

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.

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