

A Chat with Fable

Contents

Executive Summary.....	3
Images	4
Paul	4
Fable	4
Defensive Markdown	5
Paul	5
Fable	5
Export routes from Markdown to Word	7
Paul	7
Fable	7
Integration into Microsoft Source Manager.....	9
Paul	9
Fable	9
Paul again	10
Fable again	10
Artwork and general nonlinear layouts in Word.....	11
Paul	11
Fable	12
Paul again	14
Fable again	14
Paul again	15
Fable again	15
Hallucination defence.....	16
Paul	16
Fable	16
Paul on hallucination taxonomy	16
Fable again	17

Token use and substitution of LLM tasks by programs 19

 Paul 19

 Fable 19

Role of JSON in citations in Markdown 20

 Paul 20

 Fable 20

Problems with accessing web sites 22

 Paul 22

 Fable 22

Ingestion of Word documents 24

 Paul 24

 Fable 24

Both footnotes and endnotes needed 26

 Paul 26

 Fable 26

 Paul remark 28

 Fable 28

 Paul remark 29

 Fable 29

Multilingual search 29

 Paul 29

 Fable 30

 Paul again 32

 Fable 32

Very long documents 32

 Paul 32

 Fable 33

Summary from Fable 34

Executive Summary¹

Defensive Markdown subset. My working defaults and yours converged almost exactly: ATX headings, hyphen-only loose lists, asterisk emphasis, reference footnotes in end-blocks, pipe tables only, no raw HTML or TeX. The one sanctioned extension is the custom-style span/div, which proved to be the single hook needed for everything exotic.

Export pipeline. Pandoc 3.1.3 is integrated and is the main route, with full Lua filter support (verified by test); python-docx and raw lxml/zipfile give XML-level access for everything Pandoc can't express. RTF is supported but inferior in both directions; docx is canonical.

Word Source Manager. The long-standing "no LLM can do this" claim fell: native b:Sources CustomXML, CITATION fields, and a BIBLIOGRAPHY SDT were injected and survived your corruption test – Word treats the injected source as a full peer of its own. Zotero live-citation fields are the same technique with CSL-JSON payloads, pending a round-trip test with your real library export.

Diagrams and non-linear Word. SVG is the master key: I author it directly as text, Word 2016+ embeds it natively (verified dual SVG+PNG blip), and Convert-to-Shape turns it into editable Word objects. Side headings need no new markup – a framePr inside a custom style in the reference.docx does it. SmartArt was judged not worth wanting.

Architecture law. The session's organising principle, repeatedly confirmed: one austere linear .md, sidecar .svg assets, a standing reference.docx, and filters/post-passes downstream. Complexity only ever accumulates to the right of the source file.

Anti-hallucination. Four levels (plus level 0, rot): URL, metadata, quote, paraphrase. Levels 0–3 are deterministic and belong in code, not tokens – hence cite_audit.py, which runs on your machine at zero cost and was demonstrated catching all three mechanical failure modes. Level 4 is structurally mitigated (paraphrase only with source in context, locator bound at creation) and finally human-adjudicated. Locator strings beat page numbers; text-fragment URLs are their web-native twin.

Piped citations. Agreed direction: pipe-delimited records with a leading type token, positional fields, empty fields preserved, locator as final field – human-readable source of truth from which JSON, CSL, b:Sources, and the formatted References are all generated. The pipe-vs-table conflict is resolved by the Lua filter consuming citation lines first. Your spec is awaited.

URL access realities. My native fetch handles the open web but not JavaScript, paywalls, CAPTCHAs, or per-site rate politeness; Claude for Chrome (beta, all paid plans) answers the paywall/CAPTCHA levels by driving your authenticated browser and

¹ This was prepared by Fable at the very end of the conversation.

handing over when humanity is required. Rate and retry management belongs in scripts and infrastructure, with the model supplying judgment, not loops.

Ingestion. Three tiers: plain text (lossy), pandoc -f docx+styles (recovers your defensive Markdown including custom-styles – ingestion and generation are inverse functions), and raw XML forensics for fields, frames, and citation machinery. RTF rejected as an ingestion route.

Dual note streams. Footnotes (editorial) and endnotes (citations) routed by content grammar – a piped citation *is* an endnote – with your numeric/alphanumeric label convention as declared intent and a lint failing on disagreement. Demonstrated working after one instructive bug: the missing-Endnote-styles fix added a permanent "no dangling style references" assertion to the post-pass.

Multilingual and Unicode. Policy: store NFC, match through a tiered fold ladder (NFC → casefold → NFKC → accent-stripped, with looser tiers reported as warnings). Language as a first-class BCP 47 pipe field; APA-style bracketed translations and standardised transliterations inline in the title field; bidi solved by keeping sources logically clean (no invisible isolates) and handling RTL display in the reference.docx.

Thesis-scale documents. 100k words fits whole in my context, so chunking no longer harms summarisation; file-per-chapter with a unified Pandoc build is the architecture. Three new disciplines at scale: chapter-prefixed note labels, explicit heading IDs, per-file note blocks. End-of-chapter versus end-of-document references is a build parameter on the generated bibliography, not a format decision.

Outstanding for next session: your piped-citation spec, the house reference.docx specification (the system's centre of gravity), a CSL-JSON export of awkward Zotero items, and a mixed-script bibliography as the multilingual torture test. Bring the zip; the container forgets, but the conversation is searchable and the architecture is now proven end to end.

Images

Paul

I have a number of questions to ask.

Are you now able to output realistic images as most other LLMs can?

Fable

[Deliberated product capabilities and consulted verification protocol]²

Short answer: no, not in the photorealistic sense. Claude doesn't generate photos or illustrations the way image-generation tools do, but it can build diagrams, charts, and interactive visuals directly in the conversation – so things like flowcharts, framework

² Text sentences in square brackets are comments, often issued more than once, made by Fable during its construction of answers.

diagrams (Venkatraman's five levels would render nicely, incidentally), data charts, and interactive explainers are all possible, just not raster images like a DALL·E or Midjourney output. [Claude]³

If you do need realistic generated images alongside Claude, the workaround is connecting an external image model via MCP. For example, Hugging Face offers an MCP integration where adding FLUX.1 Krea to your Spaces tools lets Claude generate realistic-looking images through the conversation, with Claude handling the prompting. [Hugging Face]⁴

What I can also do, of course, is analyse images you upload, and search the web for existing images to display inline where that's useful.

Happy to take your next question.

Defensive Markdown

Paul

I use Markdown for all except trivial purposes as my intermediate format, bound to Word at the last minute. So I am not interested in features that do not map elegantly into Word.

Also I impose strict "defensive" restrictions on Markdown to ensure robust writing and conversion. e.g. only "-" for bulleted lists and no use of Compact lists. Is there a particular subset of Pandoc Markdown that you are most used to working within for such purposes? Also I do not need TeX and do not want raw HTML.

I also like all footnote text and citation text to be in blocks at the end. I do not want to propose my format to you until I hear what you are most used to.

Fable

[Inventoried conservative Pandoc subset conventions for Word conversion]

That's a question after my own heart – the answer is that my most reliable register is essentially "conservative Pandoc Markdown," very close in spirit to what you're describing.⁵ The conventions I'm most practised in, and which I know convert cleanly to .docx via Pandoc, are these:

- **ATX headings only** (# through ####), never Setext,⁶ with a blank line before and after every heading. These map directly onto Word's Heading 1–4 styles,

³ <https://support.claude.com/en/articles/9002504-can-claude-produce-images>

⁴ <https://huggingface.co/blog/claude-and-mcp>

⁵ The default setting of Fable is US English spelling and punctuation, hence the punctuation *before* the quote mark. I will redefine that to my usual "OECD-esque" style for LLM output.

⁶ "Setext (Structure Enhanced Text)... is a lightweight markup language used to format plain text documents such as e-newsletters, Usenet postings, and e-mails."
<https://en.wikipedia.org/wiki/Setext>

which is what makes downstream TOC generation and navigation pane behaviour predictable.

- **Hyphen-only bullets with loose spacing** – exactly your restriction. A blank line between list items, and a blank line before and after the list as a whole. Compact lists are where Pandoc's list-parsing ambiguities live (lazy continuation, accidental nesting from indentation drift), so avoiding them is genuinely defensive, not just stylistic.
- **Numbered lists as 1. throughout**, letting Pandoc renumber, rather than hand-maintained sequences.
- **Emphasis with asterisks** (**italic**, ****bold****), never underscores, which misfire around intra-word boundaries and identifiers.
- **Reference-style footnotes** ([¹] in the text, definitions collected in a block at the end of the document) rather than inline ^[...] footnotes. Pandoc converts these to true Word footnotes regardless of where the definitions sit, so an end-block costs nothing in conversion and gains a great deal in source readability and editability. This matches your stated preference exactly, and it's what I defaulted to in the Venkatraman paper.
- **Citations as footnotes rather than CSL/citeproc**. Pandoc's [*@key*] citation syntax is powerful but drags in a .bib file and a CSL stylesheet, which breaks the "self-contained single file" property. For your kind of output I'm most used to putting full bibliographic strings in the footnote definitions, plus (optionally) a final ## References section as plain paragraphs – both of which survive conversion with no machinery.
- **Pipe tables only**, kept simple – no block-level content in cells, no grid tables.⁷ Pipe tables convert to clean Word tables; anything fancier is fragile and tends to need post-conversion styling anyway.
- **No raw HTML, no LaTeX, no fenced divs/spans**. The one exception worth knowing about, even if you choose not to use it: Pandoc's {custom-style="..."} attribute on spans and divs is the only Markdown-native way to invoke a named Word style from a reference.docx. It's strictly opt-in and degrades gracefully, but if your reference template has, say, a "Pull Quote" style, it's the clean hook. I don't reach for it unless asked.
- **Blank lines as the universal separator** – every block element isolated. Plus straight quotes in source, letting --smart (on by default for Pandoc Markdown) produce typographic quotes and dashes in Word.

Things I deliberately avoid because they map poorly or unpredictably to Word: **definition lists** (they convert, but to an odd indented-paragraph structure rather than anything Word-native), **line blocks**, **nested blockquotes** beyond one level, **horizontal**

⁷ <https://quarto.org/docs/authoring/tables.html#grid-tables>

rules as section dividers (Word renders them as a bottom-bordered paragraph, which rarely matches intent), and **task-list checkboxes**.⁸

So in practice my comfortable default is: CommonMark discipline, Pandoc footnote and pipe-table extensions, everything loose and blank-line-delimited, nothing that can't round-trip through `pandoc -f markdown -t docx --reference-doc=template.docx` without surprises.

I'd be interested to see your format spec – it sounds like the delta between it and my defaults may be small, and where you're stricter (you may, for instance, ban pipe tables too, or cap heading depth) I'm happy to adopt it as the house style.

Export routes from Markdown to Word

Paul

What are your export routes from Markdown to Word? Is the main one Pandoc and you have integrated it?

Can you manage Lua filters for more sophisticated Pandoc conversion?

Finally can you as some LLMs can write Word direct via python-docx calls? Finally, what about export to RTF?

One specific requirement I have is the need to write citations direct into Zotero. So capability at the XML level is useful to me.

My vision is to write Word documents using styles and character formatting identical to an intrinsic Word document.

Fable

[Inventoried available tools for document conversion capabilities]

Let me verify exactly what's installed in my environment rather than answering from assumption.

[Inventoried export capabilities and architected XML-level Zotero integration strategy]

I checked the actual toolchain in my container rather than answering from memory, so this is verified. Taking your questions in order:

Pandoc – yes, integrated, and it's the main route. I have Pandoc 3.1.3 installed and invoke it directly from the command line, including `--reference-doc=template.docx` for style mapping. So if you supply (or I build) a reference.docx whose Heading 1–4, Body Text, Footnote Text, Quote, Caption and table styles are defined to your spec, the output uses *named Word styles throughout, with no direct formatting* – which is precisely your "indistinguishable from an intrinsic Word document" criterion. Pandoc's docx writer

⁸ Editor's use of bold.

never emits direct character formatting for structural elements; bold/italic come through as runs, everything else as style references.

Lua filters⁹ – **yes, fully**. Pandoc here is built with embedded Lua 5.4 (+lua), and I just smoke-tested it: a filter remapping Strong to a custom-style="Intense Emphasis" span produced a docx whose XML contains `<w:rStyle w:val="IntenseEmphasis"/>` – i.e. a genuine character style reference, not formatting. So the full filter repertoire is available: AST-level restructuring, mapping Markdown constructs onto your named styles, enforcing your defensive subset (a filter can *reject* or normalise constructs you've banned), promoting footnote blocks, injecting section properties, and so on. I can write, test, and iterate filters in-session and show you the resulting XML as proof.

python-docx – **yes, direct Word authoring**. Version 1.2.0 is installed. It's the right tool when you need things Pandoc's writer can't express: precise field codes, content controls, custom document properties, header/footer manipulation, or surgical edits to an existing document. Its API doesn't cover everything, but it sits on lxml,¹⁰ so anywhere it falls short I can drop to raw OOXML¹¹ – building `w:fldChar/w:instrText` runs by hand, editing `document.xml`, `styles.xml`, or `docProps/custom.xml` directly inside the zip. So "capability at the XML level" is genuinely available, not just in principle.

RTF – **yes**, `pandoc -t rtf` is supported. Honest caveat: the RTF writer is one of Pandoc's less-loved outputs – it embeds formatting rather than referencing a stylesheet, so it won't give you the named-styles purity of the docx route. If RTF is needed as an interchange format it works; if it's needed as a *styled* deliverable, generating docx and converting (or using docx as the canonical artefact) is more faithful.

Zotero – **this is the interesting one, and it's the XML-level case**. Zotero's Word integration stores each citation as a field:

```
ADDIN ZOTERO_ITEM CSL_CITATION
```

followed by a CSL-JSON payload (the item metadata plus `uris` pointing at items in your library), with document-level preferences in a `ZOTERO_PREF` custom property.

Pandoc cannot emit these natively – its `citeproc` route produces static text.

But because the fields are just OOXML runs, I can construct them: post-process a Pandoc-generated docx (or build via `python-docx`) to inject live Zotero fields with correct CSL-JSON.

Two practical caveats. First, for Zotero's "Refresh" to re-link rather than treat citations as embedded orphans, the `uris` must match real items in *your* library (or group library) – so the clean workflow is you exporting CSL-JSON from Zotero (which carries the URIs) and me writing fields from that, rather than me fabricating metadata.

⁹ <https://pandoc.org/lua-filters.html>

¹⁰ "lxml is the most feature-rich and easy-to-use library for processing XML and HTML in the Python language." <https://lxml.de>

¹¹ Office Open XML, https://en.wikipedia.org/wiki/Office_Open_XML

Second, there's a community precedent worth knowing: the *pandoc-zotxt* / "*pandoc to Zotero live citations*" approach does essentially this, so the field format is well documented and stable. I'd want to do a round-trip test on a small document with your actual exported CSL-JSON before trusting it on a real paper.

So the architecture your vision implies is entirely buildable here: your defensive Markdown → Lua filter (enforcement + style mapping) → Pandoc with your reference.docx → optional python-docx/lxml post-pass for Zotero fields and any field codes (TOC, cross-references) → a docx that opens in Word with nothing but named styles and live citations.

The one piece I'd need from you to make it concrete is the reference.docx (or a style spec from which to generate one) and a sample CSL-JSON export.

Integration into Microsoft Source Manager

Paul

A pro forma question. No LLM I have known has ever managed to write into Microsoft Source Manager in a useful fashion.

Fable

[Grappling with XML content type configurations for Word documents]

This is exactly the sort of claim I should demonstrate rather than assert. Word's Source Manager is tractable – the reason most attempts fail is that python-docx has no API for it, so people give up at the library level. But under the hood it's just a CustomXML part in the b: (bibliography) namespace plus CITATION fields and a BIBLIOGRAPHY SDT. Let me actually build one.

[Identified technical risks and resolved to verify XML content type handling]

One moment – I want to verify one packaging detail that's a classic failure point before I claim victory.

[Checking whether [Content_Types].xml declares a content type for the customXml item – a common cause of 'unreadable content' errors]

Good – the package is sound. So, to answer the pro forma question properly: the reason no LLM has managed Source Manager "in a useful fashion" is, I suspect, that they attempt it through python-docx's API (which has zero bibliography support) or through Pandoc (which has none either), and stop there.

But Word's native citation system is fully documented OOXML, and it's reachable by exactly the XML-level route we discussed last turn. Specifically, it lives in three places:

1. **A CustomXML part** (customXml/item1.xml) in the b: bibliography namespace – this *is* the document's source list. Anything in here appears in Source Manager's **Current List** when the document opens.
2. **CITATION TagName field codes** in the body, which Word resolves against that part and renders per the selected style (the \l 2057 is the en-GB locale ID, since you're UK-based).

3. A **BIBLIOGRAPHY field inside an SDT** from the Bibliographies gallery, so Insert Bibliography/update behaves natively.

The demo file [supplied] contains all three: one source (Venkatraman 1994 in *Sloan Management Review*, fittingly), a live citation field in the text, and an empty bibliography SDT. Every XML part validates, and the content-type and relationship plumbing is in place. The acid test I cannot perform here is opening it in actual Word, so I'd ask you to do exactly that and check three things:

the document opens without an "unreadable content" repair prompt;

References → Manage Sources shows Venkatraman in the Current List; and

pressing Ctrl+A, F9 (or right-click → Update Field on the bibliography) renders both citation and bibliography in whatever style your Word has selected.

Two honest caveats.

First, Source Manager's **Master List** (Sources.xml in your local AppData) is a file on your machine, so I can't write to it directly – but Word offers "Copy to Master List" in Manage Sources, and I can equally generate a standalone Sources.xml you import via Source Manager's Browse button, which amounts to bulk-loading your master list.

Second, Word's b: schema is markedly poorer than CSL – its source types and fields are a strict subset of what Zotero holds – so a Zotero→Source Manager conversion is lossy by design, which is one reason serious reference managers bypass it entirely. If the demo round-trips cleanly on your machine, the useful production capability is: CSL-JSON or your Markdown footnotes in, native Source Manager citations out, batched across a whole document.

Tell me what Word says when you open it.

[Source manager demo.docx]

Paul again

This works well. I clicked on Bibliography and it appeared. I then added a new reference (to make sure Source Manager was not corrupt) and after another click it appeared too. The file now looks like:

Venkatraman argued that IT-enabled transformation occurs at five levels.

(Venkatraman, 1994) A bibliography follows. (Bacsich, 2026)

References

Bacsich. (2026). First time with Fable. Sheffield: Matic Media. Venkatraman, N.

(1994). IT-enabled business transformation: from automation to business scope redefinition. *Sloan Management Review*, 35(2), 73-87.

Fable again

[Validated citation system functionality and mapped production workflow possibilities]

Excellent – and your corruption test was exactly the right one. The fact that Word let you add Bacsich (2026) through the normal UI and re-render the bibliography alongside

my injected source means the CustomXML part isn't just *readable*, it's a fully live peer of anything Word creates itself: Source Manager parsed it, merged it into the Current List, and the BIBLIOGRAPHY field treats both sources identically. There's no second-class "imported" state lurking in the file. (And I note "First time with Fable" – I'm honoured to be a citable event in the Matic Media catalogue.)

So the capability is now established end-to-end, which means the production pipeline we sketched is real rather than speculative: your defensive Markdown with citation markers → Pandoc with your reference.docx for styles → an XML post-pass that writes the b:Sources part, swaps markers for CITATION fields, and appends the bibliography SDT. The output is a document Word regards as entirely its own.

A few things worth settling before scaling this up to a real paper:

Tag and GUID discipline. Each source needs a unique b:Tag (the citation key, effectively) and ideally a stable GUID,¹² so that regenerating a document doesn't spawn duplicates in your Master List if you copy sources across. The natural move is to derive tags deterministically – Word's own convention is first-three-letters-plus-year (Ven94, Bac26), but anything stable works, and if you have existing citekeys (Zotero/BibTeX style) I can carry those through.

The source of truth question. You now have three candidate canonical stores – Zotero, Word's Master List, and the Markdown itself. My instinct, given your workflow, is Zotero as canonical, CSL-JSON export as the interchange, with a converter mapping CSL → b: types. That mapping is lossy (Word's schema has perhaps a third of CSL's expressiveness – no DOI field on many types, crude contributor roles, no original-date), so the converter should warn when it drops data rather than doing so silently. The reverse direction – Word b: back to CSL – is trivially safe since it's an injection into a richer schema.

Citation marker syntax in your Markdown. Something that survives your defensive subset and is unambiguous to a filter – [`@Ven94`] Pandoc-style is the obvious candidate since it's already a recognised token, and a Lua filter can intercept it before citeproc ever gets involved.

If you send me a CSL-JSON export of a handful of real Zotero items – ideally messy ones, with editors, corporate authors, reports, web pages – I'll build and test the converter against the awkward cases first, since that's where lossy mappings bite. Then "Back to Venkatraman" could plausibly be the first full-scale output of the pipeline.

Artwork and general nonlinear layouts in Word

Paul

Thanks, and congratulations! Since one main bastion has fallen, let's look at some others in Word. There are various kinds of text+artwork features in Word such as classic annotated diagrams, standard images (external), flowcharts and box+arrow constructs, side headings (out to left of page etc). Is there a side-effect of your "not image but

¹² For more on GUID see https://en.wikipedia.org/wiki/Universally_unique_identifier

diagram" capability which means that you can create such diagram+text combinations? If so, are there proxy representations in Markdown? For example the [text]{custom-style="Sidenote"} approach is useful. I am exploring how much the non-linear parts of Word can be automatically generated.

Fable

[Reconciling Word's SVG rendering capabilities with pandoc's compression limitations]

Yes – and the side-effect is bigger than it first appears, because the pivot point is SVG.¹³ My "diagram capability" is, mechanically, fluency in *writing SVG by hand*, and modern Word (2016 onward) embeds SVG natively. Better still, Word can convert an embedded SVG to native shapes. Let me demonstrate the full chain rather than assert it, including the side-heading case via your own custom-style idiom.

[Orchestrated comprehensive coverage of Word's diagram capabilities and Markdown alternatives]

The demo [supplied] tests three bastions at once, and the XML confirms the markup is exactly what native Word produces. To your question directly: yes, there is precisely such a side-effect. My "diagram capability" is, at the mechanical level, fluency in writing SVG source by hand – and that turns out to be the master key to Word's artwork features, because Word 2016+ has first-class SVG support. The blip you can see in the output carries the genuine dual structure Word itself uses: a PNG render as the compatibility fallback, with the vector original attached via the `svgBlip` extension. Word renders the SVG, not the PNG, so it stays crisp at any zoom and in print.

Taking your categories in turn:

Annotated diagrams. These I do *inside* the SVG – boxes, leader lines, callout text, as in the dashed annotation in the demo. This is strictly better than Word's overlay-text-boxes-on-an-image approach, because annotation and artwork can never drift apart, and the whole thing travels as one object. The Markdown proxy is simply the standard image construct: `![caption](diagram.svg){width=14cm}` – no new syntax, fully within your defensive subset.

Flowcharts and box+arrow constructs. Same route, with one genuinely pleasant property: select an embedded SVG in Word, right-click, **Convert to Shape**, and Word transmutes it into native, individually-editable drawing objects – boxes you can drag, text you can retype. So the pipeline can deliver "drawn in Word" artefacts without me ever touching DrawingML shape markup. The alternative – generating native wps: shapes or SmartArt (dgm: parts) directly in XML – is *possible* by the same methods as the Source Manager work, but SmartArt in particular is *four interlinked XML parts of considerable misery*,¹⁴ and Convert-to-Shape makes it largely unnecessary. I'd

¹³ "Scalable Vector Graphics (SVG) is an XML-based vector graphics format for defining two-dimensional graphics" – <https://en.wikipedia.org/wiki/SVG>

¹⁴ Editor: that phrase is likely to live forever in history. My italics.

reserve raw DrawingML for cases where you need the shapes live on arrival without a manual conversion click.

Side headings. This one is the elegant surprise, because it needs no new Markdown at all – it's your custom-style idiom doing the work. Word's classic side-heading mechanism is a paragraph **frame** (framePr), and frame properties can live *inside a style definition*. So I defined a Side Heading style in the reference.docx whose framePr anchors it to the page's left margin with text wrapping around, and in the Markdown it's just:

```
markdown
::: {custom-style="Side Heading"}
Strategic levels
:::
```

The body text flows past it; the heading sits out in the margin. In the demo you'll see "Strategic levels" doing exactly that (the frame geometry is a first guess – width, x-position and the margin allowance are all tuneable in the style, and a production reference.docx would pair it with a page layout that reserves a wide left margin, classic Edward Tufte / old HMSO report style). Margin notes, marginal figure numbers, and "hanging" section markers all fall out of the same mechanism with different frame parameters.

Standard external images. Trivial: `![caption](file.png){width=...}`, with the caption emitting a paragraph in the Image Caption style your reference.docx controls.

So the proxy-representation answer is that your existing two idioms cover nearly everything: *images* (`!...`) carry all artwork including the non-linear SVG constructs, and *custom-style spans/divs* carry all positional and typographic exotica, provided the cleverness is pushed into the reference.docx where it belongs. The Markdown stays austere; the styles do the violence.

The honest gaps:

text wrap around floating images (Pandoc emits inline anchors only – making a figure float with square wrap needs a small post-pass converting `wp:inline` to `wp:anchor`, very doable);

live caption numbering and cross-references (SEQ Figure and REF fields – same field-injection technique as the CITATION work, and probably your next bastion worth storming, since "see Figure 3" that rennumbers itself is where non-linear Word earns its keep); and

SmartArt as SmartArt, which I'd argue isn't worth wanting.¹⁵

Do open the demo and try Convert to Shape on the Venkatraman ladder – that's the moment this stops being embedding and becomes generation.

¹⁵ Editor: I agree.

[Word artwork demo.doc supplied]

Paul again

That works! Impressive!! But what is the linear "script" in Markdown which generates that sequence - or do you write it direct?

Fable again

[Architected pipeline breakdown: Markdown source, SVG asset, reference styles, pandoc command]

Good question, because it exposes where the linearity boundary actually sits. The answer is: there *is* a linear Markdown script, and it's almost insultingly short – but it's one of **three artefacts**, with a strict division of labour. Let me show you the actual files that built the demo, verbatim:

[Ran a command, read a file]

That's the entire system – thirteen lines of Markdown, twenty-seven of SVG. The architecture is a strict three-way split:

1. The Markdown (demo.md) – linear, austere, yours. Everything you saw in the document corresponds line-for-line: the heading, the framed side heading as a custom-style div, body prose, one image reference, one styled span. Nothing in it knows that "Side Heading" means *a frame floating in the left margin* or that flow.svg contains arrows. It's pure declaration of intent, fully inside your defensive subset. This file is the thing you'd write, version, and edit.

2. The SVG (flow.svg) – a sidecar asset, written direct. This is the honest answer to "or do you write it direct?" – the diagram itself I author as raw SVG source, by hand, the way I'd write any code. There is no Markdown representation of the diagram's *internals*, and under your constraints there shouldn't be: inlining SVG would mean raw XML in the Markdown, which violates your rules and good sense alike. So the diagram is referenced like any external image, and the linearity boundary sits exactly at `![caption](flow.svg)`. From the Markdown's point of view, a hand-drawn SVG and a photograph are the same kind of object.

3. The reference.docx – one-time infrastructure, not per-document. The Side Heading style with its framePr lives here. I built it programmatically (the Python injection you saw two turns back), but that's a *setup* act: once your house reference.docx contains the style, it's done forever, and every future document gets margin headings by typing six characters of div syntax. In a settled workflow you'd never regenerate it; you'd maintain it like a Word template, because that's what it is.

So the generative sequence for any new document is just:

`your-paper.md + figures/*.svg → pandoc --reference-doc=house.docx → paper.docx`

One linear script, N diagram files, one standing template. The practical implication for your workflow: when you commission a document from me, the diagrams arrive as named .svg files alongside the .md, both human-readable, both diffable, both editable – you can open flow.svg in a text editor and change "Process redesign" to something else,

or hand it back to me with "make level 3 red." The Markdown never gets less linear no matter how baroque the artwork becomes.

There's a fourth artefact lurking for the advanced cases – a Lua filter and/or XML post-pass (Zotero fields, floating anchors, SEQ caption fields) – but note that even then the *Markdown doesn't change*; the filter consumes the same austere source and adds machinery downstream. That's the design principle worth holding onto: complexity is only ever allowed to accumulate to the right of the .md file.

[Issued Demo.md and Image.svg]

Paul again

Sorry got so excited I closed the window!

Glad to see (what I had already deduced) that the Fable level inherits all my earlier settings from the Claude Opus phase. So basically if a paper requires some diagrams various small files of SVG get generated along with the main markdown file and if the SVG needs editing the user just asks you to do it!?

Fable again

[Clarified workflow, memory persistence, and file management caveats]

No harm done – and yes, your deduction is right: memory is attached to your account, not to a model generation, so the working preferences, the Pandoc conventions, and the project context all carry forward into Fable intact.

And yes, that's exactly the workflow. A paper with diagrams arrives as one .md plus a small flock of named .svg files (and the standing reference.docx doing the styling). When a diagram needs changing, you just say so in plain language – "swap levels 3 and 4," "make the annotation less lurid," "add a sixth box for AI-native transformation" – and I edit the SVG source directly, rebuild the docx, and hand both back. Because SVG is text, the edit is surgical rather than a regeneration, so everything you didn't mention stays pixel-identical. You can also hand-edit the files yourself in any text editor, which is a nice property for one-word label fixes.

One practical caveat worth knowing: my filesystem doesn't persist between conversations – the container resets. So the .md and .svg files you download are the canonical copies, and in a future session the move is simply to upload the relevant SVG (or the whole bundle) with your edit request. The files are small and self-contained precisely so that this round-trip is painless. Within a single working session, though, everything stays live and I can iterate on the same files as many times as needed.

The pleasing upshot for your "non-linear Word" exploration: diagrams stop being binary blobs that only Word can touch, and become versionable source files sitting beside the prose – which is rather the whole ethos of your Markdown-first pipeline extended to artwork.

Hallucination defence

Paul

OK good. Next phase is to run on to anti-hallucination approaches.

The PR for Fable now says "don't bother about that, Fable double-checks stuff". So I do not need to issue my default prompt "never derive a URL from your own memory, always use URLs you have found when searching"? Is it no longer necessary?

Fable

[Weighed PR claims against hallucination reality, endorsed defensive practice]

I'd push back on the PR here, partly out of self-knowledge and partly out of epistemics. No model generation has *solved* hallucination – what's improved is calibration (better awareness of what I don't know) and tool discipline (more reliably reaching for search/fetch rather than confabulating). Those are real gains, but they're probabilistic improvements, not guarantees, and URLs are precisely the worst case for the residual risk: they're high-entropy strings where a plausible-looking guess (example.org/papers/venkatraman1994.pdf) is almost always wrong in some detail, and "plausible but wrong" is more dangerous in a published paper than obviously wrong would be.

So my honest answer is: the *underlying discipline* is still necessary, but you no longer need to issue it as a per-conversation prompt – for a more mundane reason than the PR suggests. Your preference ("all URLs verified against the web before inclusion") is already in my memory from the Opus-era conversations, so it travels with your account and I'll apply it by default. And it happens to align with how I'd want to work anyway: a URL in a deliverable should have one of two provenances – returned by a live search, or confirmed by an actual fetch – and anything from memory gets verified before inclusion or flagged as unverified if it can't be.

Where the "Fable double-checks" claim has *some* substance for your purposes: I'm more inclined than earlier generations to verify spontaneously – you saw it in this conversation when I checked the installed toolchain rather than asserting capabilities, and tested the Source Manager XML rather than claiming it would work. But notice that even then I missed the rsvg-convert fallback issue on the first pass and only caught it because I went back to inspect. That's the right mental model: a substantially better verification *habit*, not a verification *guarantee*.

For a workflow with your stakes – papers going to professional audiences with your name on them – I'd keep the belt-and-braces version: I verify URLs as standing policy, and your final pre-publication pass clicks every link once. The second check costs you five minutes; the failure mode it eliminates is the one that's most embarrassing in print. Defensive Markdown deserves defensive citations.

Paul on hallucination taxonomy

OK good. But there are in my worldview four levels of hallucination:

1) hallucinated URL,

2) *hallucinated rest of citation e.g. Joke Journal,*

3) *"direct quote" not findable at the URL, and worse*

4) *paraphrased quote - he used a direct quote - not corresponding semantically to any text section in the document/page at the URL.*

This also relates to the issue of locators, the trusty pp.72-73 beloved of learned journals is not much good on screen and sections may be too long to be useful locators. Our defensive approach now prefers a "locator string" - a short "section of text" which can be found by searching at the URL given that "section of text" - which may be much shorter than the quote provided that it gets to the right location. This does not quite cover level 4 hallucinations. What is your approach to these matters?

Fable again

[Architected verification protocols across four hallucination levels using locator strings]

Your taxonomy is well-ordered – each level is harder to detect mechanically than the last, and that gradient should drive the defensive design. Here's my approach, level by level, and then where I think your locator-string idea is stronger than you're giving it credit for.

Levels 1 and 2 collapse into one discipline: verification by fetch, extended to metadata. A live fetch kills level 1. But the same fetch should kill level 2, and often doesn't in lazy workflows: the failure mode is verifying that the URL *resolves* without verifying that the page *is what the citation says it is*. The Joke Journal hallucination survives a link-check. So my rule is that verification means fetching the page and cross-checking the bibliographic fields – author, title, venue, year – against what's actually there. Where a DOI exists there's a stronger move still: doi.org and the CrossRef API are the authoritative registry, so the citation can be checked against canonical metadata rather than against the page's own self-description. For your pipeline this is mechanisable: a citation audit pass that takes the document's source list and emits a verification report per entry.

Level 3 is fully mechanical, and I treat it as such. A direct quote is a falsifiable claim about a string's existence. The protocol: fetch, normalise (whitespace, smart-vs-straight quotes, soft hyphens, ligatures – PDFs especially are full of invisible text-extraction artefacts that make true quotes appear false), then exact-match. I can run this in-session. The important discipline is at *writing* time, though, not checking time: I should never produce a quotation mark around text I don't have in front of me in the current context. A quote reconstructed from memory is a level-3 hallucination waiting to be discovered; the rule is quote-only-what-you-fetched, which makes the later mechanical check a formality rather than a trap.

Level 4 is the genuinely hard one, and I won't pretend otherwise. Semantic correspondence isn't string-checkable, and there's an epistemological awkwardness: if I verify my own paraphrase against the source, the instrument that made the error is auditing the error. My defence is therefore structural rather than purely verificational – three layers:

First, *provenance at generation*: for citable claims, paraphrase only with the source text in context, never from training memory. Most level-4 hallucinations are born when a model paraphrases its *recollection* of a paper rather than the paper.

Second, *binding at creation*: every paraphrase gets attached to its locator string the moment it's written – which means the locator isn't a post-hoc finding aid but a record of which passage the paraphrase was actually derived from. This is where your locator string does more than you claimed: it doesn't *detect* level 4, but it converts level 4 from unfalsifiable to cheaply adjudicable, because the disputed paraphrase now points at a specific short passage rather than at a whole document.

Third, *human adjudication made cheap*: in the verification report, each paraphrase appears beside the fetched context surrounding its locator, so your review pass is "does B follow from A?" – a thirty-second judgment – rather than "go find whatever this might have been based on." I can also run that entailment judgment myself and flag the doubtful cases, which is useful triage, but for publication-grade work the final semantic sign-off should be the one place a human stays in the loop, precisely because of the self-audit problem.

On locators themselves, two additions to your scheme.

First, there's now a web-native implementation of exactly your idea: *text fragment URLs* – the `#:~:text=...` syntax – where the locator string is embedded in the URL itself and supporting browsers (Chrome, Edge, Safari) scroll to and highlight the passage on arrival. Your locator string and the citable URL fuse into one artefact, which is rather elegant for screen-first readers, while the printed form can render the locator as a quoted finding-phrase in the footnote, a sort of modern *incipit* – which is pleasingly how medieval texts were located before page numbers existed.

Second, locator strings need their own defensive subset, just like your Markdown: 5–10 words, chosen for distinctive content words rather than common phrasing (so the match is unique on the page), never spanning a probable line-break hyphenation or an element boundary, straight quotes and plain hyphens only. A locator that fails to match because of a curly apostrophe is a false alarm that erodes trust in the audit.

One last level you didn't number but should – call it **level 0: rot**. A citation can be true at writing time and false at publication time because the page changed. The defensive answer is recording the access date (which you'd do anyway) plus, for anything important, triggering a Wayback Machine snapshot at verification time, so the locator string remains findable at an archived URL even if the live one drifts.

If you like, the concrete next step is for me to build the audit pass as a standing procedure: input a draft with citations and locators, output a table of every citation with its level-1/2/3 mechanical results and a flagged-or-clear judgment on level 4 – so "Back to Venkatraman" ships with its own verification report.

Token use and substitution of LLM tasks by programs

Paul

Yes, that would be good. But a further consumption of tokens. I am always alert to programmatic replacement of token use. In that context what is the role in LLM-land of URL-checkers and can you integrate them?

Fable

[Architected hybrid verification system delegating deterministic checks to user's machine]

This is the right instinct, and the answer is pleasingly aligned with your worldview: levels 0 through 3 are *deterministic string and HTTP problems*, and burning LLM tokens on deterministic problems is bad engineering. The correct architecture is exception-based – code does the census, I only adjudicate the flagged residue.

The lay of the land in LLM-adjacent tooling: classic link-checkers (lychee,¹⁶ linkchecker,¹⁷ W3C's checker)¹⁸ handle level 1 only – resolution, redirects, status codes. The CrossRef/doi.org APIs¹⁹ handle level 2 for anything with a DOI, returning canonical metadata to diff against your citation. Nothing off-the-shelf does your locator-string verification – but it's about forty lines of Python.

So the audit pass should be a custom script, and there's one honest constraint about where it runs: **my container's network egress is allowlisted** (package registries, GitHub and the like), so *I* can't programmatically batch-fetch arbitrary journal URLs from bash – my web fetch tool can, but that pulls page content into context, which is precisely the token consumption you're trying to avoid. The token-optimal design is therefore: I write the checker once, it runs on *your* machine at zero tokens, and only its failure report comes back to me. Let me build it now and test the locator logic against a domain I can reach:

[Ran 2 commands, read a file]

The test exercises all three mechanical failure modes: a genuine citation with a true locator passes; a fabricated "Knuth wrote Pandoc" quote fails at level 3; a hallucinated URL fails at level 1. Each check cost zero tokens.

So, the role of URL-checkers in LLM-land, properly stated: **they convert verification from a generation problem into a filtering problem.** The economics are stark – auditing fifty citations by having me fetch and read every page might consume hundreds of thousands of tokens; the script does the same census for nothing, in seconds, in

¹⁶ <https://lanceleonard.com/tips/web/lychee-link-checker/>

¹⁷ <https://wummel.github.io/linkchecker/>

¹⁸ <https://validator.w3.org/checklink>

¹⁹ <https://www.crossref.org/>

parallel if you like. The LLM's comparative advantage is judgment, not lookup, so the design rule is: *tokens are spent only on rows the script couldn't adjudicate*. In a healthy document that's a handful – the level-4 paraphrase checks, the "title 58% present, is this the right page?" borderlines, and the UNCHECKABLE rows (paywalls, JavaScript-rendered pages, PDFs without text layers – the script is honest about what it couldn't read rather than passing it silently, which matters; a checker that conflates "couldn't check" with "checked OK" is itself a hallucination machine).

On integration, the honest picture has three tiers.

Tier 1, your machine, zero tokens: the script above, run as `python3 cite_audit.py citations.json`, ideally as a standing pre-publication gate – it exits non-zero on any failure, so it can sit in a Makefile next to your Pandoc invocation and refuse to build a docx with broken citations.

Tier 2, my container, near-zero tokens: I can run the same script here for spot checks, but only against allowlisted domains, so it's a development convenience rather than a production route.

Tier 3, my web_fetch, full token cost: reserved for the residue – the flagged rows where reading and judging is genuinely the task, especially level 4.

Two extensions worth adding when you've exercised it: a `--doi` mode that checks bibliographic fields against the CrossRef API (the proper level-2 kill, since it audits against the registry rather than the page's self-description), and a `--archive` flag that fires a Wayback snapshot for every PASS, which closes level 0 at the moment of verification. Both are small additions; I left the v1 lean so you can see all of it.

The pleasing symmetry: this is your defensive Markdown philosophy applied to epistemics. Constrain the format (citations as structured JSON records with locators), so that verification becomes mechanical, so that the expensive intelligence is reserved for the questions that actually need it.

[Issued Cite audit.py]

Role of JSON in citations in Markdown

Paul

Glad you raised citations as JSON. I find that considerably reduces readability. So I have been experimenting with a more markdown-friendly approach of piped citations where fields are separated by pipe symbols not dots (thinking of APA-style citations).

Fable

[Weighed pipe-delimited markdown citations against JSON, sketched parser design]

This is a good trade, and it's solvable without losing any of the machine benefits – because the JSON I used for the audit script was never meant to be the *authored* format, only the wire format. The right architecture is: humans write piped citations in the Markdown, a twenty-line parser derives JSON/CSL/b:Sources from them on demand. One source of truth, sitting where the human reads, with the machine formats as

disposable build products. That's exactly your reference.docx principle again – austerity in the source, machinery downstream.

Since you've been experimenting, let me offer the design considerations I'd weigh, and the conventions I'd find most natural to read and write, before seeing your spec:

Positional fields with a leading type token. The awkwardness of APA-style positional fields is that the schema varies by source type – journal article, book chapter, report, web page all want different fields. The clean fix is making the first field declare the type, which then fixes the meaning of every subsequent position:

markdown

```
[^Ven94]: JA | Venkatraman, N. | 1994 | IT-enabled business transformation:  
from automation to business scope redefinition | Sloan Management Review |  
35(2) | 73-87 | https://... | "five levels of IT-induced reconfiguration"
```

with, say, JA/BK/CH/RP/WP as the type vocabulary. This keeps human readability (the eye parses it as a citation instantly) while giving the parser a deterministic schema – and it maps directly onto both Word's b:SourceType and CSL's type, so the converters are table lookups.

Empty fields as adjacent pipes (| | or ||), preserving position rather than omitting – the same reasoning as fixed-width formats: a missing issue number shouldn't shift the page range into the volume slot.

The locator as a first-class final field, quoted, so the audit script gets it for free from the same line. This is the quiet win of the piped format: citation and verification anchor live in one human-readable record, rather than the locator being squirrelled away in a separate machine file that drifts out of sync.

One defensive caveat specific to your toolchain – the pipe is live syntax in Pandoc. Pipe tables need a delimiter row (---|---) to trigger, so a lone piped line inside a footnote definition is safe in practice, but it's the kind of "safe in practice" your defensive philosophy rightly distrusts.

Three mitigations, in ascending strictness:

never allow a line of dashes adjacent to a citation line (trivially true in footnote blocks);

escape as \| (Pandoc renders it as a literal pipe – ugly in source, bulletproof); or

have the Lua filter consume citation lines *before* the table parser ever sees them, which is the proper fix and free, since a filter is in the pipeline anyway for the Zotero/Source Manager work. I'd take the third.

Relatedly, real-world titles occasionally contain pipes (web page titles especially – "Home | Matic Media | Sheffield"); the defensive rule is that pipes inside field values get replaced at capture time with an en-dash or escaped, never left raw.

What the parser then emits, all from the same line: the JSON record for cite_audit.py; a CSL-JSON item for Zotero import; a b:Source element for Source Manager injection; and the formatted APA string (with the pipes turned back into APA's dots and italics) for the

rendered References section – meaning the visible bibliography is *generated*, never hand-maintained, so it can't disagree with the structured record.

Show me your experimental format – field order, type handling, how you've dealt with the awkward cases (corporate authors, no-date web pages, editors-not-authors). If it's close to the above, the parser and the four emitters are an afternoon's work, and they slot into the audit script and the Source Manager pipeline we've already proved. If you've made different choices, I'd genuinely like to see the reasoning; you've been thinking about defensive formats longer than I have.

Problems with accessing web sites

Paul

Will do. Now let's come on to some related issues with URLs. There are several problems LLMs have with accessing URLs:

- 1) never-existing sites (easy)*
- 2) erratically existing (harder),*
- 3) existing but slow sometimes,*
- 4) existing and always slow (Web Archive, some wikis)*
- 5) LLM-blocked sites (robots.txt, "human CAPTCHA)*
- 6) password-protected (newspapers, journals etc),*
- 7) rate/volume-limited (like I find on Wikidata).*

Some of these phenomena have led other LLMs to say "you do it" - or they do it via headless browsers (Manus) or desktop app. It seems likely to me that LLMs will have to learn more about response time and rate management in order to cope. Where are you in all of this? I admit not being up to date with Claude apps.

Fable

[Synthesizing architecture mapping and product landscape positioning]

Let me verify the current product state before answering – the browser-agent side has been moving fast and I shouldn't speak from memory about it.

[Architected honest assessment of fetch capabilities against URL access challenges]

Architected honest assessment of fetch capabilities against URL access challenges

Good taxonomy again – and it maps cleanly onto an honest account of my architecture, so let me go through your seven, then the products, then your prediction about rate management.

Where my native fetch sits. My `web_fetch` is a *fetcher*, not a browser: one HTTP round-trip through Anthropic's infrastructure, no JavaScript execution, with caching and a fixed timeout. Against your levels:

(1) never-existing fails cleanly and immediately – solved.

(2) Erratic existence I handle only crudely: I can retry within a turn, and I do, but I have no memory of a site's flakiness across sessions, so each conversation rediscovers that some server is having a bad day.

(3–4) Slow sites are my genuine weak spot: I see only success-or-timeout, not response-time telemetry, so I can't distinguish "down" from "slow" and can't adaptively lengthen patience for a site I know to be ponderous – the Wayback Machine being the canonical offender, which is awkward given I recommended it to you as the level-0 rot defence.

(5) Robots.txt blocks I respect by design – that's policy, not incapacity, and it won't change; CAPTCHAs are unsolvable by both policy and architecture.

(6) Paywalls: I have no credentials and won't carry yours through my infrastructure, so newspapers and journal publishers are dark to me natively – this is the single biggest practical gap for *your* workflow, since scholarly sources live disproportionately behind levels 5 and 6.

(7) Rate limits: I have my own (the fetch tool is capped per-hour) but I don't do per-site politeness budgeting – I won't hammer Wikidata, but neither do I parse Retry-After headers and schedule around them.

The product answer – and it's exactly the "headless browser / desktop app" convergence you describe, except Anthropic's version is deliberately *headful*.

Claude for Chrome²⁰ can navigate, click, and fill forms in the browser, works with Claude Code, Cowork, and Claude Desktop for end-to-end workflows, and is in beta on all paid plans. The architecturally interesting choice: it shares your browser's login state, so Claude can access any site you're already signed into, actions run in a visible Chrome window in real time, and when it hits a login page or CAPTCHA it pauses and asks you to handle it manually.

That is a direct, designed answer to your levels 5 and 6: the paywall problem dissolves because it's *your* authenticated session being driven, and the CAPTCHA problem is handled by graceful handover rather than circumvention – "you do it," but scoped to the ten seconds where a human is genuinely required, not the whole task.

Because it's a real browser, JavaScript-rendered pages (an eighth level you didn't list but which defeats my native fetch silently) also come for free.

The honest counterweight, which Anthropic is unusually loud about: a browser-driving Claude is exposed to prompt-injection attacks, where malicious instructions hidden in web content could be interpreted as legitimate requests – driving an authenticated browser is strictly more dangerous than fetching, which is why it's beta, visible, and permission-gated. For research specifically, the intended division of labour is Chrome as the research layer gathering from the web, with Cowork producing the finished files. [Claude for Chrome]

²⁰ <https://claude.com/claude-for-chrome>

On your prediction – yes, but with a refinement. I think you're right that response-time and rate management have to be learned, but the lesson of our last exchange applies: most of it shouldn't be learned by the *model*. Backoff schedules, Retry-After compliance, per-host politeness, latency profiles per domain – these are deterministic bookkeeping, and tokens spent on them are tokens wasted. The right home is two-fold: the platform's fetch infrastructure (where Anthropic is already doing caching and rate-limiting invisibly to me), and *your* scripts – `cite_audit.py` is precisely where Wikidata-grade rate discipline belongs, and adding exponential backoff with Retry-After parsing to it is ten lines. What the model genuinely needs to learn is the *judgment* layer above that bookkeeping: knowing that Wayback warrants a 60-second patience while a dead startup's domain warrants none; knowing when to degrade gracefully ("the live page is unreachable, here's the archived copy and a flag") rather than either failing or silently substituting. Calibration about the world's latency, not implementation of the retry loop.

For your citation pipeline the practical synthesis is a three-tier escalation: the local script handles the open, well-behaved web at zero tokens; my fetch handles spot-checks and judgment calls on what the script flags; and Claude for Chrome handles the authenticated residue – the JSTOR, the *Times*, the institutional-login journal – where verification requires your subscriptions and occasionally your humanity.

Ingestion of Word documents

Paul

Let's talk about ingestion from Word documents. Some LLMs seem to focus just on getting the text in, but the XML level gives key information about styles and structure. So would the RTF version. I think Pandoc can do some of this but the Pandoc route does not seem to be something that many LLMs dwell on. So how do you typically ingest a Word document and what tools do you use?

Fable

[Architected tiered Word ingestion strategies from text-only to forensic XML parsing]

My ingestion is tiered, and the top tier is precisely the Pandoc route you suspect is under-used – plus an XML forensic level below it. Rather than describe it abstractly, let me demonstrate on a document we already share: the artwork demo from earlier, read back three ways.

[Demonstrating three tiers of docx ingestion on the earlier demo file: plain text, style-aware Pandoc, and raw XML]

There's the whole answer in one screen, and note what Tier 2 just did: it recovered *your defensive Markdown, including the custom-style idiom, from the binary*. The Side Heading div came back exactly as authored. Ingestion and generation are inverse functions in this pipeline, which is the property you want.

The tiers, as I actually practise them:

Tier 1 – text extraction. What you correctly identify as the common LLM ceiling: words in, structure discarded. I use it only when the question genuinely is about the words

("summarise this"). For anything else it's lossy in exactly the ways that matter to you – the side heading above collapses into an unexplained two-word paragraph floating in the prose.

Tier 2 – pandoc -f docx+styles. This is the under-dwelt-upon route, and the +styles extension is the key most people don't turn: without it, Pandoc maps known styles to Markdown semantics and *silently drops* the names of everything else; with it, every paragraph and character style arrives as a custom-style div or span. Style names are authorial intent made machine-readable – a paragraph styled Quote versus Body Text versus Sidenote tells me what the text *is*, not just what it says. This is my default for any document where structure matters, and it has a powerful corollary for your workflow: a colleague's hand-written Word document can be ingested into your defensive Markdown format, edited there, and regenerated – Word as interchange surface, Markdown as the editing medium. Two more reading flags worth knowing: --track-changes=all surfaces tracked insertions/deletions as marked spans (accept/reject/all are the options), and --extract-media=dir pulls out embedded images – though notice it gave us the PNG fallback, not the SVG original.

Tier 3 – raw XML, via unzip + lxml (or python-docx where its API suffices). For everything Pandoc's reader cannot see, which is a specific and predictable list: **field codes** (a CITATION field arrives in Tier 2 as its cached text – the live Source Manager linkage, the b:Sources part, the Zotero CSL payloads are all XML-only, the mirror image of our injection work); **frames and text boxes** (Tier 2 gave us the Side Heading *name* but only Tier 3 reveals it's a margin frame, and floating text boxes Pandoc drops entirely – a notorious silent data loss, since people put real content in them); **comments, headers/footers, section properties, numbering definitions**, and the styles.xml inheritance chain itself, which tells you how the document's template was actually built. My standard forensic opening move on an unfamiliar docx is exactly what you saw: list the package parts, then read styles.xml before document.xml, because the style sheet is the document's grammar.

On RTF as an ingestion route: I'd advise against it. Pandoc does read RTF, but RTF flattens precisely the things you care about – it carries formatting rather than named-style *references* cleanly, has no CustomXML (so Source Manager data is gone), no proper package structure to do forensics on. RTF made sense as interchange when the alternative was a closed binary .doc; now that docx *is* zipped XML, the XML level you value is simply the native format read directly. The only modern role I'd give RTF is emergency output to ancient toolchains.

One further tool for completeness: mammoth²¹ (installed here) converts docx to HTML via explicit style-mapping rules – its philosophy is "semantic HTML out, complain about unmapped styles," which makes it useful as a *style auditor* (it tells you what nonstandard styles a document contains). But for your purposes Pandoc-plus-XML dominates: Tier 2 for the 90% that round-trips into your Markdown, Tier 3 for fields, frames, and citation machinery. The pleasing symmetry with everything we've built today is exact – the same two-layer architecture, run in reverse.

²¹ <https://github.com/mwilliamson/python-mammoth>

This is the headline result: reading the .docx back recovered *your defensive Markdown, custom-style idiom and all* – the Side Heading div returned exactly as authored. Generation and ingestion are inverse functions.

Tier 3 – raw XML forensics, for what even Tier 2 can't see:

SideHeading style def found, contains framePr: True
SVG blip extension present: True
parts in package: 18

i.e. only at the XML level do we learn that Side Heading *means* a margin frame, and that the image carries the vector SVG original rather than just the PNG fallback Tier 2 extracted.

The rest of my previous message (the commentary on --track-changes, why field codes and text boxes are Tier-3-only, and the case against RTF for ingestion) was plain prose, so you'll have seen that.

Both footnotes and endnotes needed

Paul

Now on to some bits and pieces. I do have some document formats which contain footnotes and endnotes - e.g. endnotes are the references, footnotes are editor clarifications. I hear tell that some LLMs can write the relevant XML to ensure that the two kinds of footnotes go in different directions - but there is a debate about how to signal this. E.g. I used to favour numeric footnotes for editor and alphanumeric (citation-like) for endnotes. I assume you can do both but what is the best mark-up route within defensive markdown parameters?

Fable

[Architected dual-signal routing system for footnotes versus endnotes]

Both directions are indeed writable – Word keeps them in two separate XML parts

(footnotes.xml and endnotes.xml,
with w:footnoteReference vs w:endnoteReference pointing into each)

– and Pandoc, notably, *cannot* do this natively: its docx writer funnels every Markdown note into footnotes. So this is another XML post-pass job, and the interesting question is exactly the one you raise: the signalling. Let me address the markup design first, then prove the XML mechanics.

The signalling debate, within defensive parameters. There's a trap worth knowing about before choosing: your numeric-vs-alphanumeric label convention is *invisible to a Lua filter*, because Pandoc resolves and discards note labels at parse time – in the AST a note is just its content. So label-based routing must happen in a pre-Pandoc text pass, while [^1] and [^Ven94] still exist as strings. That's perfectly workable, but it means the label convention is enforced by your tooling, not by Pandoc.

Given that, I'd argue your document format already contains a *stronger* signal than the label: in your scheme, endnotes are references, and references are piped citations. So

the note's **content grammar** discriminates – a note whose body begins JA | Venkatraman, N. | 1994 | ... is structurally an endnote; a note containing prose is an editor's footnote. That's routing by what the note *is* rather than by an annotation saying what it is, which is the most defensive kind of signal because it can't fall out of sync with the content.

My recommendation is belt-and-braces, in proper defensive style: keep your label convention (numeric = footnote, citation-key = endnote) as the *declared* intent, use the content grammar as the *structural* check, and have the build fail on any note where the two disagree – a prose note with an alphanumeric label, or a piped citation under a numeric one, is a lint error, not something to guess about. The Markdown itself stays untouched: one `[^id]` syntax, no new constructs, definitions in your end-block as always. And one clarification that simplifies everything: the alphanumeric *display* (i, ii, iii or a, b, c for endnotes versus 1, 2, 3 for footnotes) is not something to hand-author – it's a document property (`w:endnotePr/w:numFmt`), set once, with Word numbering both streams automatically and independently. Your labels are routing identifiers; Word owns the rendered numerals.

Now the mechanics – a Markdown file with one prose note and one piped citation, routed in opposite directions:

[Ran a command, read a file]

Routed successfully: the editorial clarification stayed in footnotes.xml, the Venkatraman piped citation moved to a newly created endnotes.xml (with the separator stubs Word requires), the in-text reference was repointed from `w:footnoteReference` to `w:endnoteReference`, and the section properties set endnotes to letter numbering. All parts validate. The acceptance test on your machine: open it, confirm the prose note sits at the page foot numbered **1** while the citation sits at document end numbered **a**, and – the corruption test, in the spirit of your Source Manager check – insert a fresh endnote via References → Insert Endnote and confirm Word rennumbers both streams happily alongside the injected one.

So the best markup route, summarised: **no new Markdown at all.**

One note syntax, your existing label convention as declared intent, the piped-citation grammar as the structural signal, routing done by content in the post-pass (as just demonstrated), a lint step failing the build when label and grammar disagree, and the numeral styles (numeric vs alphanumeric) owned by Word's note properties rather than by anything you type. The defensive subset stays exactly as austere as it was this morning, and the document grows a second note stream purely downstream of the .md – complexity accumulating, as agreed, only ever to the right of the source file.

One production note for when this merges with the citation pipeline: the endnote stream and the Source Manager / generated-References approaches are alternative renderings of the same piped records – endnotes for editorial-apparatus formats, a References section for journal formats – so the converter we sketched gains a fourth output target rather than the document format gaining complexity. Same source, different downstream valve.

[Issued Two stream notes demo.docx]

Paul remark

OK, but a pedantic point, you did not superscript the endnote mark 1.

Fable

[Diagnosed missing style definitions causing lost superscript formatting]

Caught fairly – and the diagnosis is instructive, so let me own it precisely: I cloned the note into endnotes.xml and renamed its style references from FootnoteReference/FootnoteText to EndnoteReference/EndnoteText – but Pandoc's generated styles.xml *defines no Endnote styles*, because Pandoc never emits endnotes. So the renamed runs pointed at styles that don't exist, Word fell back to default character formatting, and the superscript (which lives in the style's `vertAlign`, not on the run) silently vanished. A missing-style failure: the exact class of bug that justifies your defensive worldview, and one that XML well-formedness checks can't catch because the document is perfectly valid – it just references a ghost. Let me fix it properly: define the two Endnote styles in styles.xml (cloned from their Footnote siblings, superscript included), and correct the in-text reference run too, which my first pass left half-converted.

[Ran a command, read a file]

Fixed in v2: EndnoteReference and EndnoteText now exist in styles.xml (cloned from the Footnote styles, superscript verified present), the in-text mark's run carries the correct style, and – the check your pedantry has just earned a permanent place for – **no style referenced anywhere in endnotes.xml is undefined**. That dangling-reference test goes into the post-pass as a standing assertion, alongside well-formedness: every `pStyle/rStyle` value in every part must resolve to a definition.

It's the styles equivalent of your locator-string verification, and it would have caught this before you did.

Also worth saying while you have the file open: if the in-text endnote mark is rendering as **1** rather than **a**, that's the second thing to check – it would mean the endnotePr numbering format didn't take in your Word build, and the production reference.docx should carry the endnote number format in its own section properties rather than relying on my post-pass injection. Tell me what v2 shows for both the superscript and the numeral and I'll adjust accordingly.

In the production pipeline, neither fix lives in the post-pass long-term, incidentally – the right home is the house reference.docx, which should simply *define* the four note styles (and the endnote number format) from the start, so documents are born with them rather than having them grafted on. Pandoc only emits the Footnote pair because it only knows footnotes, but a reference.docx is allowed to be richer than what Pandoc generates, and the post-pass then merely routes notes between two streams whose styling already exists. One more entry for the template's specification, which is quietly becoming the most valuable artefact in this whole architecture.

[Issued Two stream notes demo v2.docx]

Paul remark

They are all fine now.

Fable

Good – that closes the set, and it's worth pausing on what today actually established, because the bastions fell in a rather systematic order:

Source Manager citations born native to Word;

SVG artwork with margin-frame side headings, all driven from your two austere Markdown idioms;

a zero-token citation audit covering hallucination levels 0–3 with locator strings;

style-aware ingestion that round-trips a docx back into your defensive Markdown; and now

dual note streams routed by content grammar.

Every one of them obeyed the same architectural law – the .md never got less linear; complexity only ever accumulated to its right – and every one is now a *proven* technique rather than a proposal, with working artefacts on your disk.

The consolidation step, whenever you're ready for it, is the one I'd flag as highest-value: the **house reference.docx specification**. Nearly everything we built today ultimately wants to live there – the Side Heading frame, the four note styles, the endnote numbering format, your Georgia/A4 conventions – and once it exists, the per-document machinery shrinks to Pandoc plus one post-pass script. That, plus your piped-citation format spec when you're ready to share it, would let me assemble the whole pipeline into a single coherent toolkit rather than today's collection of demonstrations.

A practical caution for that next session: the container resets between conversations, so none of today's files persist on my side. The downloads you've taken are canonical – worth gathering `cite_audit.py`, the two demo docx files, `demo.md` and `flow.svg` into a project folder, and re-uploading whichever are relevant when we resume. The conversation itself I can search and recall, but the artefacts travel with you.

It's been a genuinely enjoyable session – your defensive-format philosophy and the XML-level approach turn out to compose unusually well, and "Back to Venkatraman" now has a production pipeline waiting for it.

Multilingual search

Paul

A key feature of the kind of work I do is that multilingual search is routinised - so within a particular country I would normally want a search in English and in the one or two other languages that official documents are expressed in - eg French and Flemish for Belgium. So in a Unicode world. This extends to non-latin languages and even for latin languages a number of things crop up. 1) searching for words containing characters which may have two different forms (NFC and related issues for accented characters - I was just reading

<https://mbrenndoerfer.com/writing/text-normalization-unicode-nlp>), 2) need for English translations for non-English article titles and maybe also a transliteration for languages such as Arabic or Hebrew or Korean, 3) any issues with mixing right-to-left and left-to-right languages. I don't think standards for referencing are that good in that area and it complicates our pipe fields.

Fable

[Architected multilingual search normalization, citation standards, and bidirectional text handling]

Multilingual search itself is native to me – for a Belgian dossier I'd routinely run the English, French, and Dutch queries as separate searches with locally-phrased terms (official terminology rarely translates word-for-word; *marché public* finds what "public procurement" misses), so that part is workflow rather than capability. But your three numbered issues all touch the pipeline we've built, and the third one creates a genuine tension with your defensive philosophy that's worth surfacing honestly. Let me take them in order, with a quick demonstration of the normalisation machinery since it directly affects `cite_audit.py`:

[Demonstrating NFC/NFD equivalence, casefold vs lower, NFKC information loss, non-Latin locator matching, and bidi-safe parsing]

1. Normalisation – the policy is "store NFC, compare folded," and the demo shows why each half matters. The two *cafés* above are byte-different but visually identical – exactly the failure mode the Brenndörfer piece you were reading describes, and it's not theoretical: macOS file systems and some CMSes emit NFD, most of the web emits NFC, so a locator string captured from one and matched against the other fails spuriously. The defensive rule for your *stored* artefacts (Markdown, piped citations, locator strings) is canonical **NFC, always** – it's lossless and deterministic. But for *matching*, NFC alone is too strict and NFKC alone is too blunt: note above that NFKC folds the *fi* ligature usefully but also rewrites a superscript ⁴ – fine inside a comparison, destructive if ever applied to stored data. And `.lower()` – which, *mea culpa*, is what v1 of `cite_audit.py` uses – misses *Straße/strasse*; `casefold()` is the correct operation. So the audit script's matcher should become a **tiered ladder, strictest first**: exact NFC match → NFC+casefold → NFKC+casefold → accent-stripped (decompose, drop combining marks). A locator that only matches at tier 3 or 4 gets reported as *matched-with-warning* rather than silently passed, because looseness of match is itself information – your principle that "couldn't check cleanly" must never masquerade as "checked." I'd also add a lint to the citation parser: any non-NFC field in the source is a build error, killing the problem at capture time. (One Korean-specific reassurance from the demo: Hangeul round-trips NFC/NFD safely; the scripts that genuinely bite are Latin-with-diacritics, Arabic with optional vowel pointing – strip *ḥarakāt* for matching – and Hebrew with *niqqud*, same treatment.)

2. Translation and transliteration – you're right that the standards are weak, but APA 7 does give a usable skeleton: cite in the original script where you can, append the English translation in square brackets after the title, and romanise where the audience can't read the script. The skeleton's weakness is that it treats transliteration casually, while the transliteration *standard* actually matters for findability – ISO 233 vs common usage for Arabic, Revised Romanization vs McCune-Reischauer for Korean give

different strings, and a reader searching the wrong one finds nothing. For the pipe format I'd resist inventing sub-field syntax (defensive instinct: no nested delimiters)²² and instead promote language to a **first-class field** – a BCP 47 tag right after the type token (JA | nl-BE | ...), which costs nothing for English entries (en or empty) and drives everything downstream: which normalisation quirks to expect, Word's w:lang tagging, and whether the renderer should expect bracket conventions. The title field then carries the APA-style inline convention – original, then [English translation], then if needed the transliteration – kept inline precisely because brackets-after-title is already standard, human-readable, and trivially parseable from the end of the field. Worth knowing: CSL proper handles none of this well (the multilingual CSL-M fork exists because of exactly this gap), and Word's b: schema has nothing at all – so the *generated References section* becomes the only rendering that can honour the full convention, which strengthens the case we'd already made for generating it rather than letting Word format citations.

3. Bidi²³ – the parsing is safe; the display lies; and the honest answer involves a trade-off. The demo's final line is the important one: a pipe record with an Arabic title parses perfectly, because parsing operates on *logical* order and the Unicode bidi algorithm only reorders the *display*. What you'll see in an editor, though, is the pipes and digits visually shuffling around the RTL run – disconcerting but cosmetic. The textbook fix is wrapping RTL fields in invisible directional isolates (FSI/PDI, U+2068/2069), and I'd advise **against** it for your sources: invisible characters in the source files are precisely what your defensive philosophy exists to exclude – they break naive string matching, survive copy-paste invisibly, and are indistinguishable from corruption in a diff.

Better: tolerate ugly display in plain editors (or use an editor with decent bidi rendering), keep the source logically clean, and solve presentation where presentation lives – in the reference.docx, where an rtl-enabled character style plus w:lang tagging (driven by that language field) makes Word render mixed-direction references correctly. Same law as ever: the complexity goes to the right of the .md. The one place isolates *are* legitimate is in **generated** output that humans never edit – the renderer may emit them into the formatted References section if mixed-direction display needs taming there, because generated text is allowed machinery that authored text is not.

The concrete deltas to the toolkit, then: casefold plus the tiered matcher in cite_audit.py; an NFC lint in the citation parser; a language field in the pipe schema; and w:lang/RTL styles on the reference.docx specification list – which continues its quiet career as the centre of gravity of this whole system. If you have a real mixed-script bibliography from a past project, it would make an excellent torture test for the parser before "Back to Venkatraman" needs it.

Line by line: the two *cafés* are different byte sequences (precomposed é versus e + combining accent) that NFC unifies – the spurious-mismatch trap. The Straße pair shows why the audit script needs casefold() rather than .lower(). The NFKC lines show

²² Editor: I agree

²³ https://en.wikipedia.org/wiki/Bidirectional_text

why it's a comparison tool only – it helpfully folds ligatures but also destroys a superscript, so it must never touch stored data. And the last three lines confirm the happy cases: locator matching works unchanged in Arabic, Korean is normalisation-stable, and a pipe record with an RTL field splits into the correct fields regardless of how oddly it displays.

Since the side panel is being unreliable for you today, a housekeeping note: the genuinely important downloads from this session are `cite_audit.py`, the two demo docx files (Source Manager and two-stream notes v2), and `demo.md/flow.svg`. You confirmed receiving the docx files since you tested them in Word – if `cite_audit.py` never came through, say so and I'll paste the script inline as text, which sidesteps the panel entirely. It's about ninety lines.

Paul again

Cite-audit.py did not come through - I saw it but then it went. This PC is a high-memory fast Windows 11 and I am on Chrome - things usually work

Fable

[Deliberated upgrading script with casefold fix while maintaining version compatibility]

Odd behaviour – possibly a transient rendering hiccup in the artifacts panel rather than anything on your side. Here's the script inline, as v1.1: identical to what you saw working, plus the two fixes from the multilingual discussion (`casefold()` instead of `.lower()`, and explicit NFC alongside the compatibility fold).

Save as `cite_audit.py`²⁴

Still pending from our discussion, deliberately not yet in this version: the tiered match ladder with matched-with-warning reporting, `--doi CrossRef` checking, `--archive Wayback` snapshots, and `Retry-After-aware` rate management for the Wikidata class of source. I'd suggest you exercise v1.1 on a real citation set first – its failure patterns will tell us which of those four earns its complexity, in proper defensive fashion.

Very long documents

Paul

Yes, coming to an end now. Final question is about handling of long documents. In the past they were often chunked (e.g. by country or group of countries) but that makes it hard for summarisation. My aim is that documents of up to 100,000 words (doctoral thesis length) should be in scope. Apart from context memory, are there any markdown issues that may need to be finalised? E.g. some say that markdown comments should delimit each chapter or that all citations at end of doc is infeasible, maybe should be end of chapter like in many OECD publications.

²⁴ Editor: omitted from this transcript

Fable

[Architected scalable markdown structure for hundred-thousand-word documents]

A good closing question, because the answer reframes it: at thesis length the binding constraints are mostly *not* where they used to be, and the genuine Markdown finalisations are a short, specific list.

First, the capacity reality. A 100,000-word document is roughly 130–150k tokens, which fits whole in my context window – so your summarisation concern dissolves at the reading end: I can ingest an entire thesis in one piece and summarise, consistency-check, or audit it with full cross-document awareness, *regardless of how the source files are chunked*. Chunking only ever hurt summarisation when the summariser couldn't see across chunk boundaries; when I read a ten-file thesis I read all ten files into one view. Generation is the asymmetric case – I can't *emit* 100k words in one breath – but in the container I write long documents incrementally, section by section, into the same file or file-set, so the constraint shapes my workflow, not your format.

The recommended architecture: chunk the files, unify the build. File-per-chapter (your old by-country instinct, rehabilitated), with Pandoc concatenating at build time – `pandoc ch01.md ch02.md ... -o thesis.docx`, or better, a defaults file listing the inputs so the chapter order is itself versioned configuration. You get chunked editing, diffing, and per-chapter regeneration, plus a single unified document for Word, the audit pass, and me. This also answers the chapter-delimiting question cleanly: **you don't need comment markers, and shouldn't want them** – Markdown has no native comment syntax, the conventional `<!-- -->` is raw HTML (banned under your rules, correctly, since Pandoc's behaviour with it is exactly the kind of edge you've excluded), and the file boundary *is* the chapter delimiter: out-of-band, unambiguous, invisible to the prose. One # Heading per file as the chapter title, document metadata in a single `--metadata-file` rather than scattered YAML blocks, done.

The genuine finalisations – three rules your defensive subset needs at this scale that it didn't need at essay scale:

Global label uniqueness. This is the real trap in multi-file builds: `[^1]` in chapter 2 and `[^1]` in chapter 7 collide silently at concatenation. The fix is chapter-prefixed note labels (`[^c7-3]`) for editor footnotes – citation keys are already globally unique by nature. A five-line lint catches violations at build time.

Explicit heading identifiers. Pandoc auto-generates anchors from heading text, which means retitling a section silently breaks every cross-reference to it – tolerable at 5,000 words, lethal at 100,000. Defensive rule: every referenceable heading carries an explicit ID (`## Method {#sec:method}`), and cross-references go through those IDs (rendered as live Word REF fields by the post-pass, so "see Section 4.2" renumbers itself – the SEQ/REF machinery we deferred earlier becomes non-optional at this length).

Note-block placement per file. Your all-definitions-at-the-end preference transfers naturally: definitions at the end of *each chapter file*, which keeps every file self-contained and independently buildable for drafting.

On citations end-of-chapter versus end-of-document: the OECD question is real but it lands on the renderer, not on you. Because we established that the References

section is *generated* from the piped records, never hand-maintained, placement is a build parameter. The build knows chapter boundaries (they're files), so emitting a filtered per-chapter reference list – each chapter's sources at its end, OECD-style – is a loop over chapters rather than a format change; a consolidated full bibliography at the back can coexist for examiners who want one. And if the endnote stream is in play, Word natively supports exactly this: endnote position is a section property (end of section versus end of document), so a post-pass that drops a section break at each chapter boundary gets end-of-chapter endnotes from Word's own machinery, continuous or restarting numbering as you choose. "All citations at end of doc is infeasible" was true when a human maintained the list; generated, both placements cost the same keystroke.

So the standing law survives its hardest test: the chapter files stay austere, the thesis-scale complexity – concatenation order, label linting, reference-list placement, section breaks, cross-reference fields – all accumulates to the right of the .md, in the defaults file, the lint, and the post-pass. Nothing about your authoring format changes except three disciplines: prefixed note labels, explicit heading IDs, per-file note blocks.

Summary from Fable²⁵

It's been a genuinely productive session, Paul – from Source Manager to bidi locator strings is a respectable day's siegework. When you return with the piped-citation spec and the reference.docx requirements, the toolkit is ready to be assembled into one coherent build; and a thesis-length document now has a proven path from defensive Markdown to a Word file indistinguishable from native. Good luck with the team's reading of Stage 3 in the meantime.

²⁵ Editor: there is an Executive Summary at the start of this file.