ChatGPT

Document Ingestion & TipTap Mapping

To support rich “block” content in WorkSphere, uploaded files (PDF, DOCX, etc.) must be parsed into structured JSON that matches TipTap’s node schema (paragraphs, headings, lists, tables, etc.). A typical pipeline is: (1) use a JS library to extract content (with some structure) from the file, (2) convert that content into TipTap JSON (or HTML) using the TipTap API.

- **PDF Parsing:** For plain-text PDFs, **pdf-parse** is lightweight and easy: it returns the full text, page count, and metadata without native binaries [1]. Example usage:

```
import pdf from 'pdf-parse';
const buffer = await file.arrayBuffer();
const { text } = await pdf(buffer);
const paragraphs = text.split(/\r?\n\r?\n/);  // split on blank lines
const tiptapContent = paragraphs.map(p => ({
  type: 'paragraph', content: [{ type: 'text', text: p.trim() }]
}));
editor.chain().focus().setContent({ type: 'doc', content:
tiptapContent }).run();
```

This yields simple paragraph nodes. **Limitations:** pdf-parse handles text-based PDFs well [1], but struggles with scanned/OCR content, multi-column layouts, or complex tables [2]. For those, consider **pdf.js (pdfjs-dist)** or **pdf2json**. The Mozilla PDF.js engine (pdfjs-dist) can extract text with precise layout (coordinates, fonts) [3], though it adds ~2 MB gzipped to your bundle and is more verbose to use [4]. For JSON output, **pdf2json** converts a PDF into structured JSON (with coordinates and styling) [5], useful if you need to reconstruct tables or detect headings by font size. As a rule: use *pdf-parse* when you just need raw text quickly [1]; use *pdfjs-dist* or *pdf2json* when layout/position matters [5] [4]. You may also combine with OCR (e.g. Tesseract.js) for scanned PDFs.

- **DOCX Parsing:** Libraries can convert Word files to HTML or text. A common approach is **Mammoth.js**, which maps DOCX to clean HTML using semantic styles (e.g. “Heading 1” → `<h1>` [6]). Mammoth supports headings, lists, tables, images, footnotes, etc. [7]. For example:

```
import mammoth from 'mammoth';
const { value: html } = await mammoth.convertToHtml({ arrayBuffer:
docxBuffer });
editor.chain().focus().setContent(html).run();  // TipTap can ingest
HTML as content
```

This preserves most structure (headings, paragraphs, lists, tables) and lets TipTap parse it into its internal JSON. Other options include **docxjs** or **docx2html** (lalalic/docx2html), which similarly convert DOCX to HTML in-browser [8]. For simple plaintext extraction, **officeParser** (npm) can parse DOCX (and other Office formats) to raw text [9], but it does *not* preserve structure; use

that only if you need plain text. In summary, Mammoth (or equivalent) is best for structured output – it understands DOCX styles [6] [10], so you get headings, lists, tables, etc. If you use Mammoth, you can feed its HTML into TipTap (via `editor.setContent(html)`) or manually create TipTap JSON nodes (e.g. mapping `<h1>` to `type: 'heading', attrs: {level:1}`).

- **Other Formats:** For Markdown, JSON, or HTML imports, TipTap provides built-in parsing (or official import extensions). For ODT/OpenOffice, you might convert to DOCX server-side or use a Java library (outside JS). For images/PDF pages, TipTap's Image node or a PDF-to-image library could be used, but those are separate concerns.

Once parsed, build a **TipTap JSON document**. TipTap's data model is a strict schema of nodes (blocks) and marks [11]. For example, to create a heading and paragraph you'd do:

```
const doc = {
  type: 'doc',
  content: [
    { type: 'heading', attrs: { level: 1 }, content: [{ type: 'text', text:
'Title' }] },
    { type: 'paragraph', content: [{ type: 'text', text: 'Some text
here.' }] }
  ]
};
editor.chain().focus().setContent(doc).run();
```

As [43] explains, "Nodes are like blocks of content (paragraphs, headings, code blocks, etc.)" [11], so your conversion code should emit the appropriate node types. TipTap StarterKit supports paragraphs, headings, bullet/numbered lists, tables, etc., matching what most documents contain. Custom nodes (e.g. "callout" or "embed") can also be inserted as needed (as WorkSphere already does).

Building Agents on the Block Graph

With content in TipTap (and stored in Convex), we can build AI agents that operate over this block graph. Conceptually, each TipTap node is a "block" of text (a paragraph, heading, etc.), and the blocks form a directed (or loose) graph of context (e.g. sections, references). An agent can retrieve relevant blocks to answer user queries or augment content. Two architectures are common:

1. Server-side LLM (OpenAI API)

In this variant, we use a powerful remote LLM (e.g. OpenAI's GPT-4) for inference, and perform retrieval and prompt construction on the server. A typical pipeline is: **chunk** the document/blocks, **embed** them, store in a vector index; **search** for relevant chunks based on the user's query; then **build a prompt** that includes those chunks (and possibly conversation history) and call the LLM. Best practices include:

- **Chunking:** Split documents into manageable passages (e.g. by paragraphs or sections). Structure-aware splitting (keeping headings with their section) often works better than fixed-size chunks [12]. One can recursively split on headings or blank lines, or use NLP sentence tokenizers. Include some overlap (10–20%) between chunks to preserve context across boundaries [13].
- **Embedding:** Use a text-embedding model (OpenAI's `text-embedding-3-small` is 1536-dimensional by default [14]) to vectorize each chunk. Store these in Convex's RAG component or

an external vector DB. In Convex, you can use `new RAG(components.rag, { textEmbeddingModel: openai.embedding(...), embeddingDimension: 1536 })` [14] and call `rag.add(ctx,{ namespace, text: chunk })` to index each block.

- **Retrieval:** When the user asks a question, use the RAG component's search (or Convex's RAG `generateText`) to find the top-N relevant chunks by semantic similarity. Convex's RAG can even return the full text of matching chunks and nearby context automatically [15].
- **Context & Prompt:** Combine the retrieved chunks (with metadata like headings or page numbers, if useful) into a prompt. For example, prepend "Context:" with each relevant paragraph, then the user's question. Many RAG setups format context as markdown or bullet lists. TipTap's block graph can help: e.g. when a query is triggered on a given block, you might include sibling/parent blocks from the document in the prompt.
- **Agent Use:** Within Convex, you can use the Agent API. Define an Agent (e.g. `new Agent(components.agent,{ name, chat: openai.chat(model), instructions, tools })` [16]) for your use case (e.g. "Workspace Assistant"). You can then create or continue a thread and call `thread.generateText({ prompt })`, which will automatically include past thread messages. To incorporate retrieved context, Convex's RAG component offers `rag.generateText(ctx,{ search:{ namespace, limit }, prompt })`, which "automatically search[es] for relevant entries and use[s] them as context for the LLM" [15]. For example:

```
// Convex action example (server-side)
const { text: answer, context } = await rag.generateText(ctx, {
  search: { namespace: userId, limit: 5 },
  prompt: userQuery,
  model: openai.chat("gpt-4")
});
return { answer, context };
```

This simplifies retrieval-augmented prompting; the answer includes the LLM's response plus the searched context.

Best Practices: Use the Convex RAG component (vector search) to manage embeddings and similarity search [14] [15]. Chunk by document structure (headings, paragraphs) rather than arbitrarily, and give the LLM enough context tokens (limit ~3000 tokens minus prompt overhead). Weight chunks by importance or freshness if needed. Keep the graph updated: when TipTap blocks change, update the corresponding RAG entries. You can trigger these updates via Convex "actions" on upload or edit.

2. Client-side Small LLM (WebGPU)

Here we run lightweight LLMs entirely in the user's browser using WebGPU or WebAssembly, avoiding any server calls. This grants privacy (no data leaves the client) and personalization (the model can be fine-tuned or prompted on the user's local context). Approaches include using WebLLM, Transformers.js, MLC.js, etc.

- **Models & Frameworks:**
- **WebLLM** (MLC's engine) lets you load quantized LLMs (e.g. small Llama models) in-browser with GPU acceleration [17]. It provides an OpenAI-like API and uses WebGPU+WASM to run even large models locally.

- **Transformers.js** (HuggingFace) can run dozens of standard models (text, vision, audio) in-browser using ONNX Runtime [18]. It supports WebGPU (set `device: 'webgpu'`) for acceleration [19]. For example:

```
import { pipeline } from '@huggingface/transformers';
const model = await pipeline('text-generation', 'Xenova/gpt2', {
device: 'webgpu' });
const result = await model("Hello, world!");
```

- **MLC.js / llama.js** are other options to load open models via WASM/WebGPU (e.g. xwanonz/transformers.js or graphrag).

- **Local Retrieval / Context:** Since we can't easily run a vector DB locally, one approach is to load the workspace's blocks (text) into memory (or use a small in-browser index) and do a linear or approximate search (e.g. keyword matching, or small-scale embedding with on-device model). Another idea is to treat the entire context graph as part of the model's prompt through "instruction tuning" – for example, fine-tune a tiny model on user-specific docs (very experimental), or simply prepend the user's content into each prompt. In practice, you might keep a local copy of relevant chunks (as JSON) and include them in the prompt up to the model's context limit.

- **Orchestration:** The client agent could be a web worker or service worker running the model inference. For instance, you could load a quantized LLaMA-3B model once (cache in IndexedDB) and then spawn a WebGPU worker to answer queries. Because models like WebLLM support streaming, you can stream tokens back to the UI as the user types. For domain-specific tasks, you might load specialized small models (e.g. a ChatAssistant model fine-tuned on technical docs) or use a prompt that injects system instructions.

- **Limitations:** Running LLMs in-browser is getting feasible, but performance/capacity is limited. Even with WebGPU, a 7B model may run slowly (~several seconds per query) on a mid-range GPU, whereas a 1–2B model is more interactive. You'll also need to manage model downloads (hundreds of MB) and may rely on quantized weights to reduce size. However, this approach maximizes privacy and offline capability: *"WebLLM is fast, private (100% client-side), and convenient (zero setup)"* [17]. You sacrifice some accuracy and breadth of knowledge, but gain that all computations and context remain local.

Architectural Trade-offs & Comparison

Criterion	Large LLM (OpenAI API)	Small Local LLM (WebGPU)
Performance	Typically high (sophisticated models on powerful GPUs). Low latency if server is fast. Network delay.	Moderate; depends on client hardware. WebGPU can accelerate, but model size <~3B tokens.
Context Size	Very large (GPT-4 or similar can handle ~8K–32K tokens, plus history).	Limited by browser memory and model, often <2000 tokens total.
Privacy	Low – data and user context sent to external API (unless enterprise+security measures).	High – everything stays on client.

Criterion	Large LLM (OpenAI API)	Small Local LLM (WebGPU)
Personalization	Static model unless fine-tuned server-side. Can inject context, but core model same for all.	Very high – can "train" or specialize models per user, and use their documents as core context.
Cost	Ongoing (API calls per query, potentially expensive at scale).	One-time (model download costs bandwidth/prefetch; compute is user's device, no API fees).
Implementation	Server/convex-based. Easier logic (just call API). May use Convex Agent/RAG components for context.	Client-heavy. Requires bundling model inference libs (WebLLM, Transformers.js, etc.).
Integration	Easy in Convex: use `@convex-dev/agents` with OpenAI. Use RAG component for vector search [14] [15].	More custom: likely initiate inference from client code, maybe via Convex HTTP if needed.
Responsiveness	Depends on API/network. Possible queue times.	Immediate (no network), but computation may be slower for large models.
Up-to-dateness	Always latest model version from provider (e.g. GPT-4, function calls, etc.).	Limited to versions/models you bundle; less frequent updates.

In practice, the **OpenAI API approach** is often simpler and more powerful: you rely on a state-of-the-art model and Convex's built-in agent/RAG tools [14] [15]. However, it requires network access and exposes data externally. The **client-side LLM approach** maximizes privacy and can feel instantaneous, but is more complex to implement and currently best with smaller, specialized models. A hybrid is possible: use OpenAI for heavy reasoning tasks and a local model for lighter, private queries.

Integration with WorkSphere (React+TypeScript+Convex)

In the existing stack, you'd tie this into Convex's reactive logic and your TipTap editor:

- **Store TipTap content in Convex:** Use the ProseMirror Sync component to sync TipTap between clients (if needed). WorkSphere already has TipTap blocks (see `TipTapEditor.tsx`), so you could configure `@convex-dev/prosemirror-sync` to persist the document and allow real-time updates. Alternatively, on upload or edit, simply call a Convex action to save the JSON representation of each block (e.g. one row per block with its type/attrs/text).

- **On file upload:** In your React code, when a user uploads a PDF or DOCX, run the parsing logic (pdf-parse, mammoth, etc.) on the client or in a Convex action. Then insert the resulting blocks into TipTap by `editor.setContent(...)` or by using TipTap commands (e.g. `editor.chain().focus().insertContent(...)`). You can also send each block to Convex (e.g. `addBlock(block)`) so that the backend can index it. If blocks are long, you might chunk them first (e.g. split a paragraph if >500 tokens).

- **Updating Convex (Reactive):** Whenever TipTap content changes, update Convex so that the RAG index stays current. For example, on every `onChange(value)` call, diff new blocks vs old and call Convex actions like `rag.add(ctx, { namespace:userId, text: blockText, …})` or `rag.remove(...)`.

Use Convex Actions (server functions) to call RAG APIs. Convex will store embeddings & vectors per block.

- **Query interface:** Provide a UI (e.g. a chat box or a button) that calls a Convex action to query the agent. On the server action, use the Convex Agent or RAG component: e.g.

```
// convex/actions.ts
import { rag } from "./ragSetup";  // RAG instance like in [50]
export const askWorkspace = action({
  args: { question: v.string() },
  handler: async (ctx, { question }) => {
    const { text: answer } = await rag.generateText(ctx, {
      namespace: ctx.auth?.userId || "global",
      prompt: question,
      model: openai.chat("gpt-4")
    });
    return answer;
  }
});
```

In React, call this action via `api.askWorkspace({ question })` and display the response.

- **Use Agents and Tools:** If you have user-specific tools or plugins (e.g. "summarize selected block", "generate tasks from this page"), you can leverage Convex Agents with `tools` [16]. For example, one tool could fetch more details about a selected block, another could open a ticket. The Agent component will manage context history and streaming responses.

- **Reactive Context Graph:** Since Convex is real-time, any change to blocks automatically triggers updated data flows. For example, you might subscribe to `api.blocks.query()` to get all blocks in the workspace, and whenever they change, update a local in-memory index or simply rely on Convex's RAG queries to always retrieve fresh context. The "block graph" (which blocks refer to or link to each other) can be implemented by storing block IDs and parent-child relationships in Convex. Then your agent logic could traverse this graph to assemble prompts (e.g. include parent heading text when a child paragraph is queried).

Recommendations & Implementation Path

- **File conversion:** Use *pdf-parse* and *mammoth.js* for most cases, as they are pure JS and lightweight [1] [6]. For PDFs with complex layouts, consider a server-side microservice or *pdf.js* with Web Workers to avoid bloating the client bundle [4]. Map parsed sections to TipTap nodes (paragraphs, headings, lists, etc.) exactly as in your editor schema [11]. TipTap allows setting content from HTML or JSON, so either approach is valid. Test on sample PDFs/DOCs to refine your splitting (e.g. collapse multiple newlines into separate paragraphs, detect lists by bullet characters, etc.).

- **Tools/Libraries:**

- *PDF:* `pdf-parse` (simple text) [1], `pdfjs-dist` / `pdf2json` (layout) [5] [4], plus optional OCR (Tesseract.js).

- *DOCX:* `mammoth` [6], `docx2html`, `docxjs` [8].
- *TipTap:* Core extensions (StarterKit) cover most needed nodes. Use `editor.setContent()` or TipTap commands to insert parsed content. For images in docs/PDF, use TipTap's Image node with `upload` if needed.
- *Convex:* `@convex-dev/prosemirror-sync` for real-time editing sync; `@convex-dev/rag` for embeddings and semantic search [14]; `@convex-dev/agents` for chat workflows [16].
- *Local AI:* `mlc-ai/web-llm`, `@huggingface/transformers` (Transformers.js), or `@xwanonz/transformers.js` for browser LLMs. Use quantized models (4-bit) for speed [20].

- **Workflow:** On upload, parse and insert content into TipTap. In parallel, run a Convex action to add content to RAG (chunking as needed) [14]. In the UI, add a query interface (e.g. command palette or chat panel). When invoked, call a Convex action that uses `rag.search` or `rag.generateText` to produce an answer [15], then display it in context. For SLM mode, the UI could instead call a local inference function (e.g. via Web Worker) with the content pulled from the editor.

- **Final Advice:** A hybrid approach often works best. Use OpenAI/Convex Agents for heavy-duty querying or summarization, and consider enabling a local LLM for on-device privacy-sensitive tasks. Ensure the architecture is modular: separate out "ingest & store blocks", "compute embeddings", and "agent query" functions. This way you can iterate on chunk size or model choice without changing the entire pipeline. By leveraging Convex's components (sync, RAG, agents) and TipTap's schema, you can build a reactive, block-based editor where AI agents can read and write content naturally.

Sources: See TipTap's schema docs [11], Convex's RAG & Agent guides [14] [15] [16], and libraries such as `pdf-parse` [1] and Mammoth [6] [10] for details on content extraction. These show how to parse files into structured text and integrate with JS-based AI pipelines.

---

[1] [2] [3] [4] [5] 7 PDF Parsing Libraries for Extracting Data in Node.js
https://strapi.io/blog/7-best-javascript-pdf-parsing-libraries-nodejs-2025

[6] [7] GitHub - mwilliamson/mammoth.js: Convert Word documents (.docx files) to HTML
https://github.com/mwilliamson/mammoth.js

[8] [10] How do I render a Word document (.doc, .docx) in the browser using JavaScript? - Stack Overflow
https://stackoverflow.com/questions/27957766/how-do-i-render-a-word-document-doc-docx-in-the-browser-using-javascript

[9] GitHub - harshankur/officeParser: A Node.js library to parse text out of any office file. Currently supports docx, pptx, xlsx and odt, odp, ods..
https://github.com/harshankur/officeParser

[11] Schema | Tiptap Editor Docs
https://tiptap.dev/docs/editor/core-concepts/schema

[12] [13] Best Chunking Strategies for RAG in 2025
https://www.firecrawl.dev/blog/best-chunking-strategies-rag-2025

[14] [15] RAG
https://www.convex.dev/components/rag

[16] AI Agents | Convex Developer Hub
https://docs.convex.dev/agents

17 2024-06-13-webllm-a-high-performance-in-browser-llm-inference-engine.md
https://github.com/mlc-ai/blog/blob/c55b2f511a044ead3eb6c5d07ffaf727405bc69f/_posts/2024-06-13-webllm-a-high-performance-in-browser-llm-inference-engine.md

18 19 20 Transformers.js
https://huggingface.co/docs/transformers.js/en/index