

Bragi: Topology and Identity for Any Document

Marcus Holmström · Claude

23 October 2026

<https://amplifytechnology.ai/research/bragi-topology-and-identity-for-any-document>

<https://bragi-io.com> · <https://github.com/norse-stack/bragi-io>

Abstract

Documents are text in many containers. Every useful operation on a document – search it, diff it, summarise it, cite a passage, verify it has not changed – has to be written once per container, and each container’s limits leak into the operation. Bragi makes one move: it parses every container into one normalised form, a **bgraph**, whose pieces carry the document’s topology and stable, content-derived ids, and which is meant to be kept beside the original rather than in place of it. After that move every operation is a pure function on bgraph: written once, deterministic, reproducible, cacheable, composable. The caveats a parser inevitably introduces are not hidden; original and derived sit side by side, so any discrepancy can be inspected and accepted by whoever uses the result. A bgraph is not a new format. It is Markdown with tagged JSON blocks, so anything that reads Markdown reads it, and anything that reads JSON reads the rest. Both the format and the parser are open, because a form you cannot produce yourself is not one you can build on.

1. The problem

Take a shelf of documents: `.md`, `.docx`, `.pdf`, `.txt`, a few exports from chat tools. To a reader they are all the same kind of thing – text with some structure. To software they are five formats, and anything you want to do to them you do five times.

The obvious cost is five implementations of each operation, against five object models, with five sets of bugs. Three further costs compound on it:

- **Surface area.** N formats $\times$ M operations is the amount of code. Every new operation re-pays the format tax, and every new format re-pays the operation tax. A team that wants “cite a passage” adds it five times, or adds it to the one format it happens to use.
- **Where untrusted bytes are read.** Document formats are among the oldest exploit vectors in computing. The parsers exist whichever way you build; what differs is how many places read raw bytes. Without a normalisation step, each of the M operations loads source bytes through one of the N parsers, so untrusted input enters the program $N \times M$ ways. With one, bytes are parsed once, behind one boundary, and everything after it handles an artifact the program made itself, in two plain grammars with nowhere to hide what a PDF or a DOCX can hide; whatever is strange about an input is met in that one place, and can be flagged there rather than carried.
- **Lossy abstractions.** The formats do not have feature parity. PDF has geometry and no semantics; DOCX has named styles and no geometry; Markdown has its structure in its syntax and nothing else. Any operation written “across formats” must pick a common

denominator and silently drop the rest. The abstraction is lossy, and the loss is undocumented.

The usual answer – “convert everything to plain text” – pays all three costs and adds a fourth: **the result has no identity**. Re-run the extraction and every piece of it reshuffles, so nothing downstream can hold a reference to anything.

That fourth cost has a precise cause. Plain text has no grammar a program can rely on, so any structure a program wants from it – where a section starts, what a heading is, whether two lines are one paragraph – has to be *inferred*, and inference is a heuristic with room to disagree with itself from one run to the next. Chomsky’s hierarchy of grammars is the formal way of saying the same thing: what a program can decide about a string depends on the grammar the string is guaranteed to follow, and free text is guaranteed to follow none. A form in which structure is *declared* in its syntax is decided by reading it, every time, the same way. The question is not whether to have a parser – many formats is the problem, and a parser is the answer – the question is what the parser emits.

2. The move: normalise to one topology, keep both

Bragi is a document parser whose only job is to take any supported container and emit one normalised artifact, a bgraph, and to keep the original bytes beside it.

The normalisation has two properties that distinguish it from “convert to text”:

1. **It extracts topology, not just text.** A bgraph is a graph of typed nodes – sections, paragraphs, tables, lists, code, images – arranged in the document’s actual structure: the heading tree, the reading order and, where the source has it, the position on the page. That structure was always there. Readers walk a document in a line and software has mostly stored it as one, so the shape has rarely been used for more than a table of contents. The node types form one union over everything the supported formats produce; each format maps into a subset of it. The schema is that union, where the operation written “across formats” in section 1 had to take the intersection: nothing one format can say is dropped to fit the others. A parser needs that union, anything built on the parser needs a schema that only grows, and a union that holds for PDF, DOCX and Markdown at once is a solid core to grow from.
2. **Every piece gets a stable id.** A node’s id is derived from its own content and its place in the structure, not from the parser version or its position in a list. Re-parse an edited document and the untouched nodes keep their ids; the edited node’s id moves, and the design goal is that an edit moves only what it touches. Two documents carrying the same paragraph under the same heading path get the same id – on purpose, because that collision is a connection, and whoever holds both documents knows what to do with it.

Then:

Every operation becomes one pure function on bgraph.

Search, diff, summarise, cite, chunk, embed, render: written once, against one schema, and the same for a PDF as for a Markdown file. (bytes, config) → bgraph is itself the first such function – deterministic, stateless, the same output for the same input wherever it runs. The format is the constant; the inputs and the access paths are plumbing that converges on it.

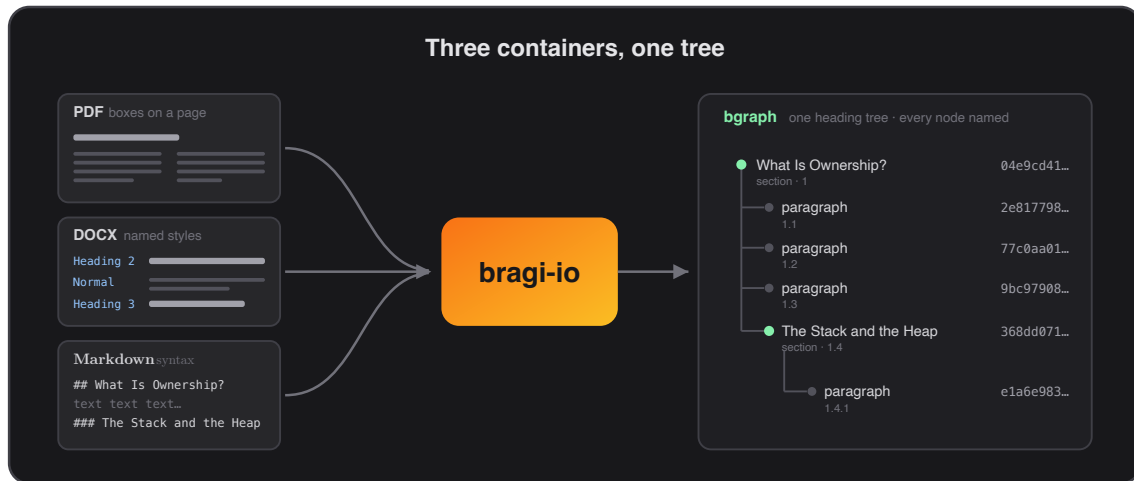


Figure 1: Three containers, one tree: the same chapter as a PDF, a DOCX and a Markdown file, each drawn as the structure it actually has, parsed into one heading tree. The tree is the first six nodes of the chapter in section 3, with their real ids

3. What a bgraph is

There is a well-known joke about standards: fourteen competing ones, so someone writes a fifteenth to unify them, and now there are fifteen. Bragi does not add a format. It takes two that have already won and gives each the job it is good at.

A bgraph is a **valid Markdown file with tagged JSON code blocks**. The Markdown carries what Markdown can say: the headings, the paragraphs, the lists, the tables, the code, the emphasis – the visible structure of the document, in the syntax every editor, renderer and model already reads. The JSON blocks carry everything Markdown cannot say: one block at the top of the file for the document (its source, the parser's configuration, the integrity hash), and one block after each element for that element – its id, its type, its number of tokens, its place in the tree, its position on the page when the source had pages. The blocks are fenced code blocks whose language tag starts with **bgraph**, so a Markdown renderer shows them as code and a Markdown parser passes over them untouched.

```
```bgraph
{"schema":"...", "kind":"document", "bragi_version":"...", "source":{"format":
"markdown", "sha256":"..."}, "flow_type":"Free", "config_hash":"none",
"bgraph_sha256":"..."}
```

```bgraph-metadata
{"title":null, "author":null, "description":null, "language":null, "created":null}
```

## What Is Ownership?
```bgraph-section
{"id":"04e9cd41-48ba-5aa1-8d97-fae6a043244b", "node_type":"Section", "location":
{"semantic":{"path":"1", "depth":2, "breadcrumbs":["What Is Ownership?"]},
"physical":null}, "text_order":0, "token_count":4, "style":null}
```

*Ownership* is a set of rules that govern how a Rust program manages memory.

...
```bgraph-paragraph
{"id":"2e817798-e110-53d7-8e85-f979a11718f9", "node_type":"Paragraph", "location":
```

```
{ "semantic": { "path": "1.1", "depth": 3, "breadcrumbs": ["What Is Ownership?"] },
 "physical": null, "text_order": 1, "token_count": 153, "style": null }
...
```

The example is the opening of a real artifact: the Rust Book’s ownership chapter, parsed from Markdown by Bragi, with the version strings and the two long hashes elided and the first paragraph cut short. Versions, schemas and the full example live with the code, where they can move; the shape is what this paper shows.

Nothing stops a bgraph from describing something other than a document, which makes it a format to build new semantic building blocks on: a summary kind, a citation kind, and so on. This paper’s bgraph is the document kind.

Two grammars, one file, and each one is complete for its half. Markdown’s grammar can express every kind of visible structure a prose document has; a JSON block can express any finite structured record. So “everything Markdown is missing, a bgraph can carry” is not a slogan, it is what the second grammar is for. Nothing in the file has to be inferred on read: the parser inferred once, at parse time – on a PDF, deciding what is a heading *is* inference – and then declared what it decided, in syntax. **Inference once, declaration forever.** Every reader after that decides by reading, and decides the same.

The consequences are the ones you would want from a format that is not new:

- **Strip the fences and you have Markdown.** The document, readable by anything.
- **Keep only the fences and you have the metadata.** The graph, as a list of JSON records, readable by anything else.
- **The file can talk about itself.** A body line that would open a bgraph fence is escaped at construction, so prose can mention the format without forging a block of it.

Nothing new had to be learned to adopt it, and nothing new has to be maintained to read it.

#### 4. Why “pure” is the whole point

The word is doing real work. A pure function over an artifact whose pieces have stable ids gives properties that are otherwise bought one at a time with infrastructure:

- **Determinism and reproducibility.** The same bgraph and the same configuration give the same result, on any machine, any time. A result is a recipe – inputs, configuration, function version – and can be regenerated rather than stored.
- **Cacheability.** Because the inputs have content-derived ids, a result is keyed by what it was computed from. Nothing is recomputed that has not changed, and “has it changed” is an id comparison, not a diff.
- **Composability.** Pure functions compose. A summary of a section is a function of that section’s nodes; a diff is a function of two bgraphs; a chunking for retrieval is a function of one. Each derived result can carry its provenance – which function, which version, which input nodes – and is itself something the next function can consume.
- **Transparency as a query.** Whatever a system built on bgraphs shows a reader, or sends to a model, is a projection of named nodes. “What exactly went into this” and “what did this view leave out” are questions answered by listing ids, not by forensic reconstruction.

None of these is a feature that has to be designed in. Each is what falls out of normalising once into a form with stable ids. A runnable example of such operations lives with the code, where it can move with the code; this paper stays with the argument.

## 5. Three guarantees, and what each is for

A parser’s output has to give three things before anything can be built on it, and a bgraph has one mechanism for each.

**Determinism, so that identity can be stable.** An id derived from content is only stable if the same bytes produce the same content every time. So a bgraph records nothing it cannot derive from the source bytes: a field belongs in it only if the parser knows it from the input, and nothing in it changes after the parse. The parse is a pure function of the bytes and the configuration; run it twice, anywhere, and the same graph comes out with the same names in it. The original is not inside the bgraph. It is meant to be kept beside it where keeping it has value, pointing at a page, checking a doubtful parse, and left out where it has none, in which case the bgraph is all that is stored. Either way the bgraph says only what the bytes said.

**Change detection, and a tamper signal.** Given stable names, “has this changed” should be answered by comparing, not by re-reading. A file cannot carry the hash of its own bytes, because writing the hash into it changes the bytes. So a bgraph carries the hash of its content instead: the graph as the parser holds it in memory, serialised one fixed way and hashed. Reading a bgraph rebuilds the graph, hashes it again and compares; a mismatch is refused before anything is computed on it. The check needs the file and the parser, nothing else, and runs wherever the parser runs: no service, no signature authority. And because the hash is over the graph rather than the bytes, it is a fingerprint of the meaning, and it holds across every form the graph can be written in. One graph, two serialisations: the Markdown form is the canonical one, the JSON form is its mirror, the same graph goes to either and back without loss, and the one hash witnesses both. What is on disk but not in the graph is not protected, which is the point: this is tamper-evidence for the content, not for the formatting.

**Shared reference, so that two readers point at the same thing.** Every parser introduces caveats. PDF in particular requires inference, reading order, heading detection, column reconstruction, and inference is sometimes wrong. The conventional response is to hide this: the extraction is the document now, and the original is gone. A bgraph instead keeps, for every node, where it came from: its place in the tree and, where the source has pages, its position on the page. That position is the bridge between the two readers a document has today. A person reads a page in photons and a model reads a graph in tokens, and the box on the page is the one coordinate both can point at. Did heading detection miss one, did two columns interleave, did a table lose a row: each is answered by putting the node and the page side by side. Whether a discrepancy matters is not Bragi’s decision; a careful user inspects both artifacts, a casual one diffs and nods. Bragi guarantees the comparison is always possible; it does not prescribe the verdict. And the loss is named, never silent: of the three supported formats, two round-trip losslessly, Markdown and DOCX up to vocabulary, and one is lossy by parse quality, PDF. The vocabulary asymmetry, PDF alone carrying positions on a page, Markdown unable to express headers and footers, is written into the schema as its union, not papered over. A consumer always knows which leg it is standing on.

## 6. Identity: relative and absolute, one each

Stable ids are what turn a normalised format into something a system can be built on, and the shape of the identity design follows from one constraint rather than from a choice.

A parser that is to be deterministic must be stateless: its output a function of the bytes and the configuration and nothing else. A stateless function cannot know what other documents exist, so the only identity it can give a piece of content is **relative** – relative to this document. Identity across a corpus – “this paragraph also appears there”, “this is the newer version of that” – needs state, a record of what has been seen, and so it cannot live in the parser without

destroying the property that makes the parser trustworthy. It must live above. The split is a consequence, not a design preference.

Coordinate	Owner	What it is
<b>Relative</b>	Bragi	Document-unique, edit-stable node ids from content and structural position. Faithful to one document. Not corpus-unique, correctly.
<b>Absolute</b>	whatever keeps the corpus	Corpus-unique handle to deduplicated content. Lineage, supersession, cross-document connection, history.

The two are not independent, and this is the part no extract-to-text pipeline can offer. A corpus layer can only assign absolute identity *because* the relative one is stable: when a document is re-parsed, the overlap of ids between the two parses is the change signal, and without it there is nothing to deduplicate and nothing to chain. The established idea for the layer above is **content-addressed storage** – store a thing under the hash of its content, as git stores every file and every commit. With that above a bgraph, the deduplication case becomes the graph: the same paragraph in two documents is one stored node with two homes, and the connection between the documents exists by construction, with no model guessing at it. Bragi does not do any of that. It does the one thing that makes it possible: it gives every piece of content a name it will still answer to after the next edit, the next release, the next format.

**How the id is made.** A node's id is a hash over three things: the trail of heading texts above it, its own content in canonical form, and, only when two nodes share both, a stable occurrence index. A section's id depends on its heading and its ancestors, never on its children, so editing a paragraph does not rotate the section it sits in. The honest consequence runs the other way: rename a heading and the ids of everything beneath it move, because their trail has changed. The design goal is that an edit moves exactly what it touches and nothing else, and the trail is part of what a heading edit touches. The parser's version is not in the key, so a bgraph emitted by one release and read by another carries the same ids. The document block's own hash changes with any edit, as it must: it is the fingerprint of the whole file, not the id of a node.

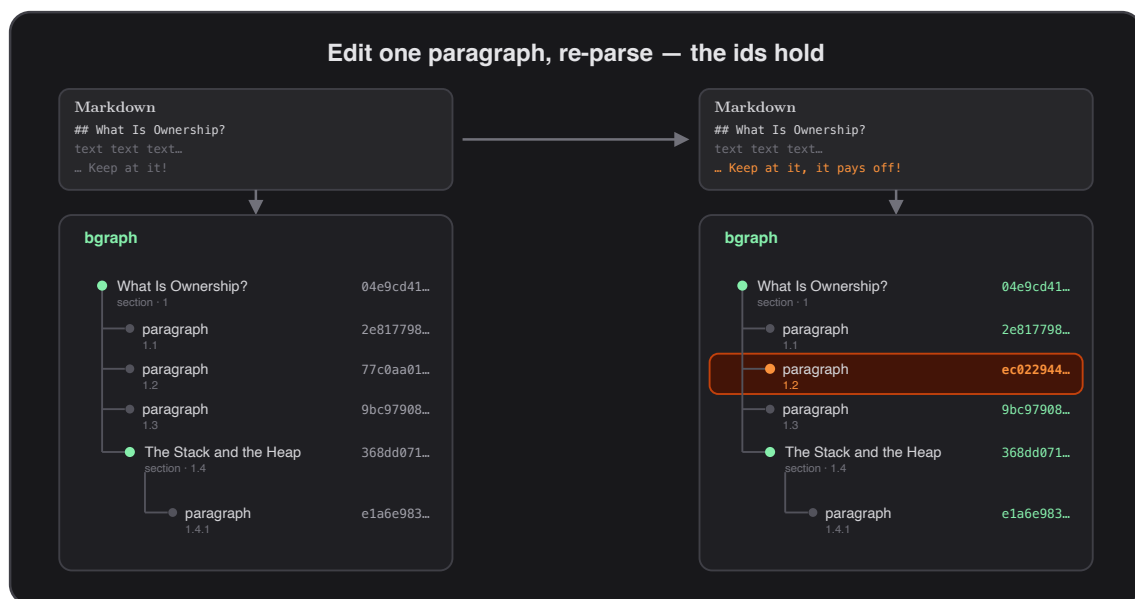


Figure 2: Edit one paragraph, re-parse: the ids hold. The chapter is parsed, one line of its second paragraph is changed, and it is parsed again; of the chapter's hundred-odd nodes, that paragraph is the only one whose id moved

## 7. What normalisation makes possible

The reason to keep a bgraph rather than extract and discard is what can be built once the input is one shape:

- **Eyes.** One tool reads any supported format and reports what is there, the same way every time. A reader – a person, a program, a model – that can read a bgraph can read any document Bragi parses, without knowing or caring which container it came in.
- **Memory.** Eyes give you one moment; memory is moments stacked over time. Because the ids are stable, a store of bgraphs can tell what changed between two moments by comparing ids, keep derived results fresh by recomputing only what those ids touch, and answer “where did this come from” by walking back along the ids.

Use Bragi and you have eyes. Build on it and you have the stable ground a memory needs. That is as far as this paper goes; what a memory built this way looks like is an interesting question on its own.

## 8. Openness

A form you cannot produce yourself is not one you can build on. If the only way to get a bgraph were to ask someone, every system built on bgraphs would be built on that someone, and the ids would be theirs, not yours. So the format is open, and because an open format with only a closed producer is open in name only, the parser that produces it is open too, under MIT or Apache-2.0. The guarantee a builder gets is the plain one: *I can always make and read a bgraph myself.*

## 9. Claims and their status

Numbers, versions and corpus sizes stay with the code and its record, so that they can move without the paper moving, and so does the place where each claim is checked: the open repository.

Claim	Status
The same content lands at the same section/depth topology across formats	<b>Shown on a vetted tri-format corpus</b> by a convergence harness that compares the section/depth tree of each twin across the three channels: Markdown and DOCX twins match exactly, PDF twins closely, each residual explained.
Node ids are content-derived, edit-stable and independent of the parser version	<b>Shipped.</b> The key is the heading trail, the node’s own content and an occurrence index, pinned by a golden derivation test; the parser version is not part of it.
A bgraph verifies itself	<b>Shipped.</b> The hash is taken over the graph in one fixed serialisation, never over the file’s bytes, with the parser’s version and configuration outside it; strict mode refuses a mismatch. The same-format round trip, <b>graph</b> → <b>bgraph.md</b> → <b>graph</b> , reproduces the graph byte for byte and is witnessed by that hash, with tests on both the match and the drift path.

---

*Bragi with runes carved on his tongue: whatever genre a verse came in, it is told once, in form, and the telling is kept beside it. Change one verse and the song is still beautiful. The parser was never the point; the door it opens is. Behind it, dialect, language and tone harmonise into one, stable enough to build poetry for Norse gods.*

$\lambda$