whole rather than node by node.
`drop-empty-containers`	Unwraps `container` and `span` nodes carrying no attributes. Demonstrates `.unwrap`, and is genuinely useful after HTML and DOCX ingestion.
`flatten-nested-tables`	Replaces a table inside a table cell with a placeholder and a report. The canonical lossy transform, and the one that shows how a filter cooperates with the diagnostics contract.
`strip-classes`	Removes classes matching a pattern. Shows attribute editing and the tag-scan fast path.

Each is under 150 lines, and they double as the worked examples in the book and as the tests for the filter machinery.

22. The Plugin Contract

22.1. Readers

A reader is a Zig file exporting one comptime-validated descriptor. Reader and writer halves for the same format use the same reverse-DNS id, so they share one manifest namespace.

```
pub const StylePair = struct {
    paragraph_id: []const u8,
    style_id: []const u8,
};

pub const Preservation = struct {
    paragraph_styles: []const StylePair,
};

pub const preservation = zenfmt.JsonData(Preservation, .{
```

```
    .version = 1,
});

pub const plugin = zenfmt.Reader(.{
    .id = "ai.insan.zenfmt.docx",
    .format = "docx",
    .extensions = &.{"docx"},
    // Container readers need the ZIP central directory.
    .input = .seekable,
    .plugin_data = preservation,
    .read = read,
});

pub const ReadContext = struct {
    gpa: std.mem.Allocator,
    io: std.Io,
    out: *Emitter,
    input: Input,
    reports: *Reports,
    /// Present only after the engine verifies the adjacent manifest digest.
    manifest_in: ?*const ArtifactManifest,
    plugin_data: *PluginData(Preservation),
    limits: Limits,
    options: Options,
};
```

- **A reader may assume** the emitter is empty, the conversion arena outlives the call, its declared input capability is available, and the engine validates the result.
- **A reader must** leave the builder balanced, respect `limits`, produce valid UTF-8, use the canonical nesting order when converting flag sets, and report every construct it recognized and chose not to represent.
- **A reader may** read only its own plugin namespace through the typed `PluginData(Preservation)` API and replace that namespace with its current schema version. `get()` returns `?*const Preservation`; `replace(value)` accepts only `Preservation`. The engine carries all other namespaces without exposing them to the reader.
- **A reader must not** write to the output stream, read a file it was not given, resolve a network reference, or call `std.debug.print`.
- **The engine guarantees** it frees everything the reader allocated, and that a reader error aborts before any output byte is written — a failed conversion never leaves a half-written file.

22.2. Writers

```
pub const plugin = zenfmt.Writer(.{
    .id = "ai.insan.zenfmt.docx",
    .format = "docx",
    .extensions = &.{"docx"},
    .plugin_data = preservation,
    .write = write,
});

pub const WriteContext = struct {
    gpa: std.mem.Allocator,
    doc: *const Document,
    out: *std.Io.Writer,
    reports: *Reports,
```

```
    plugin_data: *PluginData(Preservation),
    limits: Limits,
    options: Options,
};
```

- **A writer may assume** every invariant holds. It does not defensively check structure; a violation is an engine bug the validator catches upstream.
- **A writer must** handle every tag — the switch is exhaustive with no `else`, so adding a node to the AST is a compile error in every writer rather than a silent omission at runtime. This is the main reason the tags are a closed enum.
- **A writer must** report any construct it cannot express, and produce byte-identical output for identical input.
- **A writer may** read and update only the namespace matching its `plugin_id`. A DOCX writer can therefore consume `ai.insan.zenfmt.docx` data originally saved by a DOCX reader and carried through an intermediate Markdown file. Unknown namespaces are preserved by the engine, not rewritten by plugins.
- **A writer must not** allocate per node or buffer the whole output.

Readers and writers are both explicit about loss. A reader reports every recognized construct it drops or degrades; a writer may not skip an AST tag at all. It may degrade a construct — a table cell holding two paragraphs may need flattening — but deliberately and with a report.

`*std.Io.Writer` is the one writer surface. Zig 0.16 already provides the type-erased buffered/vectored interface, keeping plugin signatures small and avoiding monomorphization per destination.

`PluginData(T)` is a descriptor-scoped capability, not a map. Its `get` and `replace` methods are statically typed as `T`, and it has no method for naming or opening another plugin's namespace. `JsonData` supplies the default canonical JSON codec for ordinary Zig structs. A format library provides explicit `decode`, `encode`, and `migrate` callbacks only when its data needs custom validation or an older schema upgrade. Reader and writer descriptors in the same library reference the same `preservation` declaration. Their contexts expose it as a typed optional value; missing, stale, and newer data all fall back to a correct AST-only conversion.

23. The Registry and the Static Router

23.1. One table

`src/default_bundle.zig` in the umbrella library names the plugins shipped by the standard CLI. The core `Bundle` constructor turns its descriptor tuples into all lookup tables. A handwritten switch answers one question; the CLI needs several — formats, extensions, help, detection, and suggestions — and they must not drift apart.

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

pub const Default = core.Bundle(.{
    .readers = .{ text.reader, markdown.reader, docx.reader },
    .writers = .{markdown.writer},
    .filters = .{
        core.filters.shift_headings,
        core.filters.drop_empty_containers,
```

```
    },
});
```

A comptime block in `Bundle` checks the descriptor tuples: no duplicate format or plugin id within one role, no extension claimed twice, and every id is valid reverse-DNS ASCII. Reader and writer rows for the same format must share a plugin id. These are the mistakes a table invites, and catching them in the compiler costs nothing.

Format names are lowercase ASCII strings owned by descriptors, not cases in a core `Format` enum. This is what lets `zenfmt_core` remain format-blind. The bundle validates names at comptime and the CLI resolves runtime strings against the generated tables. Raw AST nodes store an interned format-name string, not a registry ordinal that would change when an application chooses a smaller bundle.

23.2. Dispatching a runtime pair into a comptime matrix

The CLI learns its formats from `argv`, so a purely comptime `getReader` cannot serve it — the preliminary sketch's version raised `@compileError` for unregistered ids, which is right for a library caller and unusable for a tool. The bridge is an unrolled search:

```
fn convertRouted(
    e: *Engine,
    from: []const u8,
    to: []const u8,
    input: Input,
    out: *std.Io.Writer,
) Error!void {
    inline for (readers) |r| {
        if (std.mem.eql(u8, r.format, from)) {
            inline for (writers) |w| {
                if (std.mem.eql(u8, w.format, to)) {
                    return e.convertStatic(r, w, input, out);
                }
            }
            return error.UnsupportedOutputFormat;
        }
    }
    return error.UnsupportedInputFormat;
}

/// The zero-dispatch entry point, for callers who know both formats.
pub fn convertStatic(
    e: *Engine,
    comptime Reader: type,
    comptime Writer: type,
    input: Input,
    out: *std.Io.Writer,
) Error!void { ... }
```

Whether the matrix explodes the binary is a fair question, and the answer is no for a structural reason: `convertStatic` is `Reader.read`, then the filter pipeline, then `Writer.write`, with the AST between them. Each reader and each writer is instantiated once — `N + M` bodies — and only the call glue is `N × M`, at a handful of instructions per cell. The AST boundary buys this, and it is a second reason to have one.

23.3. Format taxonomy and roadmap

The registry grows along four families. Markdown is the default writer and the only one in the first release.

Family	Formats	Phase
Office	DOCX, XLSX, PPTX, RTF	4–5
OpenDocument	ODT, ODS, ODP	5
Lightweight markup	Markdown (CommonMark + GFM), AsciiDoc, reStructuredText, Org, Textile, MediaWiki	2, 6
Data and web	CSV, TSV, JSON, HTML, EPUB	2, 6

The lightweight markup family is where the AST earns its keep in a way the office formats do not exercise. AsciiDoc admonitions, reStructuredText directives, and Org drawers are all block-level constructs with no dedicated node; they map to `container` with a class, which is exactly what `container` is for, and they are the reason it is in the node set from the start rather than added when the first of them arrives.

23.4. Format detection

Resolved in this order, first match winning: an explicit `--from` or `--to`; the file extension against the registry's lists; content sniffing (`PK\x03\x04` followed by `[Content_Types].xml` for OOXML, whose specific format then comes from the content-type declarations; `{\rtf` for RTF; a `%PDF` header rejected with a clear message); Markdown for output with neither flag nor extension. Otherwise an error naming the formats that were available — never a silent fallback to plain text, which produces baffling output from a misdetected binary.

24. Adjacent Artifact Manifest

Document metadata and conversion provenance are part of the result, not log ephemera. When the output is a filesystem path, success therefore means two files exist in the same directory:

```
report.md
report.md.zenfmt.json
```

JSON is retained rather than ZON or a custom binary format. Plugin authors in other languages can inspect it, its scalar/list/map vocabulary fits zenfmt metadata, and unknown fields can be carried forward. The manifest uses RFC 8785 canonical JSON rules: UTF-8, no insignificant whitespace, object keys sorted as specified by the canonicalizer, and canonical number spelling. Binary payloads do not belong in JSON; media is a sibling file with a relative path and digest.

24.1. Version 1 envelope

The following is representative; digests are shortened only for the example.

```
{
  "schema":"ai.insan.zenfmt.artifact-manifest",
  "schema_version":1,
  "ast":{"schema":"ai.insan.zenfmt.ast","version":1},
  "source":{
    "name":"report.docx",
    "format":"docx",
    "digest":{"algorithm":"blake3-256","value":"6b7a..."},
    "plugin":{"id":"ai.insan.zenfmt.docx"}
  },
```

```
  "artifact":{
    "name":"report.md",
    "format":"markdown",
    "digest":{"algorithm":"blake3-256","value":"9d13..."},
    "plugin":{"id":"ai.insan.zenfmt.markdown"}
  },
  "document_metadata":{
    "title":{"$type":"inlines","value":[{"type":"text","text":"Report"}]}
  },
  "reports":[],
  "plugins":{
    "ai.insan.zenfmt.docx":{
      "version":1,
      "data":{"paragraph_styles":{"intro":"BodyText"}}
    }
  }
}
```

`document_metadata` uses zenfmt's metadata encoding and is the portable form of `Document.meta`; it is not an unrelated string map. Reports use their stable namespaced `code` as the machine id and include severity, problem, consequence, directions, count, source context when available, and loss tier. Absolute source paths, timestamps, hostnames, and allocator/build addresses are excluded so the same invocation is reproducible and the manifest does not leak workstation details.

The BLAKE3-256 digest is computed while streaming input and output, using Zig 0.16's `std.crypto.hash.Blake3`; it does not require a second file read. The digest is an integrity binding, not a signature. A future authenticity feature must add a signature field rather than treating an unkeyed hash as proof of origin.

24.2. Loading and carrying plugin data

For input `report.md`, the engine looks only for `report.md.zenfmt.json`. It parses the manifest under explicit byte, depth, string, and object-member limits, verifies the artifact format and BLAKE3 digest, and only then attaches its plugin namespaces to the new `Document`. A missing manifest is normal input. A malformed, oversized, or digest-mismatched manifest is ignored with a `STALE OR INVALID MANIFEST` warning. The report says that conversion continued without preservation data and directs the user to regenerate the artifact from its source, or to delete the stale sidecar if the artifact was intentionally edited. Plugin data is never applied to different bytes merely because filenames match.

Plugin namespaces obey these rules:

- The reverse-DNS key is the registry's stable `plugin_id`; it is never a file path or display name.
- Each value has an integer schema version. A plugin reads and replaces only its own namespace. The engine carries every other valid namespace value-for-value, including unknown fields, and re-encodes it canonically.
- Plugin data supplements the shared AST; it cannot change the AST's meaning. A writer must still produce a correct result when the namespace is absent, stale, or from a newer version.
- Paths in plugin data are relative to the artifact directory, cannot contain `..`, cannot be absolute, and are not opened unless the user explicitly enables the feature that needs them. Each referenced side asset carries a digest.
- Defaults are `max_manifest_bytes = 16 MiB`, `max_plugin_data_bytes = 4 MiB`, and `max_manifest_depth = 64`. Format ZDS records must document the fields their plugin owns and their retention and privacy implications.

This mechanism preserves useful fidelity without promising byte-level round trips. A DOCX reader may save style ids and relationship details under the DOCX namespace; Markdown cannot express them, but the Markdown artifact carries the namespace forward. A later DOCX writer may use that data when compatible and report when it cannot.

24.3. Commit and stream behavior

For a path output, the engine creates unpredictable temporary files in the target directory, streams and flushes the artifact, finalizes its digest, writes and flushes the manifest, then renames the artifact followed by the manifest. A successful command guarantees both final paths. A crash between renames can leave an artifact without a manifest, which readers already treat as normal; it cannot cause stale metadata to be trusted because digest verification is mandatory. Existing destinations are replaced only according to the CLI overwrite policy.

Standard output has no adjacent directory. Library conversion therefore always returns an `ArtifactManifest` value to its caller; the CLI requires `--metadata-out=PATH` when a user wants to persist it with `--stdout`. It never mixes the manifest into document stdout or diagnostics stderr. File output needs no flag: the adjacent manifest is mandatory and part of successful conversion.

25. Memory Model

One arena per conversion. Persistent conversion data — the append-only store, text pool, side tables, reports, and plugin data — comes from one `std.heap.ArenaAllocator` created by the engine and deinitialized when the conversion returns. Reusable scanner and I/O buffers are bounded and owned by the engine. Container elements have no individual destruction protocol.

This is not only a simplification. It removes use-after-free and double-free from the plugin surface entirely, which matters because plugins are the part of the system most likely to be written by someone who has not read this record: a reader cannot retain ownership accidentally, because it does not own anything. Filters that make no edits append no snapshot; edited snapshots remain valid until the conversion ends.

Growth. Arrays grow geometrically, and readers that can estimate their output call `ensureTotalCapacity` up front. The engine seeds the estimate from the input size using per-format ratios in the registry, so the common case is one allocation per array for the whole conversion.

Budgets.

Quantity	Target	Rationale
Allocations per conversion	`O(log n)`	Array growth only. Zero per node or per byte of text.
Base bytes per block node	21	The five SoA columns total 21 bytes before tag-specific payload-table data; this is asserted with `@sizeOf`-based tests, not assumed from a struct's padded size. Inline base nodes are smaller.
Peak memory, text formats	`≤ 5×` input	Text pool ≈ input, plus the node arrays. Measured, not assumed.

Quantity	Target	Rationale
Peak memory per edited filter stage	≤ 1× tree	Worst case when every node changes. No-op stages append no tree; typical sparse edits bulk-copy block spines and share inline and side data.
Scanner scratch	`O(scan_chunk_bytes)`	Structural offsets are drained per bounded chunk rather than retained for the whole input.
Writer memory	`O(depth)`	Writers stream. The only state is the open-container stack, bounded by `max_depth`.

These are targets, enforced by a benchmark that records measured numbers to a result file. Following the practice inherited with this documentation tree: a number in a table comes from a recorded result file, never from a keyboard.

26. Diagnostics and Error Messages

An error message is a user interface, and most converters have a bad one. The target here is the Elm standard: say what happened in plain language, point to the relevant input or operation, explain the consequence, and give the user a specific next step. Naming the failure without giving direction is not an Elm-style diagnostic. A bare error code, a parser exception, or "invalid file" is a defect.

Every error and warning answers these questions, in this order:

1. **What happened?** A short human title and a plain-language explanation using source-format terms only when they help.
2. **Where did it happen?** A source span, archive member and logical document location, command-line argument, or output path. Binary formats use logical locations such as "table 3, row 7" rather than leaking only an XML offset.
3. **What did zenfmt do?** Stop without committing output, continue with a stated fallback, or complete with a precisely named loss.
4. **What can I do next?** At least one concrete direction: a corrected spelling, a complete command, a safe limit override with its risk, a source edit, or a statement that no automatic recovery exists and which source must be kept.

Notes answer the first three questions and may omit the last only when they describe a successful, lossless choice; a degradation note still provides a direction. "Try again" is not a direction. A command that cannot be copied and run, or advice that requires knowledge not present in the report, is not sufficient.

26.1. The report

```
pub const Severity = enum { note, warning, err };

pub const Direction = struct {
    /// Short imperative heading: "Select the format explicitly".
    title: []const u8,
    /// Why this action helps, including risk when it weakens a safety limit.
```

```
    explanation: []const u8,
    /// A complete argv vector when a command is the useful action. The
    /// renderer performs platform-appropriate quoting.
    command: ?[]const []const u8 = null,
    /// An exact replacement when the problem is a token or source span.
    replacement: ?[]const u8 = null,
};

pub const Report = struct {
    severity: Severity,
    /// Stable namespaced machine id: "core.unknown-input-format".
    code: []const u8,
    /// Short human banner: "UNKNOWN INPUT FORMAT". Wording may improve
    /// without breaking scripts or suppression files.
    title: []const u8,
    /// What happened and, when known, why.
    problem: []const u8,
    /// What was or was not produced, preserved, ignored, or changed.
    consequence: []const u8,
    /// Set when this report describes a degraded or dropped construct.
    loss: ?LossTier = null,
    /// The offending source, argv, or path context with highlighted spans.
    context: ?Context,
    /// Ordered, concrete next actions. Non-empty for `warning` and `err`.
    directions: []const Direction,
    /// How many times this report fired. Repeats are counted, not repeated.
    count: u32,
    /// A bounded set of distinct locations retained after aggregation.
    samples: []const Context,
};
```

`Context` is a tagged union covering source excerpts, logical document paths, archive members, command-line spans, and filesystem paths. It separates the human label from byte offsets so the text and JSON renderers expose the same facts without parsing formatted prose.

Rendering rules: a rule-to-margin banner carrying the title and relevant source name; the problem in prose; context with line numbers and caret spans when available; the consequence; then a `What you can do` section containing the directions. Commands are visually separate and shell-quoted for the current platform. Color is used only when the destination is a terminal, so piped output stays diffable. The renderer never depends on color alone and never prints an OS error without translating its impact on the conversion.

Reports aggregate only when `code`, consequence, and directions are identical. The aggregate retains the first few distinct contexts up to `limits.max_report_samples`, then states how many additional locations were omitted. A document with four hundred comments therefore produces one useful report rather than four hundred messages without pretending they all occurred at one location. Display flags do not affect persistence: for a successful path conversion every report is also recorded structurally in the adjacent manifest. Failed conversions commit no manifest; the reports remain available in the returned `Conversion` and through `--reports=json`.

Report construction validates the contract before accepting a value: codes are stable lowercase namespaced ASCII, titles and problem/consequence text are non-empty, warnings, errors, and every report carrying a loss tier have a direction, contexts are within the input they name, and sample/count invariants hold. Built-in reports use constructors for common cases such as an

unknown choice, a bounded-resource refusal, a malformed source span, a lossy fallback, and an I/O path failure. Plugins use the same constructors rather than composing ad hoc paragraphs.

Each format library owns a comptime diagnostic catalog beside its reader and writer descriptors. A catalog entry fixes the code, default severity, required context kind, consequence template, and direction template. The format's ZDS mapping and omissions tables cite these codes. This makes remediation part of the plugin contract and review surface, rather than wording added after the parser is finished.

The contract covers more than parser failures. CLI usage mistakes use command- line contexts and corrected invocations; path and permission failures name the operation that failed and a usable alternative; plugin descriptor `@compileError` messages point to the declaration, show the expected Zig type or signature, and state the edit required. The latter cannot be runtime `Report` values, but they follow the same four-question structure.

One failure path cannot rely on the arena: allocation exhaustion. `Conversion` therefore reserves an inline emergency report whose code, problem, consequence, and directions are static strings. The engine can return `core.out-of-memory` after cleaning up without another allocation. The CLI's report renderer also has a fixed-buffer fallback, so it can still say that no artifact was committed and suggest a smaller input or a process with more available memory. Failure to write stderr itself is the only case in which the CLI cannot communicate a diagnostic.

26.2. What it looks like

An unrecognized format name, with the nearest match by edit distance:

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

I do not recognize `docs` as an input format.

    zenfmt --from docs report.docx
                  ^^^^

Did you mean `docx`?

These are the input formats I know:

    asciidoc   csv    docx   html   markdown
    odt        rst    rtf    text   xlsx

What you can do:

    Select the intended format explicitly:

        zenfmt --from docx report.docx
```

A limit refusal, which must explain itself, because the honest response to a zip bomb is a refusal the user can understand and — if they are sure — override:

```
-- ARCHIVE EXPANDS TOO FAR ---------------------------- handbook.docx

The entry `word/document.xml` expanded to 214 MB from 486 KB, a ratio of
451:1. I stopped at the 200:1 limit.

A ratio this high is usually a file built to exhaust the memory of the
program reading it. It can also be a legitimate document with a very
large amount of repeated content. No output file was created.
```

```
What you can do:

    Verify the document came from a trusted source. If it did, raise only
    this limit for this run. This permits substantially more decompression:

        zenfmt --limit max_compression_ratio=600 handbook.docx
```

A structural problem in the source, shown in context, with what was done about it:

```
-- MISMATCHED TABLE ROW ------------------------------- quarterly.docx

Row 7 of table 3 has 5 cells, but the table declares 4 columns.

    405 | <w:tblGrid>
    406 |   <w:gridCol w:w="2400"/>        4 columns declared here
        |   ...
    412 | <w:tr>
    417 |   <w:tc>                         this is the 5th cell
        |   ^^^^^^

I kept the extra cell and widened the table to 5 columns. A Markdown
table must be rectangular, so every other row now ends with an empty
cell.

What you can do:

    Open table 3 in the source document and either remove the accidental
    fifth cell or make all rows five columns. Run with --strict if zenfmt
    should stop instead of repairing malformed tables.
```

A lossy conversion, reported at the end of a successful run:

```
-- CONVERTED WITH LOSSES ------------------------------ handbook.docx

The conversion succeeded. These constructs have no Markdown equivalent
and were dropped:

    412  comments
     38  tracked-change deletions
      6  text boxes
      2  embedded objects (1 chart, 1 spreadsheet)

Those constructs are absent from Markdown. The source DOCX is unchanged.

What you can do:

    Keep the source DOCX if those details matter. To stop before zenfmt writes
    an incomplete conversion, use:

        zenfmt --strict handbook.docx
```

26.3. The lossiness tiers

The tier of a construct is decided in the plugin's ZDS, not by the implementer at the keyboard.

Tier	Emits	Meaning
Represented	nothing	The construct survives into the AST.
Degraded	`note`	Something reached the output in a simplified form: a multi-paragraph table cell flattened, a nested table inlined, a font choice reduced to `code`. The plugin's record states which construct degrades to which node.
Dropped	`warning`	Recognized and produced nothing: comments, tracked-change deletions, headers and footers, embedded objects. The user is told what and how many.

An unrecognized construct is not a fourth tier — it is a `warning` titled `UNHANDLED CONSTRUCT` naming the element, its logical source location, and the fallback zenfmt used. Its directions ask the user to retain the source and provide the report code and producer details when filing an issue. An unimplemented case is therefore visible in the field rather than silent. This is the mechanism that distinguishes a decision from a gap at runtime, as the omissions table does at review time.

`--strict` promotes warnings to errors and exits non-zero; `--quiet` suppresses notes on the diagnostic stream; `--reports=json` renders the report stream as JSON Lines for a script driving a bulk conversion. These options do not change the canonical `reports` array stored in the artifact manifest. The JSON representation contains `code`, `severity`, `problem`, `consequence`, structured contexts, and structured directions; it is not the terminal text wrapped in JSON. Text and JSON are two renderers of the same `Report`, so an embedding application and the CLI receive the same remedy information.

27. Reading the Office Formats

27.1. The container: OOXML and ODF are ZIP archives

DOCX, XLSX, PPTX, and the ODF family are ZIP archives of XML parts. One `zip.zig` serves all of them, and it is the component that most needs its limits specified, because it is where hostile input does the most damage for the least effort.

The reader works from the central directory at the end of the archive — which is why these plugins declare `input = .seekable`. It never walks local file headers to discover entries, the parsing mode ZIP-confusion attacks exploit, and it never expands an entry it was not asked for. A DOCX conversion typically touches four entries out of several dozen.

Limit	Default	What it prevents
`max_archive_entries`	4,096	A directory with millions of entries exhausting memory before any entry is read.
`max_entry_uncompressed`	256 MiB	One entry expanding without bound.
`max_total_uncompressed`	1 GiB	Many entries each individually under the cap.
`max_compression_ratio`	200:1	The classic zip bomb, refused during streaming rather than after expansion completes.
`max_entry_name_bytes`	1,024	Pathological names.

Three rules have no override, because there is no legitimate use: an entry name that is absolute, contains a `..` component, contains a backslash, or contains a NUL rejects the archive; encrypted entries are refused with a clear message; only `stored` and `deflate` are accepted.

27.2. The XML layer

`xml.zig` is a pull parser shared by every XML-based format, returning events rather than building a tree, so peak memory is proportional to depth rather than document size. Its security posture is a short list, and all of it is refusal:

- **No DTD processing whatsoever.** A `DOCTYPE` declaration rejects the document. This makes entity-expansion attacks structurally impossible rather than bounded. No office format has a legitimate reason to carry a DTD.
- **No external entities, no `xinclude`**, and no filesystem or network resolution of any reference. The parser has no I/O.
- **The five predefined entities and numeric character references only.** References above the Unicode range or in the surrogate range are errors.
- **`max_xml_depth`**, default 256, bounding the explicit stack.
- **Namespace prefixes are resolved to URIs**, and elements matched on the URI. A document binding `w:` to something other than the WordprocessingML namespace does not get to impersonate one.

27.3. DOCX

Source construct	AST result	Notes
`w:p`	`paragraph`	The default when

Source construct	AST result	Notes
		no other rule applies.
`w:p` with a heading `w:pStyle`	`heading`, level in `attr`	Resolved through `styles.xml`: the built-in `Heading1`..`Headir` and any style whose `w:basedOn` chain reaches one. Levels above 6 clamp, with a note.
`w:p` with `w:numPr`	`list_item` in a synthesized list	The hardest mapping; see below.
`w:r` with `w:rPr` flags	nested `strong`, `emphasis`, and the rest, in canonical order	The flag-to-nesting conversion.

Source construct	AST result	Notes
		A toggle with `w:val="0"` clears rather than sets.
`w:t`	`text` and `space` nodes	`xml:space="pres` is honored.
`w:rPr/w:vertAlign`	`superscript` / `subscript`	
`w:rPr/w:rFonts` naming a monospace family	`code`	Matched against known monospace families. Degraded, with a note: the source expressed a font, the AST expresses a role.
`w:br`	`hard_break`	`w:type="page"` dropped with a note.

Source construct	AST result	Notes
`w:tab`	`space`	Markdown has no tab semantics.
`w:hyperlink`	`link`	`r:id` resolved through `word/_rels/document.xml.re` an anchor-only link becomes a fragment target.
`w:tbl` / `w:tr` / `w:tc`	`table` / `table_row` / `table_cell`	Header rows go under `table_head`, the rest under `table_body`.
`w:tcPr/w:gridSpan`, `w:vMerge`	cell spans	Continuation cells fold

Source construct	AST result	Notes
		into the originating cell.
`w:drawing`, `w:pict`	`image`	The relationship target is recorded in `media`; bytes are not extracted in the first release.
`w:footnoteReference`	`note`	The body comes from `footnotes.xml` and is appended outside `body`.
`w:sdt`	`container` with a class	The content control's tag

Source construct	AST result	Notes
		becomes the class, rather than being unwrapped and lost. This is `container` doing the job it exists for.
`w:ins`	content kept	Tracked insertions accepted.
`w:del`	dropped	Tracked deletions rejected, with a warning naming the count.
`w:instrText`	dropped except `HYPERLINK`	Field codes are

Source construct	AST result	Notes
		computed values zenfmt cannot evaluate.

27.3.1. Numbering, the hard case

DOCX has no list element. A list is a maximal run of consecutive `w:p` paragraphs each carrying a `w:numPr` with a `w:numId` and a `w:ilvl`, where `numId` points through `numbering.xml` to an abstract definition saying whether each level is a bullet or a number and where it starts. The reader must infer the structure:

1. Accumulate consecutive numbered paragraphs sharing a `numId`.
2. Open a list when an `ilvl` first appears, and a nested list inside the current `list_item` when `ilvl` increases.
3. Close lists when `ilvl` decreases, when `numId` changes, or when a paragraph without `w:numPr` intervenes.
4. Take ordered-versus-bullet, the number style, and the start from the abstract definition for that level, not from the paragraph.
5. An `ilvl` jumping by more than one opens the intervening levels as empty items, because the AST requires a list to contain `list_item`.

Tightness has no DOCX representation. Lists read from DOCX are marked tight, which round-trips more cleanly through Markdown, and the choice is recorded here so it is not rediscovered as a bug.

27.3.2. Deliberate omissions

Recognized and dropped, each with a warning: comments and comment ranges; tracked-change deletions and formatting revisions; headers, footers, and page numbering; section properties, page size, margins, and columns; explicit page and column breaks; fonts, sizes, colors, and highlighting beyond the nodes above; text boxes, shapes, and SmartArt; embedded OLE objects; bookmarks and cross-reference fields; and revision save IDs.

Deferred rather than dropped: OMML mathematics — the `math` node exists, so this is a mapping to write rather than a node set to change — and image byte extraction, where only the extraction is absent.

27.4. The other formats

Format	Approach and principal difficulty
ODT / ODS / ODP	Same container and XML machinery. `text:h` carries its level in `text:outline-level`, easier than DOCX. Character styles are the hard part: `text:span` names a style resolved through `office:automatic-styles` and `styles.xml`, following `style:parent-style-name` chains, to

Format	Approach and principal difficulty
	learn whether `fo:font-weight` is bold. Lists are explicit, which removes the DOCX inference problem entirely.
RTF	Neither XML nor a container: a brace-delimited group language. A group stack carrying character state, `\b` and `\i` as toggles inherited by nested groups, `\par` ending a paragraph, `\'hh` hex escapes in the code page named by `\ansicpg`, `\uN` with its skip-count convention, and destination groups introduced by `\*` skipped wholesale. RTF in the wild comes from dozens of producers with incompatible habits, so the reader is error-tolerant by design: an unknown control word is skipped with a note rather than failing the document.
XLSX / ODS	Each sheet becomes a `heading` naming it, followed by a `table`. Cell text usually lives in `sharedStrings.xml`. Number formats are applied for dates and percentages, since a raw serial date would be worse than useless. Formulas are not evaluated; the cached value is used, with a note when absent. Sparse sheets must be materialized to keep rows rectangular.
PPTX / ODP	Each slide becomes a `heading` from the title placeholder followed by the body placeholders. Speaker notes append as a `container` with class `notes`. Positioning, animation, and non-text shapes are dropped. A presentation loses the most in this projection, and the reports say so loudly.
Markdown	A CommonMark reader with the GFM extensions. As much a test instrument as a feature: with it and the Markdown writer, `read ∘ write ∘ read` can be checked for the fixed-point property that anchors the fuzz suite.
AsciiDoc	Blocks are delimited by run-length markers, and admonitions, sidebars, and open blocks map to `container` with a class. Attribute entries become document metadata. The `include::` directive is **not** followed — a converter that reads arbitrary files named by its input is a file-disclosure primitive — and the refusal is reported rather than silent.
reStructuredText	Indentation-structured, with directives and roles as the extension mechanism. Directives map to `container` and roles to `span`, both carrying the directive name as a class. Reference-style links need a second pass, since a target may be defined anywhere in the document.
HTML	A tolerant parser. The reason `container`, `span`, and `raw` are in the node set, and the format most likely to produce deeply nested input, so it is the primary consumer of `max_depth`. Under IR v2 (ZDS 0013) it is also the first extension-node producer: `<details>` becomes an `extension` owned by `ai.insan.zenfmt.html`, its summary and content as the fallback subtree, with a same-owner nested `<details>` degrading to a plain `container` so the validator's nesting rule holds by construction. `colspan` and `rowspan` carry into table-cell span properties, where the Markdown writer's span degradation engages.
CSV / TSV	RFC 4180 with quoting, embedded newlines, and doubled quotes. One `table` with the first row as `table_head`. Note that the preliminary

Format	Approach and principal difficulty
	sketch's example reader — which emitted every field as its own paragraph and lost the final field of a file with no trailing newline — is not what ships.
Plain text	Blank-line-separated blocks become paragraphs; line endings normalized. The phase 1 plugin, which exists to exercise the pipeline end to end.

28. The Markdown Writer

The sole writer of the first release, and therefore the definition of what zenfmt's output looks like.

Dialect. CommonMark plus the GFM extensions for tables, strikethrough, task lists, and footnotes. `--markdown-dialect` is reserved for later variants; the first release ships one.

Escaping. The minimum that preserves meaning at the position where text is emitted, because over-escaping produces output that is correct and unreadable. `#`, `-`, `+`, `>`, and a digit followed by `.` or `)` are escaped only at the start of a line. `*` and `_` only where they would start or end emphasis, which for `_` means only at a word boundary. `[`, `]`, and a `!` before a `[` always. `|` only inside a table cell. Nothing inside a code span or block; instead the fence is chosen longer than any backtick run in the content.

Inline emission. The writer walks the inline tree and emits a delimiter pair per container node — which the tree makes trivial, and which was the awkward part of the flat-run design it replaced. `**` for `strong` and `_` for `emphasis`, because the mixed pair is unambiguous in positions where one character for both is not; `~~` for `strikethrough`; backticks for `code`, with the fence one longer than the longest run inside; a trailing backslash for `hard_break` rather than two spaces, because trailing whitespace does not survive editors and diffs.

`underline` and `small_caps` have no CommonMark spelling. Their children are emitted unstyled with a `STYLE DROPPED` note — the degradation tier, and the reason that tier exists.

Block emission.

Node	Output
`heading`	ATX, never Setext. The identifier is emitted as a GFM anchor when present.
`paragraph`	Text, then a blank line. `plain` omits the blank line.
`quote`	`>` on every line, including blank ones inside the quote.
`list`	`-` for bullets; the recorded number style and start for ordered lists. Nested content indented to the marker width. Tight lists have no blank lines between items; loose ones do.
`code_block`	Fenced with at least three backticks, longer if the content contains a longer run, with the first class as the language. Indented blocks are never emitted, because they interact badly with list indentation.
`table`	A GFM pipe table with an alignment row, padded to a common column width up to a configurable limit.
`container`	Its children, attributes dropped with a note — unless `--markdown-divs` selects fenced div syntax.
`note`	Collected and emitted at the end as GFM footnote definitions.

Node	Output
`thematic_break`	`---`.
`raw_block`	Verbatim if its format is Markdown or HTML; dropped with a note otherwise.

GFM table cells cannot contain block structure. A cell holding more than one block, or any block that is not a paragraph, is flattened with a note; a nested table becomes a placeholder and a warning. These are the honest failure modes of a pipe table, and the `flatten-nested-tables` filter exists so a user can make the transformation explicit and inspect it instead.

Determinism. Byte-identical output for identical input, across platforms and build modes: `\n` always, never a trailing space, exactly one blank line between blocks, exactly one newline at end of file, no hash-ordered iteration, and no locale-dependent formatting. A golden-file suite is worthless without this, and the property is cheap to hold if it is decided at the start.

29. Repository and Build Layout

Library boundaries justify directories; miscellaneous categories do not. The repository has a core library, one library per format, small shared libraries for format families such as OOXML, an umbrella library, and the CLI. Source inside each library stays flat and shallow in the TigerBeetle style.

```
zenfmt/
├── build.zig               # modules, CLI, tests, docs, fuzz, benchmarks
├── build.zig.zon           # one release and dependency lock
├── core/                   # `zenfmt_core`, no format knowledge
│   ├── src/
│   │   ├── root.zig        # Bundle, convert contracts, public types
│   │   ├── ast.zig
│   │   ├── builder.zig
│   │   ├── payload.zig
│   │   ├── metadata.zig
│   │   ├── manifest.zig
│   │   ├── plugin.zig
│   │   ├── pipeline.zig
│   │   ├── report.zig
│   │   └── limits.zig
│   └── tests/
├── support/                # libraries shared only by related formats
│   ├── scan/src/root.zig   # scalar/vector bounded scanners
│   ├── xml/src/root.zig    # pull XML parser
│   └── ooxml/src/          # ZIP/package relationships and content types
├── formats/
│   ├── markdown/           # `zenfmt_markdown`
│   │   ├── src/{root,reader,writer}.zig
│   │   └── tests/
│   ├── docx/               # `zenfmt_docx`
│   │   ├── src/{root,reader,styles,numbering,plugin_data}.zig
│   │   └── tests/
│   ├── text/
│   ├── csv/
│   ├── odt/
│   └── rtf/
├── src/                    # umbrella `zenfmt` library
```

```
│   ├── root.zig            # re-export core API through Default
│   └── default_bundle.zig  # imports the standard format libraries
├── cli/src/main.zig        # imports only `zenfmt`
├── tests/                  # cross-format and end-to-end tests
│   ├── conversion.zig
│   ├── manifest.zig
│   └── corpus/
├── examples/
│   └── filters/            # a user project showing the build.zig pattern
├── tools/zds.zig
└── docs/
```

The root build exposes each directory above as a named Zig module. They share a release version and dependency lock, but their import graphs are enforced: format libraries depend on `zenfmt_core` and optional support libraries; core depends on none of them; the umbrella depends on the default formats; the CLI depends only on the umbrella. A format's reader and writer live together so they can share one preservation-data schema without exporting format details to core.

This is library separation, not directory nesting for its own sake. Each module can be tested and benchmarked alone, applications can link only selected formats through `zenfmt_core.Bundle`, and a format library can be published separately later without changing its API. The umbrella keeps the common developer experience at a single `@import("zenfmt")`.

Build steps. Alongside the ZDS steps from ZDS 0001: `zig build` for the library and binary, `test`, `fuzz`, `bench`, `docs` for autodoc, and `fmt-check`.

30. Command-Line Interface

```
zenfmt [options] INPUT

  -f, --from FORMAT        Input format. Default: from the extension, then
                           from content.
  -t, --to FORMAT          Output format. Default: markdown.
  -o, --output PATH        Output file. Default: INPUT with the new extension.
      --stdout             Write the document to stdout instead of a file.
      --metadata-out PATH  Persist the manifest produced with --stdout.
      --overwrite          Replace existing artifact and manifest paths.
      --filters            Run the pipeline compiled into this binary.
      --list-formats       Print the registry's readers and writers.
      --list-filters       Print the filters compiled into this binary.
      --strict             Promote warnings to errors; exit non-zero.
      --quiet              Suppress notes.
      --reports=FORM       text (default) or json.
      --limit NAME=VALUE   Override one resource limit.
  -h, --help
  -V, --version
```

The first screen of `--help` starts with the common examples:

```
zenfmt report.docx                # report.md + report.md.zenfmt.json
zenfmt report.docx -o notes.md    # notes.md + notes.md.zenfmt.json
zenfmt report.docx --stdout       # document bytes only on stdout
```

For a path input, omitting `-o` derives a sibling output path from the writer's primary extension and refuses to overwrite an existing file. This makes the safe, metadata-preserving path the shortest command. `INPUT` is `-` for stdin; stdin requires `--from` and either `-o PATH` or `--stdout`.

Reading from stdin is supported for formats whose `input` is `.bytes`. A container format needs to seek, so `zenfmt -f docx` from a pipe spills to a temporary file and says so, rather than failing.

A path output always writes exactly `<output>.zenfmt.json` beside the artifact; that location is not configurable. `--metadata-out` is accepted only with `--stdout`, where no adjacent artifact path exists. Stdout remains exclusively the requested document bytes.

Argument parsing allocates nothing: flags match a comptime table derived from the same declarations that generate `--help`, and values are slices of `argv`. Exit codes are `0` success, `1` conversion error, `2` usage error, and `3` limit exceeded — distinguished because a bulk script wants to treat a zip bomb differently from a malformed document.

31. Coding Standard

The style follows TigerBeetle's, because its rules are chosen for exactly this situation: a program that parses untrusted input, must not crash, and must stay readable for years.

Rule	How it applies here
Assert liberally	Every function asserts its preconditions and postconditions; at least two assertions in any non-trivial function. The AST invariants are the richest source: a walker asserts positive, nested `subtree_len` values as it advances, and the builder asserts stack balance on every close. Assertions are on in `Debug` and `ReleaseSafe`, and the released binary is `ReleaseSafe`.
No recursion	No traversal, parser, or writer recurses. Each keeps an explicit stack bounded by `max_depth`. A safety rule before a style rule: nesting depth is attacker-controlled, and a bounded stack turns a crash into a report.
Bound every loop	Every loop has a known upper bound, and the bound is asserted. A parser loop that could fail to advance asserts that its cursor moved.
Allocate at the boundary	TigerBeetle allocates everything at startup. A converter cannot know its document sizes in advance, so the adaptation is: one arena per conversion, capacity reserved at input/chunk boundaries, and geometric growth when an estimate is exceeded. There is no distinct allocation per node, token, or text run. The deviation is deliberate and measured.
Small functions, small files	Functions under 70 lines, files under 1,000. `docx_reader.zig` is already known to need splitting into `docx_styles.zig` and `docx_numbering.zig`, and the split is decided in advance rather than after the file overflows.
Explicit types and names	Typed `enum(u32)` indices rather than interchangeable `usize` values, so a block index cannot be passed as a byte offset or inline index. `snake_case` variables, `camelCase` functions, `PascalCase` types. No abbreviation that is not already in the format specifications.
Zig 0.16 baseline	The package minimum is exactly 0.16. Examples and implementation use explicit `std.Io`, `std.Io.Reader`/`Writer`, typed

Rule	How it applies here
	`std.Io.Dir`, and the allocator-taking unmanaged collection APIs. CI compiles every Zig snippet that is intended as complete code.
Zero technical debt	A change lands complete: with its tests, its diagnostics, and its ZDS updated. There is no "clean this up later" state in this repository.

32. Testing Strategy

Layer	What it establishes
AST validator	Runs on every conversion in `Debug` and `ReleaseSafe`, after every filter stage, in every test, and on every fuzz iteration. The oracle: a plugin that produces a structurally impossible tree fails even when nothing crashes.
Tree tests	`tests/trees.zig` asserts the AST a reader produces, with no writer involved. The only way to test a mapping without conflating it with the writer's choices, and the layer at which a format record's mapping table is verified row by row.
Golden files	Input and expected output. The regression net, and the artifact a reviewer reads to see what a mapping decision does. Updated only by an explicit `--update-goldens` run, so a change to output is always visible in a diff.
Filter tests	Input tree, filter, expected tree. Plus two properties every filter is checked against: an identity filter produces byte-identical arrays, and running a filter twice equals running it once for filters that declare themselves idempotent.
Round-trip fixed point	Markdown → AST → Markdown must be idempotent from the second pass onward. The first pass may normalize; the second must change nothing. A cheap, mechanically checkable property that catches a large class of writer bugs.
Fuzzing	`std.testing.fuzz` over each reader and each filter, with the validator as oracle. Properties: never crash, never exceed the limits, never fail to terminate, always leave the builder balanced. Seeded from the golden corpus.
Adversarial corpus	Checked-in zip bombs, quadratic-entity XML, traversal entry names, truncated archives, encrypted entries, and pathologically nested HTML — each asserting the specific refusal, so a later relaxation of a limit fails a test rather than silently widening the attack surface.
Report snapshots	Every stable report code has text and JSON golden files covering its problem, context, consequence, and directions. Tests reject any warning or error with no direction, any source-bound report with no context, commands containing unresolved placeholders, aggregation that loses its bounded samples, and text/JSON renderers that

Layer	What it establishes
	disagree. An error message is a user interface, and it regresses like one. CLI usage failures are included. Allocation-failure injection at every allocation site must still return the inline `core.out-of-memory` report and leave no artifact.
Benchmarks	Throughput, allocation counts, and the filter fast-path ratio, written to result files. Numbers quoted in any document come from those files.
Differential	Scalar and vector scanners consume the same corpus and must produce identical events, offsets, ASTs, reports, and failures. Format conformance suites remain the semantic oracle.

33. Security Considerations

The threat model is one sentence: an attacker supplies the input file, and possibly a great many of them, to a zenfmt process converting documents on someone else's behalf.

Resource exhaustion is the primary risk, addressed by the limits above, each with a default recorded here, a CLI override, and a test asserting the refusal. Ratio limits are checked during decompression rather than after, so a bomb is refused while it is expanding.

Memory safety is Zig's to provide in `ReleaseSafe`, which is what the released binary ships in. The design makes the guarantee cheap: no recursion anywhere, explicit stacks bounded by `max_depth`, and bounds-checked slicing throughout.

Entity expansion is not bounded but excluded: no DTD processing at all.

Path traversal cannot escape a directory, because zenfmt writes only its output file. Archive entry names are validated anyway, because a name escaping the archive namespace is evidence of a file built to confuse a consumer, and refusing it is free.

Reference following is refused across the board: AsciiDoc `include::`, reStructuredText `.. include::`, XML external entities, and HTML `src` attributes are never resolved. A converter that reads files named by its input is a file-disclosure primitive. Each refusal is reported rather than silent, so a user converting a document that genuinely uses includes learns why the output is incomplete.

Content dangerous downstream is deliberately out of zenfmt's hands: a link target in the input becomes a link target in the output, unmodified and unresolved. zenfmt never fetches a URL and does not sanitize one, because it cannot know the destination's context. The documentation must say so where a user will see it, and an HTML writer will need its own record on sanitization.

Filters run in-process with full program privileges. They are compiled from source the user chose, exactly as `build.zig` is, and the trust model is the same: running a filter is running a program. This is stated in the filter documentation, because the `build.zig` analogy might otherwise suggest a sandbox.

Supply chain: no third-party Zig dependencies in the first release. `zip.zig` and `xml.zig` are written here rather than vendored — a real cost in effort, and a real reduction in attack surface for a program whose entire job is parsing untrusted input.

34. Operational Considerations

A single static executable: no runtime dependencies, no configuration file, no network access. It runs the same in a container, a CI job, and a shell.

For bulk conversion the relevant surfaces are the exit codes above, `--reports=json`, and the guarantee that a failed conversion writes no output file. A caller processing ten thousand documents can distinguish "malformed" from "hostile" from "converted with losses you should look at" without parsing prose.

Cross-compilation is expected to work for every supported target without special handling: no C dependencies, and no platform-specific code outside the CLI's file handling.

35. Alternatives Considered

Alternative	Why it was not chosen
A flat inline layer of styled runs, with no inline nesting	This record's own earlier draft. Rejected: filters need node identity, and `link`, `note`, `span`, and `citation` need children regardless. The argument is under *Why the tree is a tree*. What survives is the canonical nesting order required of flag-based readers.
A pointer-based recursive tree	The natural expression of the AST, and what most implementations use. Rejected for storage only, not for semantics: pointers forfeit dense tag scans, `O(1)` subtree skips, compact typed indices, and bulk-copyable same-kind ranges. The public tree shape is unchanged.
Mutable in-place filters	Faster in the abstract, and the obvious thing to want. Rejected: insertion into a preorder array is `O(n)`, aliasing between the tree a filter reads and the one it writes is a bug factory, and the rebuild with subtree sharing is already within a constant factor of in-place for realistic edit counts. Immutability also makes a stage trivially testable.
Lua or another embedded scripting language for filters	It lowers the barrier considerably. Rejected for the first release: an interpreter is a large dependency and a large attack surface, and the `build.zig` model gives type checking and native speed. The cost — a Zig toolchain to write a filter — is

Alternative	Why it was not chosen
	real, and recorded as an open question.
External filters over a JSON protocol	Useful for language-neutral automation, but deferred until zenfmt has a stable serialized AST plus clear spawning, sandboxing, cancellation, and diagnostic behavior. The artifact manifest is not an AST protocol.
Runtime plugin registration with vtables	Would allow dynamically loaded formats and a smaller binary. Rejected: it forfeits the compile-time exhaustiveness check that makes adding a node a compile error in every writer, adds an indirect call per node, and buys a capability nobody asked for.
Parent index instead of `subtree_len`	Simpler to build, since nothing is patched on close. Rejected: it makes "iterate the children of this node" a full scan, and loses the contiguous subtree property the filter system depends on.
One node array for blocks and inlines together	Would collapse the walker to a single code path. Rejected: it forfeits the block/inline type distinction in the public API, makes the tag column heterogeneous and so less useful for scanning, and would leave the validator's structural rules expressible only at runtime.
Multiple writers in the first release	Rejected deliberately. One writer means the AST is exercised against many readers and one consumer, the configuration in which its adequacy is easiest to judge. A second writer before the AST settles would harden the wrong decisions.
No adjacent manifest	Simpler file semantics, but it permanently discards useful source details that do not belong in the shared AST and leaves plugins no safe way to cooperate across a lossy intermediate format.

Alternative	Why it was not chosen
YAML, ZON, or a binary manifest	YAML has a larger and more ambiguous parsing surface, ZON couples every consumer to Zig, and a custom binary format impedes inspection and unknown field retention. Canonical JSON has broad, language-neutral tooling.

36. Decisions Clarified by Review

The review changed six load-bearing parts of the design:

- Pandoc remains inspiration for a shared AST and filters, not a compatibility target. zenfmt owns its node and metadata schemas.
- The public AST is expressed as Zig tagged-union views generated from one comptime schema; the compact SoA representation stays private.
- The common application API is one `convert` call and plugins export one comptime-validated descriptor.
- Each format is a separate Zig library. The CLI imports only the umbrella bundle, while embedders can compose a smaller bundle; core defines no closed format enum.
- No-op transforms reuse their snapshot. Bulk copying is claimed only for genuinely contiguous same-kind ranges; cross-array references are stable append-only indices.
- Every path conversion writes a digest-bound adjacent JSON manifest, and staged/vector parsing work remains benchmark-gated and bounded.

37. Open Questions

37.1. The cost of Zig-only filters

The `build.zig` model gives type checking, native speed, and no interpreter. It also means writing a custom filter requires a Zig toolchain and a compile. For a user who only wants to shift heading levels, that is a steep gradient. Options, in increasing order of cost: ship enough built-in filters that common cases need no compile; add a small declarative select-and-rewrite expression; define a stable zenfmt JSON AST protocol for external processes. Which, and when?

37.2. Image bytes

`Media.bytes` is never filled in the first release. Should the release extract images to a sidecar directory (`--extract-media`), or is recording the archive-internal path enough for a Markdown-only release?

37.3. Table alignment source

DOCX and ODT express alignment per paragraph; a GFM table has one alignment per column. Derive column alignment from the majority of its cells, from the header row, or not at all?

37.4. Streaming for very large inputs

The design materializes the whole AST before writing. For a document larger than memory this fails. Is a streaming mode — a reader emitting blocks a writer consumes incrementally — worth the substantial complication of a two-phase plugin contract and the loss of the filter pipeline, given the document sizes actually encountered?

37.5. Filter ordering and conflict

The pipeline runs stages in declaration order, which is simple and predictable. It also means two filters that both match `container` interact in a way the user must reason about. Is declaration order enough, or should a filter declare what it consumes and produces so the pipeline can detect a conflict?

37.6. Encoding detection

RTF names its code page; plain text does not. Is a heuristic detector in scope, or does non-UTF-8 plain text require an explicit `--input-encoding`?

38. Delivery Plan

Phase	Scope	Exit criterion
0	Repository skeleton, module boundaries, `build.zig` with the ZDS steps, this record accepted.	`zig build zds` produces every record; the layout is fixed.
1	`zenfmt_core`; the `zenfmt_text` reader and `zenfmt_markdown` writer libraries; the umbrella default bundle; the CLI; manifest, tree, and golden runners.	`zenfmt input.txt -o input.md` emits Markdown and `input.md.zenfmt.json`. The validator runs on every conversion. Allocations per conversion are `O(log n)`, measured rather than asserted by inspection. Tagged-union/storage view tests cover every schema tag. Every diagnostic renders in the Elm form, carries a validated concrete direction, is stored in the manifest, and has matching text and JSON snapshot tests.
2	The `zenfmt_markdown` reader, `zenfmt_csv` library, the round-trip fixed-point suite, and reader fuzz targets.	Two more readers reach the one writer with no change to `zenfmt_core` — the claim the plugin architecture exists to make, tested rather than asserted. Markdown round-trip is a fixed point from the second pass.
3	`pipeline.zig` with the rebuild transform and subtree sharing, the `FilterContext` API, the five built-in filters, the `examples/filters/` user project, and `--filters`.	A user project compiles its own filter against the published module and produces a working binary. An identity filter is measured at zero AST allocation; sparse edits share side data and bulk-copy only changed spines. Filter fuzzing runs with the validator as oracle. If a transform cannot be written cleanly here, the node set is wrong and this is where we find out.

Phase	Scope	Exit criterion
4	The `scan`, `xml`, and `ooxml` support libraries; the `zenfmt_docx` format library with style, numbering, and plugin-data modules; and the adversarial corpus.	A real-world DOCX with headings, nested styled runs, hyperlinks, numbered and bulleted lists, tables with merged cells, and footnotes converts correctly. Every limit has a test asserting its refusal, and every refusal renders as a report a user can act on. Each file is under 1,000 lines.
5	Separate ODT, RTF, XLSX, and PPTX format libraries and a plugin ZDS for each.	The 0.1 release: every office format converts to Markdown, each with a record stating its mapping, its omissions, and its round-trip expectations.
6+	Separate AsciiDoc, reStructuredText, HTML, and remaining lightweight-markup libraries.	The formats that exercise `container` and `span` as the extension mechanism, confirming that the node set is closed and does not need to grow per format.
7	The full-coverage pass: high-fidelity upgrades for RTF, PPTX, and ODT (tables, lists, hyperlinks, footnotes, images); ODS, ODP, EPUB, and a native-Zig PDF reader; the `cfb` support library with legacy `doc`, `xls`, `ppt`, and `xlsb` readers; the complete WHATWG named-entity table; and the `zig build benchmark` harness comparing zenfmt against pandoc and anydoc over a downloaded real-world corpus.	Every input format anydoc converts, zenfmt converts — each detected by content signature, each refusing encrypted inputs, and each carrying a format record. The benchmark measures wall latency, CPU time, and peak RSS per tool per corpus file and writes `benchmarks/results/results.md`.

Two phases are placed deliberately. Phase 2 tests the architecture rather than adding capability: if adding a reader requires touching `zenfmt_core`, that is discovered at the cost of two small readers rather than after DOCX is written against the wrong contract. Phase 3 puts filters before the expensive container work for the same reason — filters are the strongest test of whether the AST is genuinely an AST, and a transform that cannot be expressed cleanly is evidence of a node set that needs fixing while fixing it is still cheap.

39. References

- ZDS 0001, The Zen Discussion Process — lifecycle, numbering, and the format-record obligations this document's successors inherit.
- ZDS 0013, Layered Document IR and Writer Lowering — supersedes the AST, builder, transform, and writer-lowering sections of this record with the IR v2 design.
- Pandoc's AST and filter documentation — historical inspiration for using a shared block/inline representation and transforms; not a compatibility or serialization contract.
- Zig 0.16.0 language and standard-library documentation, release notes, and `std.zig.Ast` source — normative for code examples, `std.Io`, collection APIs, typed indices, and the flat struct-of-arrays precedent.

- CommonMark Specification, including its parsing-strategy appendix, and the GFM extensions for tables, strikethrough, task lists, and footnotes — normative for the Markdown reader and writer.
- Langdale and Lemire, *Parsing Gigabytes of JSON per Second* — the staged structural-index design adapted by the bounded scanners.
- Keiser and Lemire, *Validating UTF-8 In Less Than One Instruction Per Byte*, and the simdutf implementation — the basis for benchmark-gated vector UTF validation and transcoding.
- Cameron, Herdy, and Lin, *High Performance XML Parsing Using Parallel Bit Stream Technology* — evidence for vector classification in XML scanning.
- WHATWG HTML Living Standard, *Parsing HTML documents* — normative tokenizer and tree-construction behavior for `text/html`.
- RFC 8785, JSON Canonicalization Scheme — normative encoding for artifact manifests and carried plugin data.
- BLAKE3 specification — the artifact and side-asset digest algorithm.
- ECMA-376, Office Open XML File Formats — Part 1 for WordprocessingML, SpreadsheetML, and PresentationML; Part 2 for the Open Packaging Conventions.
- OASIS Open Document Format for Office Applications v1.3.
- Microsoft Rich Text Format Specification v1.9.1.
- AsciiDoc Language Specification, and the reStructuredText Markup Specification.
- RFC 4180, Common Format and MIME Type for CSV Files.
- APPNOTE.TXT, the ZIP File Format Specification.
- TigerBeetle's coding style — assertions, bounded loops, no recursion, and the file size limit.
- Elm's compiler error messages, for the diagnostic format.