ChatGPT

Microsoft Word vs Open-Source Editors: Handling Advanced Rich Text Features

1. Toolbar State Highlighting (Active Formatting States)

Microsoft Word's ribbon automatically highlights toolbar buttons for bold, italic, underline, etc., when the cursor is in text with those formats. List, alignment, and style buttons also reflect the current selection's state. This active-state syncing helps users see which formatting is applied.

Open-source web editors implement this in various ways: - **CKEditor 5 / TinyMCE:** These editors handle state highlighting out-of-the-box. The frameworks listen to selection changes and update UI toggles. For example, CKEditor's built-in toolbar will show bold/italic as pressed when active, and alignment buttons behave like radio buttons (only one alignment is active at a time). TinyMCE similarly updates its button states automatically via its core plugin system. - **ProseMirror / Tiptap:** With ProseMirror (which Tiptap is built on), developers typically subscribe to editor state updates and check if a given mark or node is active in the selection. Tiptap provides helper methods like `editor.isActive('bold')` or `editor.isActive({ textAlign: 'center' })` to query this. You can then conditionally add an "active" CSS class to your button. ProseMirror's core doesn't include a toolbar, but its architecture makes it possible to track marks and nodes in the selection. A common approach is using the editor's transaction or `selectionChange` event to update a React/Vue component state. For instance, in Slate (another framework), the recommended pattern is to use the editor context and re-render toolbar components on selection changes; this ensures the bold/italic buttons stay in sync [1]. ProseMirror offers utilities as well (e.g. checking `state.selection.$from.marks()` for active marks, or using commands that provide an `active` predicate). - **Slate (with React):** Slate doesn't come with a predefined toolbar, but it's straightforward to implement. By using the `useSlate()` hook, a toolbar component can re-render on any change. The active state of a formatting button can be determined by querying the editor's current marks or block type. For example, one can define an `isMarkActive(editor, 'bold')` utility that checks if the current selection has the bold mark, and then highlight the Bold button accordingly. Because the Slate toolbar can retrieve editor state via context, it re-renders whenever the editor changes, keeping the button states synchronized [1]. This pattern is essentially the same as Word's: the UI reflects the editor state continuously.

Best Practices & Libraries: If using ProseMirror directly, the community-provided menu example (`prosemirror-menu`) can automatically handle marking buttons active when their command or mark is active. Tiptap, being higher-level, has built-in menu components (or you can use their UI kit) that utilize `editor.isActive()` under the hood. For Slate, many use React components with context as described, or adopt the **Plate** framework (an extension of Slate) which offers ready-made toolbar components that include active state logic. Overall, leveraging the editor's state (via queries or context) is key to syncing toolbar highlights.

2. Indent/Outdent Functionality

In Microsoft Word: “Increase Indent” and “Decrease Indent” adjust the paragraph’s left indent or promote/demote list levels. For normal paragraphs, Word adds a specific indent (e.g. 0.5 inch by default) to the left margin of the paragraph. In lists, these buttons increase or decrease the nesting level of the list item. Word keeps indenting consistent and nested content (like a nested list or sub-paragraph) moves together when indented.

CKEditor 5: Indentation is handled by dedicated plugins. CKEditor 5 separates **block indentation** (for paragraphs, headings, etc.) from **list indentation**. The `Indent` toolbar buttons work in conjunction with the **IndentBlock** and **List** plugins [2]. When you press the Indent button on a paragraph, the IndentBlock plugin by default increases the left margin by a configured step (40px by default in CKEditor) [3]. You can configure the indent step in pixels, em, etc., or even use CSS classes for each level [4] [5]. For lists, CKEditor’s List plugin intercepts the indent command and instead increases the list nesting (wrapping the list item in a new sub-list). The user experience is seamless – the same **Indent/Outdent** buttons control both cases. Under the hood, if only the List plugin is loaded (without IndentBlock), the buttons will solely affect list nesting [2] [6]. With all plugins enabled, CKEditor will indent paragraphs by adding a margin or an indent class, and indent list items by increasing list depth. Developers can fine-tune this: for example, limiting max indent level or providing custom CSS classes for each indent level to enforce consistent spacing [7] [8]. (CKEditor even allows defining an explicit class per level, e.g. `.custom-indent-a` with 10% margin, `.custom-indent-b` with 20%, etc., to have fixed indent steps [9] [10].)

TinyMCE: TinyMCE has built-in `indent` and `outdent` commands available in its core toolbar [11] [12]. These will indent/outdent the current block or list item. By default, TinyMCE uses `<blockquote>` tags to indent non-list content (applying a blockquote style indentation), or increases/decreases list nesting for list items. However, TinyMCE can be configured to use margin-based indentation for paragraphs if desired. In practice, pressing Tab inside a list in TinyMCE creates a sub-list (indent), and Shift+Tab outdents, similar to Word. For paragraphs, the toolbar buttons or keyboard shortcuts will wrap/unwrap a blockquote or apply a CSS class to simulate indent. TinyMCE’s behavior is generally reliable for simple use, though it doesn’t expose as fine-grained a config as CKEditor’s IndentBlock (TinyMCE tends to assume default indent sizing, etc., unless you manually change the content CSS).

ProseMirror / Tiptap: Neither provides indent/outdent by default for paragraphs (lists are handled separately via the list item node). In ProseMirror, lists can be indented using the `sinkListItem` command (for nesting list items). For plain blocks, implementing indent requires a custom extension. A recommended approach (from ProseMirror’s author) is to give block nodes an `indentation` attribute and alter their rendering style [13]. For example, you can extend the paragraph node schema to include an `indent` level, and in the node’s `toDOM`, add a `style="margin-left: Xem"` or a class based on that level. This way, indenting is a semantic change (attribute in the document) rather than injecting raw spaces. Indeed, several developers have followed this approach: you add commands to increment/decrement the indent attribute on selected blocks. When the user presses Tab, if the cursor is in a paragraph (and not in a list), you call your “indent” command; Shift+Tab calls “outdent”. You also ensure that when a new paragraph is created after an indented paragraph, it inherits the indent attribute (which might require customizing the Enter/Split command). **Tiptap** (built on ProseMirror) doesn’t have an official indent extension as of Tiptap 2, but the community has created solutions. For instance, one community extension adds a global `indent` attribute to paragraph, heading, and listItem nodes and provides `editor.commands.indent()` and `outdent()`

to adjust it [14]. Internally this uses ProseMirror transactions to set a `data-indent` attribute on the DOM element (or you could use a style attribute) [15] [16]. This approach is quite flexible: you can define min/max indent levels, which nodes are indented, etc. (The example extension sets indent 0–4 and applies it to paragraphs, headings, list items.) With that in place, you get behavior very close to Word – multiple selected paragraphs can all indent/outdent together (since the command iterates through all nodes in the selection and updates their indent) [17] [18]. Keyboard shortcuts (Tab / Shift+Tab) can be mapped to these commands easily [14]. The end result: a `<p data-indent="2">` might be styled via CSS to have `margin-left: 2rem`, etc.

Slate (with Plate): If using Slate, the core doesn't preset indenting, but the Plate rich text framework provides an **IndentPlugin** that makes indenting blocks trivial. Plate's indent plugin injects an `indent` property into specified block types (paragraphs, headings, etc.) and handles Tab/Shift+Tab key events for you [19]. You can configure the indent unit (px, em, etc.), the step size, and maximum levels [20] [21]. Plate essentially does for Slate what IndentBlock does for CKEditor: it wraps the block nodes with styles or classes. This is very production-ready if you choose Slate – you'd just include the IndentPlugin (and any List plugin for list indentation) in your Slate editor setup.

Best Practices: To ensure reliable behavior, it's best to avoid indenting by simply inserting tabs or spaces (that can be unpredictable in HTML rendering). Instead, modify the element's structure or attributes. Using CSS classes or margin styles keeps the indent consistent and reversible. Also, integrate list indentation with paragraph indentation logic – e.g., in your Tab key handler, first check if inside a list (then increase list level), otherwise indent the block. Most modern editors follow this pattern. Another tip is to limit indent depth to keep documents neat (CKEditor's class-based config is a good example of enforcing a max indent level [22]).

3. Paragraph and Line Spacing Controls

Consistent spacing in documents involves controlling line height (within a paragraph) and spacing between paragraphs, as well as handling soft breaks (Shift+Enter) versus hard breaks (Enter).

Microsoft Word: Word lets users set **line spacing** (e.g. single, 1.15, double) and **paragraph spacing** (space before/after paragraphs). Word's default template, for example, uses ~1.15 line spacing and a bit of space after each paragraph. When you press Enter, Word starts a new paragraph which inherits the style (including spacing). Pressing **Shift+Enter** in Word inserts a manual line break (a `<w:br>` in OOXML) which does **not** start a new paragraph – so no extra spacing is added. This allows a single "paragraph" to have line breaks without the paragraph spacing in between. Word ensures that paragraph styles (like "Normal" or "Heading") define spacing so that the document has consistent gaps.

Web Editor Approaches: - **Default Behavior:** Most web-based editors treat Enter as a new paragraph (`<p>` or similar) and Shift+Enter as a line break (`<br>`). Ensuring the results look consistent requires setting CSS on those elements. For example, if you want an empty line between paragraphs like Word, you might give paragraphs a bottom margin (or use CSS `margin-block-end`). If using `<p>` tags, browsers by default have a margin, which can be adjusted in the editor's content CSS. Conversely, a `<br>` inside a paragraph will not create that margin gap, mimicking Word's Shift+Enter behavior. All the listed editors use `<br>` for soft breaks by default: CKEditor, TinyMCE, ProseMirror, etc., have this built-in. (In ProseMirror's schema, there is a `hard_break` node that renders as `<br/>` and is inserted on Shift+Enter by default keymap, so soft breaks are supported out of the box.) - **Line Height:** CKEditor 5 offers a **Line Height** feature

(premium plugin) that allows setting line spacing for content. If enabled, it provides a dropdown of line height options (similar to Word's line spacing menu). Underneath, it applies a CSS style or class to the paragraph (e.g. setting `line-height: 1.5` on that block). TinyMCE recently introduced a `lineheight` control as well [23] – a dropdown where you can choose "Single (1.0)", "1.5", "Double (2.0)", etc., and TinyMCE will apply that to the selected blocks. This typically works by wrapping the content or adding an inline style `style="line-height: 1.5;"` to the block element. If those plugins are not available, you can still achieve line spacing by injecting custom CSS: e.g., if all body text should be 1.15, set that in the editor's content stylesheet. For finer control (per paragraph), a custom style dropdown can be created to apply a class like `.spacing-double` which has `line-height: 2`. In ProseMirror, you can extend the paragraph node schema to include a `lineHeight` attribute and include it in the rendered style [24], similar to how one might do indent. Or simpler: allow a set of CSS classes via a menu (if using something like Tiptap's `Extension#addGlobalAttributes`, you could allow a `data-line-height` attribute and corresponding CSS). - **Paragraph Spacing:** None of the open-source editors explicitly expose "8pt spacing after paragraph" controls like Word UI, but you can achieve it. The simplest approach is to enforce a style globally: e.g., define in the editor's content CSS that `p { margin-bottom: 1em; }` (or a specific px value). That ensures consistent spacing after paragraphs. If you need the user to control this (say toggle "No Spacing" style vs normal), you could provide a style selector that switches classes on paragraphs: e.g., a "NoSpacing" style might set `margin-bottom: 0`. In CKEditor 5, you might use the **Styles** feature to create a style that the user can apply to paragraphs which sets a class removing margin. In TinyMCE, you can define custom style formats in the config, like a format named "No Spacing" that applies a class with no margin. Another approach is to treat it as part of document theme: Word's "No Spacing" style is basically the Normal style without extra spacing. - **Soft Break (Shift+Enter):** All these editors support Shift+Enter for a line break. Typically, after pressing Shift+Enter, you remain in the same paragraph element and a `<br>` is inserted. For example, in ProseMirror's default key bindings, Shift+Enter inserts a `hard_break` node which produces a `<br>` in the HTML. CKEditor and Tiny do similarly (in fact, in CKEditor 4/5 config you can specify whether Shift+Enter produces `<br>` or `<div>`, but `<br>` is default). The important thing for consistency is that your paragraph CSS should not add extra spacing for the `<br>`. If you rely on margins on `<p>` elements for spacing, then a `<br>` inside a `<p>` won't have that margin – thus it naturally has no extra gap, matching Word's logic. One pitfall to watch: If your editor uses `<div>` for paragraphs (some older HTML editors did), Shift+Enter might still create a `<br>`, which is fine. Just ensure that consecutive `<div>` blocks have spacing if needed. Modern editors mostly use semantic blocks like `<p>` or headings, which is good for styling and later export.

Implementing Controls: To give users a Word-like control over spacing, you can add UI options: - A **"Line spacing" dropdown** – if using CKEditor 5, their line-height plugin (premium) can be enabled. In TinyMCE, include `'lineheight'` in the toolbar (it's part of core in v6). For ProseMirror/Tiptap, you would create a custom dropdown that calls commands to set a line-height attribute or apply a class. (This could be done through a mark or node attribute. A node attribute is more semantic for a block-level line height.) - **Paragraph spacing** – you might create a toggle or styles: e.g., "Toggle paragraph spacing" that adds/removes a class on the paragraph to zero out margins. Another approach is to provide an adjustable "spacing after" field, but that gets complex in HTML. Most web-based editors instead offer only line-height, and rely on paragraph separation being uniform. If variable spacing is needed, it could be done via CSS classes (like `.mt-2` for margin-top, etc., if you allow multiple templates). - **Shift+Enter handling** – generally no action needed, just ensure your editor's output preserves `<br>` on export/print. (If you find extra `<p>` being created on Shift+Enter, check the configuration – e.g., some editors have a setting for how to handle single line breaks. By default, the ones listed handle it correctly as a line break.)

Printing Consideration: When printing HTML, browsers might not honor some CSS the same way (for instance, avoid using `px` for line-height because on print it might scale with DPI; use unitless like `1.15` or use `pt` for absolute). It's wise to include a print stylesheet that ensures consistent spacing (e.g., define page-breaks after headings or between certain elements if needed). But if your on-screen spacing is managed with simple CSS, it should print similarly.

4. Case Transforms (ALL CAPS, Small Caps) and Font Size Adjustments

All Caps and Small Caps in Word: Word provides a **Change Case** tool (e.g. to toggle UPPERCASE, lowercase, Title Case, etc.) and also a font effect for All Caps/small caps (in the Font dialog). When All Caps is applied as a font effect, Word still stores the text in its original case but displays it as uppercase. Small caps in Word similarly keeps letters as typed but renders lowercase letters as scaled capital letters. Word's "Increase Font Size" and "Decrease Font Size" buttons let users bump text up or down by one preset interval (e.g., 11pt to 12pt to 14pt, etc.).

Implementing these in web editors can be done reliably: - **All Caps:** The preferable method is to use a style (CSS) rather than actually changing the characters to uppercase in the document data. For example, applying a CSS rule like `text-transform: uppercase` to the selected text. Open-source editors typically do **not** have a built-in "all caps" toggle, but it's easy to add: - In **CKEditor 5**, the **Case Change** feature (premium) actually changes the text content case (it provides UPPERCASE, lowercase, Title Case options via a command) [25] . It's meant as an editing action (you can undo it). However, if you wanted a true toggle that can be turned off, you'd instead create a custom inline style. CKEditor's General HTML Support or a custom plugin could allow a span with `style="text-transform: uppercase"` to be applied. Alternatively, you can utilize the **Styles dropdown** to define an "All Caps" style that wraps text in a `<span class="all-caps">` (and in your content CSS, define `.all-caps { text-transform: uppercase; }`). This way, the underlying text remains unchanged (so toggling off restores the original casing). - **Tiptap/ProseMirror:** You can define a Mark type (inline format) called "caps" which, in `toDOM`, renders as `["span", { style: "text-transform: uppercase" }, 0]`. Add a command to toggle this mark on the selected text. This would function similar to bold/italic marks. The benefit is that the original text is preserved. If the user types new text with that mark active, it will appear uppercase (via CSS) but actually be stored as typed. This approach covers All Caps and is reliable. (If the user applies it to "Hello" it will visually become "HELLO"; if they turn it off, "Hello" appears again). - **Slate:** You could achieve this by adding a custom "mark" for uppercase – essentially a property on text nodes that, when rendering, you wrap or style accordingly. Or use a span with a React component that applies uppercase style. Plate (Slate's library) does not have built-in case toggle, so it would be a custom addition. - **Direct Text Transformation:** An alternative some implement is directly converting text to uppercase on command (like a one-time action). For example, a ProseMirror plugin can iterate through the selected text nodes and replace them with their `.toUpperCase()` content [26] [27] . This is more like an "Edit -> Change Case" operation (you can't easily revert it without undo). It's fine for actions like Title Case or if using a cyclic case-change shortcut (Shift+F3) as Word does [25] . But for a toggling behavior, CSS is preferable.

- **Small Caps:** Small caps is best done via CSS `font-variant: small-caps;`. Similar to All Caps, you could wrap text in a span with that style. None of the mentioned editors have a default small-caps button, so you'd implement it through a custom style or extension. For instance, define a mark or style called "SmallCaps" that applies `font-variant: small-caps`. Modern browsers support

this well (as long as the font has lowercase glyphs, it will synthesize small caps). Word's small caps is essentially this effect. By using CSS, you ensure the text isn't actually changed to caps (which would lose information like proper nouns).

- **Font Size Changes:** All listed editors support setting font size on text:

- **TinyMCE:** It has a dropdown (`fontsizeselect`) that lists sizes (e.g., 8pt, 10pt, 12pt, etc.) and applying one wraps the selection in `<span style="font-size: 12pt;">` by default [28] [29] . TinyMCE also provides an optional `fontsizeinput` tool which shows the current size and has small **A⁻/A⁺** buttons to decrease or increase the size by one step [30] . This mimics Word's grow/shrink font buttons. You can configure the presets for the dropdown (via the `fontsize_formats` setting) and TinyMCE will only allow those sizes, or allow any value via the input.
- **CKEditor 5:** The **Font Size** feature (part of the Font plugin) similarly offers a dropdown of sizes [31] . By default it might use names like "Small, Default, Big" which map to specific pixel values, but you can configure it with explicit numbers (e.g., 11, 12, 14, 18pt). CKEditor's implementation applies either a class or an inline style on a `<span>` for the font size [31] . The choice of class vs style is configurable – e.g., it can use classes like `.font-size-big` defined in the content CSS for consistency, or direct styles for flexibility. In either case, it ensures the selection (or new text) is wrapped so that only that portion is resized, not the whole paragraph.
- **ProseMirror/Tiptap:** Implementing font size can be done with an inline style mark (similar to color or font-family). Tiptap actually provides a `TextStyle` extension which essentially allows any CSS style on text. You could use that to set a `style="font-size: 18px"`. For a more structured approach, you might restrict it to certain classes or data attributes. For example, add a mark type that has an attribute for size, and in `toDOM` map that to a class like `text-small` / `text-large` or a style. Since ProseMirror marks can have attributes, you can store the exact size value.
- **Slate/Plate:** Plate has a FontSize plugin that works similarly – it injects a mark with a `fontSize` style. If you use Plate's built-in UI, you get a dropdown and it applies the chosen size on the selection.

Ensuring Reliability: One tricky aspect is making sure the size or case change applies cleanly even if the selection is across multiple nodes. The editors usually handle this by splitting nodes as needed and wrapping each part. For example, if you select text spanning two paragraphs and set font size 18, the editor will likely wrap each paragraph's portion separately in a span. This is normally fine. When increasing/decreasing font size with a button, ensure you define what the "steps" are. (In Word these steps aren't linear – they jump through a list of standard font sizes. You can mimic this by coding a list of sizes and picking the next/previous when the button is pressed. E.g., from 11px to 12px to 14px, etc., or simply +1px as a simpler approach.)

Also, consider interplay with themes: if a document theme uses relative font sizing, applying an absolute font size might break that. One strategy is to use CSS custom properties for base font sizes – but that's probably overkill. Generally, treat these features as direct formatting: e.g., **apply all-caps** (style or transform) to selected text, **set font size X** on selected text. As long as you maintain them as separate spans/marks, they will be preserved on export to HTML or conversion to DOCX (more on that below).

Code-Level Tips: - Use existing plugins when possible (CKEditor's Font and CaseChange, TinyMCE's built-ins). These are well-tested. - When writing your own for ProseMirror/Tiptap, follow patterns similar to bold/italic marks. E.g., a command that either adds the mark to the selection or removes it if already applied.

ProseMirror's `toggleMark` can be used if you register your mark in the schema. - If you use classes for font size (like `fs-18`), you might also need to include those styles in the print or export CSS so that they have effect outside the editor. Inline style attributes will carry over naturally in HTML output.

5. Document Themes and Styles (Word vs Web)

Microsoft Word's Themes & Styles: Word uses a powerful style system to ensure documents have a consistent design. Key concepts: - A **Style** in Word is a named set of formatting. For example, "Heading 1" style might be defined as 16pt bold Calibri with Accent 1 color and 24pt space after. Text tagged as Heading 1 will all follow that definition. If the style definition changes, all that text updates. - Word distinguishes paragraph styles (which apply to a whole paragraph and can include spacing, indent, etc.) and character styles (for inline formatting, though "All Caps" can be a character style property). - **Themes:** A theme in Word (OOXML) defines a **color palette** (12 colors: Dark/Light for text/background, Accent 1–6, Hyperlink and Followed Hyperlink colors) and **theme fonts** (usually two fonts: one for headings and one for body, often called "+Headings" and "+Body"). Word's built-in styles often reference the theme fonts and colors instead of hard-coding them. For instance, *most* of the built-in heading styles use the **+Headings font**, and normal text uses the **+Body font** [32]. This means if you change the theme's font (say from Calibri to Times for body text), all those styles update automatically to the new font [33] [34]. Similarly, many style definitions use theme accent colors (e.g., Heading 1 might be "Accent 1 color, Darker 25%"). Changing the theme color scheme swaps out the actual RGB values used. - **Mapping in OOXML:** In the document's XML, styles like Heading 1 might have `<w:rFonts w:asciiTheme="heading" w:eastAsiaTheme="heading" .../>` to denote it uses theme heading font, and `<w:color w:val="accent1"/>` for color. The theme itself (stored in `theme/theme1.xml` in the .docx) defines what "accent1" actually is (e.g. blue) and what the heading/body fonts are named. Word's hyperlink style is also theme-aware – by default it uses the Hyperlink color (blue) and underline. There's a lot of intelligence: for example, Word knows to use a darker shade of Accent 1 for headings if the theme defines one, as a contrast against the background [35] [36].

How Theme Fonts/Colors Map to Headings, Body, Links: As noted, *Heading* styles map to the theme's Heading font, *Body* (normal text) maps to the Body font [37] [32]. By default, Word's "Normal" style uses +Body font, and all built-in headings (1–9) use +Headings font [32]. The actual font names (e.g., Calibri) are pulled from the theme. For colors, Word typically uses: - **Accent 1** for headings or strong emphasis, **Accent 2–6** for other elements (maybe different heading levels or content like tables, etc., depending on the style set). - **Hyperlink** color for hyperlinks (the theme defines this, often it's Accent 5 or Accent 6 by default blue). Indeed, Word's Hyperlink style is linked to the theme's Hyperlink color. If you change the theme to one with, say, green hyperlink, all link text in the document will turn green automatically. - The combination of theme + styles yields a **style set** (what Word calls Quick Styles gallery). For example, "Title" style might be +Headings font, Accent 2 color, 28pt, centered. All these choices come from the theme's resources.

In Web Editors: There isn't an out-of-the-box equivalent to Word's theme mechanism – HTML/CSS has no concept of "theme fonts" that automatically propagate through named styles, aside from manually changing CSS. However, we can emulate some aspects: - **Global CSS / Themable Design:** One approach is to use CSS custom properties or a CSS framework to represent theme colors and fonts. For example, define CSS variables like `--theme-body-font`, `--theme-heading-font`, `--theme-accent1`, etc., at the root. Then in your content styles, use those: e.g., `p { font-family: var(--theme-body-font); }`, `h1, h2, .heading1 { font-family: var(--theme-heading-font); color: var(--theme-accent1); }`, `a { color: var(--theme-hyperlink-color); text-decoration: underline; }`. When the user selects a different theme, you update these CSS variables (or swap out the stylesheet) to the

new values, and the content restyles instantly. This mirrors how Word applies a theme change. - **Editor Style Sets:** CKEditor 5 has a **Styles** feature which is more about applying CSS classes to content (for consistent formatting chunks), but it doesn't inherently link to a theme. However, CKEditor's **Export to Word** functionality demonstrates mapping HTML/CSS to Word styles: for example, it recognizes that an `<h1>` in HTML corresponds to Word's Heading 1 style [38], or that an `<a href="...">` should become Word's Hyperlink style [38]. This means if you use semantic tags and consistent classes, you can export a .docx that uses the proper styles. For instance, the export converter will see `<h1>` text and assign it the Heading1 style in the generated DOCX [38] [39]. Similarly, a `<p>` with a certain class could map to a custom Word style if configured. - **Applying Themes in Editors:** If you want users to be able to choose a theme (like a set of fonts/colors) for their document in your web app, you have to implement it at the app level. For example, define a JSON structure for "Theme" with font names and a palette. When the user picks one, load a CSS specific to that theme into the editor's iframe or style tag. Tiptap/ProseMirror allow you to programmatically change the content's styling via updating the parent page's CSS or the editor's style module. TinyMCE lets you specify a `content_css` file – you could generate that CSS based on the theme. For instance, if the theme's Body font is "Arial" and Heading font "Georgia" with certain colors, you'd produce content CSS that sets `body, p { font-family: Arial; } h1,h2, .heading1,.heading2 { font-family: Georgia; color: #123456; } a { color: #654321; }`. - **Predefined Templates:** Another strategy is to create templates with classes that mimic Word styles. For example, define CSS for classes named `.WordHeading1`, `.WordNormal`, etc., reflecting a particular look. Apply those classes to elements accordingly. This is somewhat like how **LibreOffice or Google Docs** HTML exports work (they often export with class names corresponding to styles). You could even allow a user to import a .docx, parse its styles (with a library), and then apply corresponding classes in the editor. There are libraries like **Mammoth.js** which convert .docx to HTML while preserving some style semantics (often mapping Word styles to classes or inline styles). Conversely, for going back to .docx, you could map your HTML classes back to Word styles using a tool or custom code. For instance, CKEditor's export is doing a similar mapping internally [40] [38]. - **Color Palettes:** For font and background color pickers, you can configure them to use theme colors. CKEditor 5's font color plugin can be configured with a set of colors – you could supply the theme's palette so that users only pick from those (and maybe an "Automatic" which corresponds to Text/Background). TinyMCE's textcolor picker also can be fed a custom color map. By doing so, you ensure users are using the theme colors (so if the theme changes, ideally those color codes would update – though in practice, if you applied a concrete hex code, changing theme won't change that hex code in the content). A more dynamic approach: have classes for "Accent1" etc., and apply those to text instead of direct color. Then the CSS for `.accent1` sets the actual color value, which can change with the theme CSS. This is complex to manage but is closest to Word. For example, `<span class="accent1">Important</span>` and in CSS: `.accent1 { color: var(--theme-accent1); }`. Switching theme updates `--theme-accent1`. This way "Important" text could automatically change color when theme changes. This mimics the Word behavior of theme-aware coloring [33].

Consistent Style Application: To achieve "styles applied consistently across elements," the simplest method is to **restrict formatting choices and use styles**. That is, define a set of block styles (Normal, Heading 1-6, Quote, etc.) and inline styles (maybe Emphasis, Strong, etc.) that the user can apply, rather than having them individually choose font size, font, color for each bit. This is what Word encourages via the Styles gallery. In web editors, you can implement something similar by providing a dropdown of styles. CKEditor 5's **Styles** plugin or **Heading** dropdown partially covers this (e.g., a dropdown for paragraph vs heading levels, which maps to `<p>, <h1>, <h2>, ...` in output). You can add custom styles to CKEditor's style dropdown, like "Subtitle" that wraps a `<p class="subtitle">` with certain styling. In TinyMCE, you have the format dropdown (blocks) and you can also configure the style format options.

Tiptap/ProseMirror would need a custom UI to select styles (one could implement a menu that, for example, when “Heading 1” is chosen, it transforms the current block to an `<h1>` or applies a specific CSS class).

Using semantic tags for major styles (h1..h6, blockquote, etc.) is advantageous because, as noted, many tools will recognize them. For instance, when exporting to PDF or Word, you can map those easily. CKEditor’s Word export explicitly maps `<h1>-<h6>` to Word’s Heading styles [38], `<blockquote>` to Quote, etc., which ensures consistency.

Ready-Made Components/Libraries for Themes: There isn’t a plug-and-play “theme manager” for ProseMirror or Tiptap widely available. It mostly comes down to CSS. However, you might leverage existing **CSS frameworks** if you’d like: e.g., Bootstrap or Tailwind could define a set of typography classes. Tailwind, for example, could let you apply classes like `text-xl` or `font-serif` which you could switch via a theme toggle. But mapping that to Word’s concept of theme would be manual. There are also libraries for reading OOXML – for instance, **docx.js** (an NPM library) can create Word files; you could potentially feed it your theme info to generate a `.docx` with the correct theme part. If you needed to import/export themes, you’d have to parse the theme XML.

DOCX/PDF Export Considerations: Since the goal is to export to DOCX and PDF: - If you use **CKEditor 5**, the premium Export to Word plugin will handle a lot of this for you – it converts HTML+CSS to a .docx, mapping to Word styles where possible (as we discussed) [41] [40]. It will embed your theme fonts/colors as direct formatting unless it detects a pattern matching a style. It can also embed the used fonts. - Without that, you can use libraries or services. For DOCX, one approach is to take the content HTML and run it through a converter like **Mammoth** (which tries to produce a clean docx, but you might lose some style fidelity), or use a lower-level library like `docx` (which lets you construct a docx with styles programmatically – you could map your semantic HTML elements to Word style definitions in code). - For PDF, you can either print the HTML to PDF (using the browser or a headless browser, which will respect your CSS), or use a dedicated PDF renderer (like **pdfmake** or **paged.js** if you need fine control over pagination, headers, etc.). Make sure your content CSS is print-friendly (for example, use `cm` or `in` for page margins if using paged.js).

Choosing CKEditor/TinyMCE vs ProseMirror (Tiptap) Approach:

CKEditor 5 / TinyMCE (Word-like “shell”): These are feature-rich and will get you closest to Word functionality fastest. **Pros:** They have a wide array of built-in plugins (lists, indent, alignment, tables, images, footnotes, etc.) and UI elements (toolbars, dropdowns) that you can configure rather than build from scratch. They handle the tricky parts of selection, focus, clipboard (pasting from Word with formatting – CKEditor even has a Paste-from-Office filter). They also provide official solutions for exporting to PDF/DOCX – for example, CKEditor’s Export to Word we discussed, or its collaboration plugins, etc., which are well-supported [41]. TinyMCE has community and some premium offerings (e.g., export to PDF and even a Word export via third-party or their cloud). Using these, you can be confident in cross-browser behavior and accessibility. **Cons:** They can be heavy (larger bundle sizes). Also, some advanced features are behind a commercial license (CKEditor’s export, track changes, etc., require a paid license unless your project is open-source GPL). If you need to deeply customize behavior or schema, it can be harder – you’re limited to their extension APIs. TinyMCE and CKEditor allow custom plugins, but you work within their framework.

ProseMirror/Tiptap (DIY approach): Pros: Extreme flexibility. You define the schema (document structure), so you can represent the document in a more semantic way. For example, you could have a node for "Heading" that has an attribute for the level, rather than just using `<h1>` HTML – this might make it easier to manage in code. You can also enforce constraints (ProseMirror ensures the document adheres to the schema). It's easier to integrate custom domain-specific elements if needed (like say "comment" nodes or placeholders). Tiptap, specifically, provides a high-level API and many ready-made extensions (bold, italic, underline, strike, code, blockquote, list, table, image, textAlign, link, etc.), so you're not completely on your own. The developer experience with Tiptap is quite good if you use Vue/React. **Cons:** To match feature parity with Word, you have to assemble many parts and possibly write some yourself. As we saw: indent we'd craft an extension for, line spacing too, small caps toggle – all custom. There is a community extension for indent, and maybe for line-height or font-family (often solved by the generic TextStyle). It requires more development effort and careful testing. Also, ProseMirror has a learning curve (especially around its transaction-based state and schema). Tiptap abstracts some of it but if something goes wrong, you might need to dive into ProseMirror understanding. Another consideration: collaboration – ProseMirror can be used with operational transform or CRDT solutions (like Yjs), but that's a whole additional layer; CKEditor and Tiny offer ready collab solutions if needed. For a single-user editor, that may not matter.

Slate (with Plate): This is another route if you are into React. Plate provides a lot of pre-built rich text features (indent, alignment, bold, tables, etc.) similar to CKEditor, but as React components/hooks. It's fairly robust and MIT licensed. The **pros** are that it integrates naturally in React apps and you can use React components as part of your editor output (for custom elements). **Cons:** Historically, Slate had performance issues with very large documents or complex operations, but Plate has mitigated some of that with perf optimizations. It's an option if you prefer a middle ground: not as heavy as CKEditor, more structured than ProseMirror perhaps, but with lots of features.

Recommendation Summary: If your priority is to **quickly support a full Word-like feature set** including solid DOCX/PDF export, and you don't mind using a partially commercial solution, **CKEditor 5** is a strong choice. It's actively maintained, and as of 2025 has a large community. You could leverage the GPL version for your project (if your project's license allows) or get a commercial license for the convenience of export and support. CKEditor will give you toolbar state syncing, indent, spacing, font styles, etc., mostly by enabling plugins and configuring. For example, enabling the indent plugins and line-height (premium) plugin, and using the font plugin for font size and all-caps (you might implement all-caps as a custom style). The Export to Word feature would handle mapping your headings and such to real Word styles [40] [42] – which is a big plus for high fidelity DOCX output. **TinyMCE** is also capable, though its ecosystem for export is less built-in (there are external converters or you'd do HTML->DOCX yourself). TinyMCE is very stable and also has a modern v6.

If your goal is **maximum control and integration** (for example, your app has very custom requirements or you want to store documents in a structured format), then **ProseMirror/Tiptap** is worth it. You can integrate components piecemeal, e.g., use Tiptap for core editing and perhaps use an existing library like **html-to-docx** or **Pandoc** for exports. Be prepared to implement some features manually (like the theme system – which in practice means dynamically swapping CSS as discussed).

Given it's a single-user editor (not Google Docs collaboration), the complexity is a bit lower – you won't need real-time conflict resolution, etc. So a heavy framework is acceptable. On balance, many developers choose an existing solution (CKEditor/Tiny/Plate) to save time, unless they specifically need something those can't do.

Finally, ensure to **store your documents in a robust format** in your portal. HTML with semantic tags and data-attributes for additional metadata is a common choice. Some store a JSON (ProseMirror's JSON or Slate's JSON) for fidelity, but that then needs conversion on export. Storing HTML or Markdown plus some front-matter for theme could suffice. If DOCX export is critical, you might even consider storing as DOCX (some apps do that, editing the docx with an API) – but that's cumbersome for live editing. It's usually easier to store in HTML and convert on demand for export.

In summary: Use the **best-supported components** for each feature – e.g., CKEditor's built-in indent and font plugins, or Tiptap's community extensions – rather than reinventing if they exist. Where not available, follow the best practices above (like CSS for case toggles, node attributes for spacing). By doing so, you'll create a web-based word processor that approaches the richness of Word while remaining maintainable.

[32] [42]

---

[1] Rendering | Slate
https://docs.slatejs.org/concepts/09-rendering

[2] [3] [4] [5] [6] [7] [8] [9] [10] [22] Block indentation | CKEditor 5 Documentation
https://ckeditor.com/docs/ckeditor5/latest/features/indent.html

[11] [12] [23] [28] [29] [30] Toolbar Buttons Available for TinyMCE | TinyMCE Documentation
https://www.tiny.cloud/docs/tinymce/latest/available-toolbar-buttons/

[13] Indent/Undent Nodes - discuss.ProseMirror
https://discuss.prosemirror.net/t/indent-undent-nodes/1150

[14] [15] [16] [17] [18] javascript - indent action in tiptap 2 - Stack Overflow
https://stackoverflow.com/questions/75902464/indent-action-in-tiptap-2

[19] [20] [21] Indent - Plate
https://platejs.org/docs/indent

[24] How to update the basic schema's paragraph to support line-height and text-align? - discuss.ProseMirror
https://discuss.prosemirror.net/t/how-to-update-the-basic-schemas-paragraph-to-support-line-height-and-text-align/3000

[25] Case change | CKEditor 5 Documentation
https://ckeditor.com/docs/ckeditor5/latest/features/case-change.html

[26] [27] Some pointers in creating a casing plugin - discuss.ProseMirror
https://discuss.prosemirror.net/t/some-pointers-in-creating-a-casing-plugin/2805

[31] Font family, size, and color | CKEditor 5 Documentation
https://ckeditor.com/docs/ckeditor5/latest/features/font.html

[32] [33] [34] [37] +Body and +Heading | PC Review
https://www.pcreview.co.uk/threads/body-and-heading.4041123/

[35] [36] Accent 1 color in my theme not being used in headings - Microsoft Q&A
https://learn.microsoft.com/en-us/answers/questions/5020337/accent-1-color-in-my-theme-not-being-used-in-headi

[38] [39] [40] [41] [42] Styles - Export To Word | CKEditor Cloud Services Documentation
https://ckeditor.com/docs/cs/latest/guides/export-to-word/styles.html