ChatGPT

diffDOM Operation Patterns and Production Usage

1. Operation Type Catalog (with real-world examples)

diffDOM produces a list of **diff operations** to transform one DOM into another. Each operation is an object with an `action` type and properties like `route` (path to the affected node) and relevant values. Below is a catalog of all operation types in diffDOM, including their structure, when they occur, and examples:

- **addElement** – *Insert a new element node.*
  Structure: `{ action: 'addElement', route: [..parentIndex, newIndex], element: {...} }` where `element` is a **virtual DOM object** representing the entire new subtree. The `route` points to the parent and index where the node should be inserted.
  Triggers: When a node is present in the **toDOM** but not in the **fromDOM**. For example, adding a new `<p>` tag.
  Minimal Example: Original HTML: `<div></div>`; New HTML: `<div><p>Hello</p></div>`. The diff will include an `addElement` for the `<p>` at `route: [0]` (insert at index 0 under the `<div>`).
  Production Example: In collaborative editors like *Fidus Writer*, when a user inserts a new section or figure, diffDOM generates an `addElement` containing the entire new section subtree. The diff entry's `element` property holds all nested content so the addition can be applied in one operation. This operation is **common** in document edits whenever new elements (paragraphs, list items, etc.) are created.

- **removeElement** – *Remove an existing element node (and its subtree).*
  Structure: `{ action: 'removeElement', route: [..targetNodePath], element: {...} }`. The `route` points to the element to remove; `element` contains the removed node's structure (used for undo or reference).
  Triggers: When a node exists in the old DOM but not in the new DOM. E.g. a paragraph or div deleted.
  Minimal Example: Original: `<ul><li>A</li><li>B</li></ul>`; New: `<ul><li>A</li></ul>`. The second `<li>` is gone, producing a `removeElement` with `route: [0,1]` (the second `<li>` under the `<ul>`).
  Production Example: In rich text editors, deleting a content block (like removing a citation element) yields a `removeElement` diff. The operation includes the full content of that element, so the removal is captured as one diff. **Frequency:** Frequent when users delete structured content. diffDOM optimizes by treating a whole subtree deletion as a single operation (it doesn't list removals for each child), reducing noise.

- **replaceElement** – *Replace one element node with another.*
  Structure: `{ action: 'replaceElement', route: [..targetPath], oldValue: {...},`

`newValue: {...} }`. It contains the entire old node and new node in their virtual DOM forms.
Triggers: When diffDOM finds a node in the same position but with a different tag or fundamentally different structure (not similar enough to treat as modifications). For example, an `<img>` tag changed into a `<figure>` with a caption could be a replace. If a text node is turned into an element (e.g. wrapping text in `<strong>`), diffDOM may output a `replaceElement` of the text node with a new element node.
Minimal Example: Original: `<span>Text</span>`; New: `<div>Text</div>`. diffDOM will likely produce `replaceElement` at the `<span>`'s route with `oldValue: {nodeName: 'SPAN', ...}` and `newValue: {nodeName: 'DIV', ...}`.
Production Example: This is **less common** than add/remove, but appears in document conversions or user formatting changes. For instance, in *WYSIWYG editors*, changing an inline text into a header tag might register as replacing one element with another. The entire new element (with its attributes/children) is included in `newValue`. When applying, diffDOM simply calls `parent.replaceChild(newNode, oldNode)`. Undo is supported by swapping `oldValue`/`newValue`.

- **addTextElement** – *Insert a new text node.*
  Structure: `{ action: 'addTextElement', route: [..parentPath, textIndex], value: "text content" }`. The `value` holds the string to insert.
  Triggers: When plain text is added where there wasn't a text node before (or a text node is split). For example, inserting a piece of text in an empty element or between existing nodes.
  Example: Original: `<p>Hello</p>`; New: `<p>Hello world</p>`. If the change created a new text node (say the space+"world" was a new node), it produces `addTextElement` for `" world"`. In most cases, though, adding text within an existing node is just a modify (see below). But if you add a separate text node child (like adding text after a `<br>`), this operation appears.
  Production Example: In diffing HTML, pure text insertions are usually output as text node modifications (if continuous). However, some systems split text for annotation. For instance, *visual diff tools* might intentionally break text nodes to isolate a change; then an inserted word could be an `addTextElement`. **Frequency:** moderate, often accompanied by `removeTextElement` if text was split or rejoined.

- **removeTextElement** – *Remove an existing text node.*
  Structure: `{ action: 'removeTextElement', route: [..textNodePath], value: "text content" }`. The `route` points to the text node, and `value` (or sometimes `oldValue`) contains its content for reference/undo.
  Triggers: When a text node present in the old DOM is gone in the new DOM. E.g. a piece of text deleted, or two text nodes merged (one disappears).
  Example: Original: `<p>Hello <em>world</em></p>`; New: `<p>Hello </p>` (removed the emphasized "world"). This yields a `removeTextElement` for the `"world"` text node (assuming the `<em>` tag also removed via `removeElement`). In a simpler case, if a trailing text node is dropped entirely, you get a `removeTextElement`.
  Production Example: Common in text edits. For instance, *Matrix/Element* message edits that delete words produce `removeTextElement` diffs for those words (often alongside `addTextElement` for inserted words). This operation is often paired with `addTextElement` or `modifyTextElement` as text is re-written. Real-world diff viewers may highlight the removed text on the left pane (e.g. strike-through "world" on the original side).

- **modifyTextElement** – *Modify the content of a text node.*
  Structure:
  `{ action: 'modifyTextElement', route: [..textNodePath], oldValue: "old text", newValue: "new text" }`.
  Triggers: When the text content of a node has changed (characters altered but the text node itself remains). This is the most common diff for edited text.
  Example: Original: `<p>Hello **worlt**!</p>`; New: `<p>Hello **world**!</p>`. Only one letter changed, so diffDOM outputs a single `modifyTextElement` for that text node with `oldValue: "worlt", newValue: "world"`.
  Production Example: Very frequent in document editing and messaging apps. It represents in-place text edits. For instance, in *Matrix's Element* client, when showing an edited message, a `modifyTextElement` diff indicates what text changed. Production systems often post-process this diff for visualization – e.g. using **Google's diff-match-patch** to highlight the specific added and removed characters within the text. Instead of marking the whole sentence as changed, they break down the diff to wrap inserted/deleted substrings in `<ins>`/`<del>` for a user-friendly view.

- **addAttribute** – *Add a new attribute to an element.*
  Structure: `{ action: 'addAttribute', route: [..elementPath], name: "attrName", value: "attrValue" }`.
  Triggers: An attribute that did not exist before is now present. For example, adding a `class` or `style` attribute that wasn't there in the original.
  Example: Original: `<img src="cat.png">`; New: `<img src="cat.png" alt="A cat">`. The `alt` attribute addition produces `addAttribute` with `name: "alt", value: "A cat"`.
  Production Example: Adding formatting or data attributes is common. In a CMS like *Fidus Writer*, toggling a setting might introduce a new attribute (e.g., `data-citation-id`). The diff will capture it as an addition. This is usually visualized by showing the new attribute on the right pane highlighted (often in **green** for "added"). The left pane (old DOM) has no such attribute, so nothing corresponding is highlighted there.

- **removeAttribute** – *Remove an existing attribute from an element.*
  Structure:
  `{ action: 'removeAttribute', route: [..elementPath], name: "attrName", value: "oldValue" }`. The `value` (or `oldValue`) holds the attribute's prior value.
  Triggers: An attribute present in the old DOM is missing in the new DOM. For example, an inline style or class was removed.
  Example: Original: `<input type="text" disabled>`; New: `<input type="text">`. Removal of the `disabled` attribute yields `removeAttribute` with `name: "disabled", value: ""` (value might be empty or the old value if it had one).
  Production Example: Often appears when users clear formatting or when default attributes get dropped. For instance, a real-time editor might remove a `style="..."` attribute when a style is reset. On the left pane (before), the attribute is shown (highlighted in **red** or strike-through to indicate deletion), while on the right it's absent. Undo is straightforward: diffDOM stores the removed value so it can be re-added if needed.

- **modifyAttribute** – *Change the value of an existing attribute.*
  Structure:
  `{ action: 'modifyAttribute', route: [..elementPath], name: "attrName",`

`oldValue: "old", newValue: "new" }`.
Triggers: The attribute exists in both DOMs but its value/string changed. E.g. class name changed, src URL changed, style text changed.
Example: Original: `<a href="page1.html">Link</a>`; New: `<a href="page2.html">Link</a>`. This produces `modifyAttribute` on the `<a>` with `name: "href", oldValue: "page1.html", newValue: "page2.html"`.
Production Example: Very common in diffs of edited content. Changing a heading's `id`, editing an image's `src`, or modifying a CSS class on an element will all yield `modifyAttribute`. In *visual HTML diff tools* (and dual-pane viewers), the changed attribute value is highlighted on both sides (e.g., original value "page1.html" marked as changed on the left, new value "page2.html" on the right). Some systems further **refine** this by highlighting the specific substring difference within the attribute (especially for long strings like class lists or URLs). For instance, if a class list changed from `"btn primary"` to `"btn secondary"`, a viewer might highlight just the word that changed.

- **modifyValue** – *Change in an element's* `.value` *property (typically form controls).*
Structure: `{ action: 'modifyValue', route: [..elementPath], oldValue: "...", newValue: "..." }` (where old/new are the control's value).
Triggers: When using `valueDiffing: true`, diffDOM compares form field values (like user-entered text in inputs, or option selections). If a form field's current value differs between DOMs (even if the HTML attribute didn't change), it outputs a `modifyValue`. For example, the user typed in a text field or changed a slider.
Example: Original DOM: `<input type="text" value="John">` (user's unsaved value "John"); New DOM: `<input type="text" value="Jane">`. If the `value` attributes differ, diffDOM might treat it as `modifyAttribute`. However, if the HTML `value` wasn't updated but the live DOM's property was (common if user typed but not saved to attribute), diffDOM still catches it via `modifyValue`. E.g., old had `value="John"` attribute, user typed "Johnny" (DOM property now "Johnny"), new snapshot has `value="Johnny"`, diffDOM outputs `modifyValue` with `oldValue: "John", newValue: "Johnny"`.
Production Example: In collaborative apps, this helps track unsaved form changes. It's not very common in static document diffs, but *Matrix/Element* may use it when diffing contenteditable message drafts or form inputs. Visualization can be tricky because it's an internal state change: often one will treat it like an attribute change on the `value` attribute for display purposes (showing the old vs new text). If showing two versions of a form, the left pane might show the input's old value text highlighted, and the right pane the new text.

- **modifyChecked** – *Checkbox or radio checked state changed.*
Structure: `{ action: 'modifyChecked', route: [..inputPath], oldValue: false, newValue: true }` (or vice versa).
Triggers: When a checkbox or radio input's `checked` property differs between the two DOMs (with `valueDiffing` on). For example, a checkbox got toggled on or off. HTML's `checked` attribute might not be present or changed, so diffDOM explicitly notes the property change.
Example: Original: `<input type="checkbox" name="opt">` (unchecked); New: `<input type="checkbox" name="opt" checked>` (checked). If the `checked` attribute actually appears/disappears, diffDOM could output an attribute add/remove. But often, frameworks don't update the attribute, only the DOM property, so diffDOM yields `modifyChecked` with `oldValue: false, newValue: true`.

Production Example: Seen in form diffs or stateful UI diffs. For example, *React state hydration diff* might reveal that a checkbox was checked. In a dual-pane viewer, one might show the checkbox on left as unchecked and on right as checked (perhaps with a special highlight or an added "checked" marker on the right). Some diff tools simply represent this by adding a `checked` attribute highlight on the right side.

- **modifySelected** – *Option selection state changed.*
  Structure: `{ action: 'modifySelected', route: [..optionPath], oldValue: false, newValue: true }` (or vice versa).
  Triggers: When an `<option>` element's selected state differs. E.g. a different item in a `<select>` was chosen. As with `checked`, this catches state changes not reflected as actual DOM differences in the markup (since `<option selected>` might not accurately represent current UI selection unless updated).
  Example: Original: `<option value="1" selected>One</option> ... <option value="2">Two</option>`; New: `<option value="1">One</option> ... <option value="2" selected>Two</option>`. diffDOM may output a `modifySelected` on the two `<option>` elements: first goes from true to false, second from false to true. (It might also represent this as one selected attribute added and one removed; but with valueDiffing, it tends to use `modifySelected` properties).
  Production Example: In web apps diffing form states, this shows selection changes. A diff viewer could highlight the newly selected option on the right and possibly mark the previously selected on the left. Accessibility considerations are important (ensuring screen readers know an option changed selection).

- **modifyComment** – *Comment node content changed.*
  Structure:
  `{ action: 'modifyComment', route: [..commentNodePath], oldValue: "old comment", newValue: "new comment" }`.
  Triggers: If an HTML comment's text is different between the versions. Comments aren't usually user-edited content, but diffDOM will catch it.
  Example: `<!-- TODO: old note -->` changed to `<!-- TODO: new note -->` produces a `modifyComment` diff.
  Production Example: Rare in typical user-facing diffs (since comments are invisible), but could matter in HTML code diffs or template changes. Tools might simply display the changed comment text in diff output (perhaps only in code mode). This is typically low-frequency and often filtered out in visual diffs unless specifically needed.

- **relocateGroup** – *Move a group of nodes (non-destructive reordering).*
  Structure: `{ action: 'relocateGroup', route: [..parentPath], from: startIndex, to: newIndex, groupLength: N, group: [...] }`. The `route` points to the parent containing the group. `from` and `to` are the start indices of the block in old and new order, and `groupLength` is how many sibling nodes were moved as one block. (The `group` might list the moved nodes internally; however, apply uses the live DOM nodes instead of needing their data.)
  Triggers: When diffDOM detects that a contiguous sequence of sibling nodes has relocated within the *same parent* without changes to the nodes themselves. This is diffDOM's way of representing a **move** without deleting and re-inserting nodes. **Important:** diffDOM tries to prefer relocation over separate remove/add when possible (it's a "non-destructive" diff). For example, reordering list items

or moving a paragraph up/down in the same section triggers `relocateGroup`.

Minimal Example: Original: `<ul><li>A</li><li>B</li><li>C</li></ul>`; New: `<ul><li>B</li><li>A</li><li>C</li></ul>`. Here, `<li>A>` and `<li>B>` swapped. diffDOM might output a `relocateGroup` on the `<ul>` with `from:0, to:1, groupLength:1` (move the first item to position 1). If multiple adjacent items moved together, e.g., moving `<li>B><li>C>` as a block, then `groupLength:2`.

Production Example: This occurs in **document reordering operations**. For instance, *Google Docs*-style operations (or *Fidus Writer*) that move whole sections or list entries trigger relocate diffs if using diffDOM. In practice, not all systems use this directly because older diffDOM versions lacked it (some apps historically saw moves as remove+add). But modern diffDOM does handle moves: e.g., *CryptPad* noted needing a patch for moves in older versions. Now, with `relocateGroup`, production diffs can highlight moves distinctly. **Frequency:** moderate – it only occurs if the algorithm confidently matches a moved node by unique identity (like an `id` or unique content). If multiple similar nodes exist, diffDOM might not risk a wrong relocation. When present, this single op replaces what would have been separate removes and adds.

Common Patterns: Many operations above come in **pairs or groups**. For example, adding a new element often includes adding text/attributes inside it (but those are part of the `element` object, not separate ops). Deleting an element similarly removes its content implicitly. A text modification (`modifyTextElement`) might appear alongside attribute changes on the same element. Overall, diffDOM tries to keep diffs concise per node. It will **not** list child removal/adds if a whole parent was removed or added (to avoid redundant sub-diffs). Each diff entry stands for a distinct structural change.

2. Rendering Implications (Dual-Pane View Considerations)

In a dual-pane diff viewer – with a **left pane** showing the "before" DOM and a **right pane** showing the "after" DOM – each diffDOM operation guides how to visualize changes. We need to decide **which pane(s)** to highlight and how to represent the change (using CSS highlights, inserted markers, etc.) while keeping the DOMs independently navigable. Here's how to interpret and render each operation type:

- **addElement:** This means a new element appears in the after DOM. **Visualize on the right pane:** highlight the new element (and its subtree) as an insertion. For example, add a CSS class like `.diff-added` that gives a green background or border to the new element in the right pane. The left pane has no corresponding node (it didn't exist before), so typically nothing is highlighted on the left. In a side-by-side HTML view, you might leave an empty gap or a placeholder comment on the left for alignment, but often it's unnecessary – the new content simply shows up on the right with a highlight. **Implementation:** you can insert an `<ins>` tag around the new content or apply a style to the element node itself (e.g., a green outline). Using semantic `<ins>` (insert) tags can improve accessibility, as screen readers may announce inserted content. However, wrapping the actual node may disrupt layout/scripts, so alternatively adding a non-intrusive wrapper `<span class="diff-added">…</span>` or applying a style via a data-attribute could be safer. **Accessibility:** ensure the added content is still readable – e.g., avoid color-only indication; possibly add `aria-label="added content"` on the wrapper or use the `<ins>` element (which is by default recognized as addition).

- **removeElement:** An element was removed in the new DOM. **Visualize on the left pane:** highlight the element (and subtree) on the left as deleted content. Commonly, this is shown in red or with strikethrough. For example, add a class `.diff-removed` to the node in the left pane and maybe overlay a strike-through or semi-transparent red background. The right pane will not have this element. One approach is to keep a placeholder on the right for alignment (some diff tools insert an empty marker or the word "[deleted]"), but in a dual *structural* view, you can often simply not show it on right. If maintaining alignment is important (e.g., in a table diff), an empty placeholder element with a notation could be inserted on the right side. **Implementation:** using an HTML `<del>` (deleted) tag wrapping the content on the left can convey deletion semantically (screen readers might announce it as deleted) and can be styled (e.g., `<del style="background:#ffe6e6">…</del>`). Alternatively, a CSS class on the element itself can apply styles like `text-decoration: line-through`. Be careful: if you literally remove the node from the left DOM, it won't appear at all – instead, you want it to remain visible in the "before" view but marked as removed. Since the left pane is essentially the original DOM, you keep the node but style it. **Accessibility:** Consider adding an `aria-hidden="true"` on deleted content if you don't want screen readers reading out redundant info, and/or provide an `aria-label` on a deletion indicator like "removed [element type]".

- **replaceElement:** One element replaced another. **Visualization:** this can be thought of as a removal of the old element and an addition of a new element at the same location. In the left pane, highlight the old element as removed (e.g., red outline/strike), and in the right pane highlight the new element as added (green highlight). If possible, you may also visually connect them (since they occupy the same position in the DOM structure). For instance, in a rendered diff UI, you might overlay the new element on the right and show the old on the left, both marked. In HTML code diff view, some tools show a replacement by split-highlighting (but in separate panes, we do it independently). **Implementation:** treat it as `removeElement` + `addElement` in terms of styling. If the elements are of the same type but with many changes, you might instead highlight differences within them, but diffDOM choosing `replaceElement` usually means they're fundamentally different. For accessibility, an `<del>` around the old and `<ins>` around the new conveys a replacement. Make sure to label them appropriately (like `aria-label="removed section"` and `"added section"`).

- **addTextElement:** New text node inserted. **Right pane highlight:** show the new text in an insertion style. For instance, if text "world" was added in "Hello world", you could wrap "world" in a `<ins class="diff-added-text">world</ins>` with underlining or green font. The left pane would just show "Hello " without "world" (no highlight, since nothing was there). If the text was added at the end or beginning, ensure the right pane clearly indicates it (underline or color). **Implementation:** This can often be handled similarly to modifyText, but since it's a wholly new node, using `<ins>` specifically around that text is appropriate. This way, screen readers will identify it as inserted text (and you can style it e.g. italic green). If the added text node is whitespace or line-break sensitive, take care not to alter the exact content when wrapping (you may need to preserve spaces around the ins tag). Also consider the context: if the text node is a separate block (like a standalone text in a paragraph), highlight it as a block insertion.

- **removeTextElement:** Text node removed. **Left pane highlight:** show the text that was removed with a deletion style. For example, wrap the text in `<del class="diff-removed-text">…</del>` with red strike-through. The right pane will simply

not have that text. If the removal causes an empty container in the new DOM, you might show an empty placeholder or just nothing. **Example:** original left: `<p>Hello <del class="removed">world</del></p>` vs right: `<p>Hello </p>`. The left "world" is struck out. **Implementation:** using `<del>` for inline text is semantically correct and can be styled (red, struck-through) [1]. Ensure that punctuation/spaces remain correctly if a word in middle was removed (i.e., keep one space and remove the word). Accessibility: `<del>` will indicate deletion; you might add an `aria-label="deleted text"` if needed.

- **modifyTextElement:** Text changed (some characters replaced). **Visualization:** This is an in-place change, so ideally we highlight the specific differences within the text on both sides. A straightforward but less precise approach is to treat the entire text node as removed and added (like highlight whole old text in left, whole new text in right). But a better approach (used in production systems) is to **refine the text diff**: find the exact changed pieces and mark them. For example, if "worlt" changed to "world", highlight just the differing letter (left pane shows "wor~~l~~t", right pane shows "wor<ins>d</ins>"). diffDOM provides `oldValue` and `newValue`, so you can compute a character-level diff (using an algorithm like **DiffMatchPatch**). Many systems do this for user-friendly diffs – e.g. *Matrix/Element's* diff viewer runs a text diff on `oldValue` vs `newValue` to generate `<ins>` and `<del>` within the text.
  Implementation: Use a diff algorithm to split the text into unchanged, removed, and added spans. Then in the left pane, wrap removed parts in `<del class="diff-removed-text">…</del>`, and in the right pane, wrap added parts in `<ins class="diff-added-text">…</ins>`. Unchanged parts remain plain. If not doing fine-grained highlighting, you could simply show the whole old text with a red highlight and the whole new text with green, but that's less clear for readers. **Accessibility:** using `<ins>`/`<del>` for intratext changes is good – screen readers might read them as "Insertion"/"Deletion" (though not all do by default). To be safe, you could add `aria-hidden="true"` on the visual <del>/<ins> and provide a hidden summary like `aria-label="changed 'worlt' to 'world'"` on the container.

- **addAttribute:** An attribute added in the new DOM. **Right pane highlight:** show the new attribute name/value in a distinct style (e.g., green text or background). In an HTML code view, you might render attributes as part of element tags; you can wrap the whole `name="value"` segment in a `<ins class="diff-added-attr">…</ins>` to indicate it's new. In a rendered view (if showing effects of attributes), you might not have a visible representation (e.g., adding an `id` might not visibly change the content). In that case, consider showing the attribute text in a tooltip or as a faded annotation. But in a DOM diff UI, it's common to display changes in the HTML markup. So, for dual-pane, in the right pane's representation of the element, include the new attribute highlighted. The left pane simply won't list that attribute.
  Example: Left: `<img src="cat.png">`; Right: `<img src="cat.png" <ins class="diff-added-attr">alt="A cat"</ins>>`. Visually, "alt='A cat'" appears highlighted on right.
  Accessibility: If using `<ins>` around attribute text, note this is in the middle of a tag (which is fine in a code-oriented view). Alternatively, use a `<span>` with a special class and perhaps an `aria-label` indicating "added attribute alt with value A cat". If the viewer is not purely text (e.g., a rendered image), you might overlay a small badge "+ alt" on the image on the right.

- **removeAttribute:** An attribute was removed. **Left pane highlight:** display the attribute in the "before" element with a deletion marker. In an HTML text view, wrap the `name="value"` in `<del`

`class="diff-removed-attr">…</del>` with red or strike-through. In a rendered view, if the attribute had visual effects, those will be absent on right side (e.g., a class removal might remove some styling – which itself is visible). We should still explicitly denote it: e.g., show the old attribute faded on the left.
Example: Left: `<td <del class="diff-removed-attr">rowspan="2"</del>>Content</td>`; Right: `<td>Content</td>`. The left shows that `rowspan="2"` was removed (maybe with a line through it).
Implementation: Similar to addAttribute, but on left. Use `<del>` or a styled `<span>` for the attribute text. For attributes that affect rendering (like `disabled`, `checked`), consider also reflecting the effect on the right: e.g., if `disabled` removed, the right pane's form field is now enabled. But the diff highlight itself is the attribute text on left.
Accessibility: Indicate the removal to screen readers – e.g., `<del>` will hint deletion. Possibly add `aria-label="removed attribute rowspan"` on the element or the del span.

- **modifyAttribute:** Attribute value changed. **Both panes highlight:** The element exists on both sides, with the same attribute name, but different value. We should highlight the change in value on each side. E.g., left pane's element shows `attr="old"` highlighted as changed, and right pane shows `attr="new"` highlighted. If the value is a structured string (like a list of classes or a URL), you might highlight just the differing portion. For example, `class="btn **primary**"` -> `class="btn **secondary**"`: ideally, highlight "primary" vs "secondary" rather than the entire class string. This can be done by diffing the attribute strings similarly to text nodes (perhaps splitting on spaces or so). In simpler cases, just highlighting the whole value is acceptable.
Example: Left: `<a href="<del class="diff-removed-attr-value">page1.html</del>">Link</a>`; Right: `<a href="<ins class="diff-added-attr-value">page2.html</ins>">Link</a>`. Here we wrapped only the differing part (or the entire URL) in ins/del within the attribute. In a pure text representation of the tag, it might be easier to wrap the whole `href="page1.html"` on left and `href="page2.html"` on right since they're different.
Implementation: ensure the attribute name itself is present on both, but the value is highlighted. This can be done with something like: `<a href="<del>old</del>">` vs `<a href="<ins>new</ins>">` in the HTML diff display. If showing a rendered view, perhaps overlay a small text indicating the old vs new value (since users can't "see" an attribute change on the rendered element, except through side-effects). For instance, if a class name changed and thus style changed, the right side element will look different; you might still annotate "class changed from X to Y" via a tooltip or icon.
Accessibility: Indicate that an attribute value changed. One approach: add a visually hidden note after the element, e.g., `<span class="visually-hidden">Link destination changed from page1.html to page2.html</span>` for screen reader users, triggered by the diff info. If using ins/del in the middle of the tag text, that's more useful for visual diff.

- **modifyValue (for inputs):** This is effectively a content change of an input/textarea's value. For rendering, it's similar to a text node change, but the text is inside a form field. If you are showing HTML code, treat it like an attribute change of the `value` attribute. E.g., left `<input value="John">` vs right `<input value="Jane">`, highlight "John" vs "Jane" in the attribute. If the original HTML didn't have a `value` attribute (meaning initial was blank), you might pretend it had one for diff display or highlight the property change as text. In a live dual-pane UI, you could even set the left pane input's value to "John" and the right pane's to "Jane", then highlight the input background to indicate a changed field. For example, add a yellow highlight to the input box on both

sides and maybe a small label “changed” on it.
Implementation: The simplest is to inject the old value as a `value="..."` in the left DOM if not already, purely for displaying the diff. Many diff tools for HTML will include form values as attributes in the diff output to make changes explicit. Since diffDOM gives you `oldValue` / `newValue`, you can set the left pane’s DOM input value to the old and the right to new (ensuring those are visible in the UI). Then highlight the difference similarly to text.
Accessibility: If marking input value changes visually, ensure it’s conveyed e.g., an input with changed value might get `aria-label="value changed from X to Y"` or use the `<ins>/<del>` on a label next to it.

- **modifyChecked / modifySelected:** These represent boolean state changes. Visually, one can highlight the control itself. For a checkbox that got checked (old -> new), the right pane should show it checked (because that’s the new state) and one might put a green outline or a subtle “+” icon indicating it’s a newly checked state. The left pane shows it unchecked, possibly with a red outline indicating a removal of the checked state. Another approach: explicitly show the attribute toggling. E.g., in left pane HTML, show `<input type="checkbox" <del class="diff-removed-attr">checked</del>>` (indicating it was checked before and now it’s not), and on the right show `<input type="checkbox" <ins class="diff-added-attr">checked</ins>>` if it became checked. This effectively treats it like an attribute addition/removal for display, which users can understand (checked vs not checked). This is reasonable since `checked` (and `selected`) can be treated as boolean attributes.
  For `modifySelected`, similar logic: highlight which `<option>` became selected and which lost selection. You could add a class to the newly selected option on the right (e.g., background highlight) and to the no-longer-selected option on the left. Or render the `selected` attribute text with ins/del.
  Implementation: If doing a form-element rendering, perhaps overlay a checkmark symbol on the right checkbox to show it’s now checked, and a cross on the left to show it was unchecked (or vice versa). But using the attribute text approach is simpler in an HTML diff context. You might need to inject `selected` or `checked` attributes into the HTML snippet for the side that had it. diffDOM likely does include those in the virtual DOM of the old side if they were set (even if via property, it adds them for diffing purposes). So you can rely on diff to know that, for example, old had `checked=true`. In code view, show it as `checked` attribute in that side’s markup with a deletion or addition marker.
  Accessibility: Changing a checkbox’s state is inherently visible to screen reader users by focusing the field (it will announce checked/unchecked). But to explicitly convey the change, you might add a textual note like “(now checked)” in the right pane visually (perhaps only for screen readers). If using ins/del on the attribute, `<del>checked</del>` vs `<ins>checked</ins>` within the markup is actually a clear way in a textual diff for all users.

- **relocateGroup (Move):** Moves are tricky to visualize. diffDOM’s `relocateGroup` means a block of DOM moved from one position to another without changes. In the left pane, that content appears at its old position; in the right pane, at its new position. To show a move, you might want to mark the content in both places and indicate it's the same content moved. One strategy is to highlight the moved block on both sides with a special color (different from pure add or remove). For example, use a **blue** or orange highlight on the element in the left (where it was) and the same on the right (where it is now), indicating “this item moved”. You could also add an icon or number to link them (some diff tools label moved lines with markers like “↱1” and “↲1” on each side). In a dual pane UI, an arrow or

line could connect the two panes at the location of the moved item, but that's more complex. A simpler approach: mark it as removed-from-old-location and added-to-new-location, but with an understanding that it's a move. This could mean combining the visual styles: e.g., outline the left instance in red and blue (meaning removed and moved) and the right instance in green and blue (added and moved). However, that might confuse users. If move detection is reliable, using a distinct color purely for moved content is clearer.

Implementation: When processing a `relocateGroup` diff, identify the DOM nodes involved. On the left pane, you might leave the moved nodes in their original place but style them as "moved out" (e.g., with a gold border and maybe a little arrow pointing to the right). On the right pane, style the moved nodes as "moved in" (gold border and arrow pointing from left). If the moved group's old position and new position are far apart in the document, a user might not easily connect them, so tooltips could help (hovering could say "Moved from position X to Y"). You might also suppress showing them as deletions/additions to avoid double-counting changes – since they weren't *newly created* or *finally deleted*, just repositioned.

If implementing minimal, you could treat a relocate as a deletion + addition for visuals, but that loses the info that it's the same node. Production systems often struggled here: many simply didn't highlight moves specially, showing them as remove/add (because not all diff algorithms detect moves). Since diffDOM explicitly flags moves, it's good to utilize it. For example, *DaisyDiff*-style tools historically didn't show move arrows, but some modern tools do highlight moves distinctly (like showing the moved text with a different color).

Accessibility: Moves are not naturally conveyed via ins/del. You could announce it in text, e.g., for screen reader or caption: "Section 2 moved after Section 4." Possibly insert a note in the left pane at the old location: "[Moved to new position]" and in the right pane at new location: "[Moved from old position]" as visually hidden text. This way, someone not seeing colors or arrows can understand a move happened.

CSS vs DOM Manipulation: We have choices on *how* to implement these highlights: - Using **CSS classes** on existing DOM nodes: This is minimally invasive. We add classes like `.diff-added`, `.diff-removed` to elements or even text nodes (by wrapping them in span). This preserves the original structure as much as possible, which is good for side-by-side rendering (less risk of altering layout significantly). For text changes, we'd wrap parts of text in `<ins>`/`<del>` or `<span>` with classes. - Using **actual insertion of** `<ins>` **and** `<del>` **elements:** This is semantically meaningful and automatically conveys additions/deletions (browsers often style `<ins>` as underlined and `<del>` as strikethrough by default). It also helps keep added content separate from removed content in markup. However, injecting these tags technically alters the DOM structure slightly (e.g., splitting text nodes). Usually this is fine for read-only diff views. We must avoid breaking block-level structure though (e.g., don't wrap a block-level element with an inline `<ins>` – use CSS for that). It's wise to not wrap entire block elements with ins/del directly in HTML view headings; instead, apply class styles. - **Keeping panes independent:** We apply changes separately to each pane's DOM. For example, to highlight a removal, we manipulate only the left DOM (add a `<del>` or class in it), and for an addition only the right DOM. Both DOMs remain valid and can be scrolled or inspected independently.

Pane responsibilities: The **left pane** should reflect all content that was removed or changed from the original. The **right pane** reflects all new or changed content in the final. So: - Removals: only marked on left. - Additions: only on right. - Modifications: marked on both (old version on left, new on right). - Moves: marked on both (since the content exists in both, just at different locations).

Use of DOM vs text view: If the viewer is showing actual rendered HTML (not just a text diff of code), using visual cues like background color, borders, strikethrough on text, etc., is key. If it's showing the HTML *source* side by side (as code), then literally inserting `<ins>` / `<del>` tags in the code display works (as done in many text-based diff outputs). In either case, ensure a consistent legend for colors: e.g., green = added, red = removed, blue = moved, etc., so users can easily scan.

Accessibility considerations:
- **Color contrast:** Use high-contrast colors or combined cues (like red + line-through, green + underline) so that colorblind users or those in high-glare conditions can distinguish changes. - **Screen readers:** Raw ins/del tags in a text-based view might not be ideal (they may read "insertion" or skip them). If we expect screen reader use, it might be better to provide summary of changes or ensure the ins/del have proper labels. Another tactic: use `role="text"` or aria markers on these diff tags. Alternatively, provide an option to output the diff in a textual summary form (like "Added: 'world'; Removed: 'worl'" etc.). - **Keyboard navigation:** If the diff view is interactive, allow jumping to next change. Each diff could be an anchor target or focusable element with tabindex so users can navigate through differences easily (especially moves, which might need a mechanism to jump from the old location to the new location counterpart). - **Preserving meaning:** For content that has meaning (like list numbering), be careful when highlighting. E.g., if a list item is removed, the numbers on right might shift – the diff highlight should not confuse this. Possibly, number the items via CSS counters so removal doesn't renumber left pane independently. Or explicitly indicate the change in numbering if needed.

In summary, each diffDOM operation guides a specific highlighting strategy in dual-pane: - Highlight **only in the pane(s)** where the change is present. - Use **in-situ markers** (classes or ins/del tags) to emphasize changes without altering other content. - Keep both the "before" and "after" DOM valid and readable, augmenting them with diff indicators as annotations.

3. Operational Sequences

Real-world DOM changes often produce **sequences of diffDOM operations**. Understanding common patterns helps optimize rendering and possibly simplify the diff for users:

- **Combined Inserts or Deletes:** When a user inserts a large block of content, diffDOM typically yields one `addElement` for the container and does *not* list separate adds for each child. For example, pasting a new paragraph with text and styling might come as one `addElement` (with the entire subtree in `element`). Similarly, deleting a section produces one `removeElement` for that section. This means the diff is already consolidated – a single operation covers many internal changes. In a viewer, you might still highlight sub-parts (e.g., text inside the new element) if you want fine detail, but often it's clearer to mark the whole block as added/removed rather than every nested difference. **Sequence pattern:** a lone `addElement` or `removeElement` representing a big addition/deletion, possibly accompanied by a few `modifyAttribute` diffs on the parent (if context around changed).

- **Attribute and Text changes together:** It's common to see a `modifyTextElement` and one or more `modifyAttribute` operations on the *same element*. For instance, editing a link's text and also its URL produces one diff for text, one for `href`. These tend to appear adjacent in the diff list (though order might depend on diff algorithm traversal). Such a group means one element changed in multiple ways. **Pattern:** multiple `modify...` ops with the same `route` prefix (e.g., route to the

element for attribute, route to a child text node for text). In rendering, you might group these in presentation (e.g., show one composite change: "Link text and href changed"), but that's a higher-level interpretation. Production systems (like diff viewers in wikis or CMS) sometimes merge these for clarity. diffDOM itself doesn't merge them (each aspect is separate), but as a developer you can detect when an element had multiple changes and possibly visually group them (like one combined highlight around the element with details on what changed).

- **Sequential text edits:** If multiple consecutive text nodes in the DOM were edited, you'll see multiple `modifyTextElement` operations. E.g., if two adjacent text nodes (maybe separate because of formatting spans) both changed, diffDOM lists each. If those changes are part of one user action (like one sentence split across two nodes), it might be perceived as one logical edit. Production diff tools sometimes **merge adjacent text diffs** for readability. For example, if a sentence is broken into nodes per word (rare, but imagine `<span>Hi</span> <span>there</span>` both change), you might present it as one changed phrase. This is a case-by-case decision. Typically, diffDOM's granularity is fine.

- **Moves combined with modifications:** A tricky sequence is when content is moved *and* changed. diffDOM will usually not treat it as a relocate if the node isn't identical. If a paragraph moved to a new place and also got edited, the algorithm might not detect it as a pure relocate (since the unique descriptors differ due to text change). Instead, you may get a `removeElement` of the old version and an `addElement` of a slightly different new version. In effect, it treats it as deletion+insertion rather than move, because the content changed. **Pattern:** a remove followed by an add that look related (perhaps same tag, similar content). There isn't an explicit diffDOM op linking them as a move in this case. Production systems may want to detect such cases to avoid showing it as completely new content. However, reliably matching a moved-and-modified element is hard – diffDOM errs on side of caution, only relocating exact matches. So, expect sequences like: `removeElement` (with old content) and separately an `addElement` (with new content) if something was moved and edited. If you know in your application that IDs or keys can track identity, you might post-process to label it a move with changes.

- **Sorting/Reordering patterns:** When a list is re-ordered, diffDOM can produce one `relocateGroup` if it's a simple contiguous move, or multiple if things swapped. For example, swapping two items might come out as two relocations (each item moved one position) or one relocation of a block. It often depends on context: if A and B swap, the algorithm might remove A and insert it after B (one relocate) and correspondingly adjust B's position – but B's move might be implicitly handled by A's relocation (since moving A pushes B). In testing, diffDOM tends to output one relocate for the item that moved out-of-order. **Pattern:** Single `relocateGroup` for many reorderings, because often one contiguous block move can account for the new order. If multiple disjoint moves happened, you'll get multiple relocate ops. If the diff shows separate `removeElement` and `addElement` for items in a reorder, it may mean it failed to detect as relocate (maybe items not unique enough). Real apps might then show those as removal/addition (which could confuse as it looks like content was deleted and new content inserted, when actually it was just reordering). If you know the domain (say, list items with identical text moved), you might add your own logic to treat them as moves for the UI.

- **Merges and splits:** Consider a user action like splitting a paragraph into two, or merging two paragraphs into one. How does that manifest?

- *Splitting one node into two:* Suppose one `<p>Paragraph content</p>` is split into `<p>Para</p><p>graph content</p>` (at the break point). diffDOM could interpret this as: a `modifyTextElement` on the first `<p>` (shortening its text), plus an `addElement` for a new `<p>` with the remaining text. Alternatively, it might do `replaceElement` if it doesn't see them as the same tag? But since both are `<p>`, more likely it's modify + add. So the sequence is a modify followed by an add. These two are logically connected (together meaning a split occurred). Similarly, merging two paragraphs into one might yield a `modifyTextElement` (or `addTextElement`) to append text to the first paragraph and a `removeElement` for the second paragraph. The diff won't explicitly say "merged", but the pattern modify+remove (or add+modify) on siblings hints it. In an advanced viewer, you could detect that and label it, but that's optional.
- Another merge scenario: two adjacent text nodes becoming one (or vice versa). This often happens when formatting is added or removed. E.g. "Hello world!" vs "Hello world!" (unformatted). Removing `<b>` could cause the text nodes to merge. diffDOM might output a `removeElement` for `<b>` and a `modifyTextElement` for the now-merged text node. So sequence: remove formatting element, and text changed (actually text concatenated). Recognizing that as a formatting removal might let you present it as such (e.g., highlight text style change instead of raw text difference).

- **Pattern summary:** Splits produce modify + add; merges produce modify + remove, often on related nodes. These often appear together. Performance-wise not an issue, but important for **noise reduction**: if the text content largely remains same but got resegmented, diffDOM can generate what looks like a lot of changes. Real systems sometimes *post-filter* or adjust these. For example, if a formatting tag is removed but text stays same, they might prefer to show "style removed" rather than deletion and readdition of identical text.

- **Styling/formatting changes:** Minor formatting like adding emphasis or changing a tag around text may yield a burst of operations: e.g., wrapping a word in `<em>` could appear as `removeTextElement` (old plain text removed), `addElement` (new `<em>`), and `addTextElement` (the text inside `<em>` reinserted as a new node). Essentially, diffDOM might not identify that as a simple formatting addition, but as replacing text with an element. If the library was smart, it might have done `replaceElement` of text node with an `<em>` element. It's possible diffDOM does do that (text -> element replacement). But if it doesn't, you'll see those 2-3 ops. **Pattern:** removeText + addElement + addText (child) which together represent "text X was wrapped with tag Y". Conversely, unwrapping a tag produces `removeElement` (the tag) + `removeText`? Actually unwrapping might produce a `replaceElement` as well. Keep an eye for these clusters. Some systems might compress or interpret them (like "applied italic to X").

- **Performance considerations – sequence length:** The number of diff operations can explode in certain scenarios:

- Large table row insertion might yield one addElement (with all cells as children) – efficient. But inserting 100 new rows one by one might yield 100 addElement ops, which is fine but perhaps heavy to apply sequentially.
- Worst-case diff: completely different DOMs where nothing matches – diffDOM might output many operations (every node added and every old one removed). For example, replacing an entire document with another of equal size might list removals for all old nodes and additions for all new nodes. That could be hundreds of ops. In such cases, some systems choose not to display a fine-

grained diff at all, instead maybe saying “document replaced” or highlighting the top-level change to avoid overwhelming the user.

- **Expensive sequences:** Many sequential text modifications (like diffDOM splitting a long text into many diffs) could be slow to render with fancy highlighting. If diffDOM’s `textDiff` option is not customized, by default it replaces the whole text, which results in one op per changed text node, which is fine. But if one tries to highlight each character difference by wrapping spans, that can be heavy for very long text changes. Production tip: limit rendering of extremely large text diffs (maybe show a summary or allow toggling detailed view).
- Another performance angle: **applying diffs**. diffDOM can apply a list of operations to a DOM in one go. In our case, we aren’t applying them to the actual live DOM of the app, but rather using them to drive UI. The cost is mostly in manipulating the DOM for highlighting. Doing that naively for each diff (like finding nodes by route each time) can be slow if there are many. Real systems often optimize by resolving all routes one time in one tree-traversal, or mapping node identities if possible (e.g., know which element corresponds to which route from an initial traversal). Also, if you have extremely large DOMs (thousands of nodes), querying by deep route for each op could be heavy. One could improve by storing references to nodes in a dictionary keyed by something (though route is dynamic if other diffs mutate the tree).

- **Batching updates:** If updating the actual DOM, it’s faster to apply all changes in one fragment or with minimal reflows. For highlighting, adding classes might be okay, but adding wrapper elements repeatedly can cause many reflows. Possibly accumulate the changes and apply them together or use `DocumentFragment` for building the diff-marked DOM.

- **Conflict or canceling patterns:** diffDOM’s output generally reflects the net differences only, so you shouldn’t see outright contradictory ops (like an element added and removed with no net change – diffDOM would normally omit such a no-op). However, if the diff list is generated from sequential patches, one might accidentally feed in overlapping changes. For example, if someone tried to merge two diffs (like patch on patch), operations could conflict. One user in diffDOM issues attempted to apply a diff to a DOM that already diverged in other ways, leading to failures. This is outside normal static diff usage but worth noting: each diff assumes it’s applied to the original state. If you try to apply them out of order or to a slightly different base, some ops might not find the node (thus diffDOM’s `apply` would return false). In a dual-pane viewer, this isn’t an issue since we’re not applying them to mutate, just highlighting.

- **Compression and filtering of sequences:** Some production systems introduce a **post-processing step** to simplify diff sequences:

- **Merging adjacent operations:** e.g., if multiple text diffs happen in a row within the same parent, maybe combine them into one if it makes sense (particularly for display, if they’re part of one sentence).
- **Filtering trivial ops:** e.g., ignoring changes in insignificant whitespace or empty text nodes. diffDOM might diff every text node, including purely whitespace differences (like one side has a line break or space and the other doesn’t). These yield modifyText or add/remove text ops that might be considered noise. A system could detect if a `modifyTextElement` is only a whitespace change and decide not to show it (or show it only in a “whitespace visible” mode). Similarly, reordering of attributes (if diffDOM did that, though usually it doesn’t care about order of attributes, just values) could be ignored.

- In diffDOM, the `filterOuterDiff` option allows skipping certain diffs entirely based on custom logic. For example, as in issue #81, treating nodes with a certain attribute as identical to skip their internals. In sequence terms, this can collapse potentially many ops (it basically prunes diffing inside those nodes). In production, if you have sections of DOM that are known to be generated or irrelevant, you can filter them to reduce diff noise.

Sequences and performance: In general, diffDOM tries to produce minimal necessary operations, but the count can still be large. The `maxChildCount` and `maxDepth` options in diffDOM can limit diff depth to cap the number of ops. E.g., `maxChildCount: 50` ensures it won't generate diffs more than number of children (it might give up fine-diffing a very large list and just treat it as replaced if too many changes). In practice, performance issues may arise around 1000+ operations; below that, modern browsers handle highlighting that many differences fine, especially if just adding spans.

Expensive sequences to watch: - **Highly repetitive changes:** e.g., toggling a class on 100 list items (diffDOM will output 100 modifyAttribute ops). Rendering 100 highlights is okay, but if each item is large, the UI might become cluttered. Possibly group them (e.g., "changed class on all list items" message) if that pattern arises frequently. - **Deeply nested changes:** If a deep subtree is entirely new, diffDOM may add it as one operation – efficient. If a deep subtree is *slightly* changed in many places, diffDOM will produce many small ops scattered. Navigating those might be hard for a user. Sometimes a user prefers to see "section X heavily changed" rather than 50 tiny diffs inside it. Some systems provide a summary or collapsible diff regions to address this (e.g., hide fine diffs under a toggler).

In summary, common diffDOM sequences mirror user edits: - Rewritten paragraph: yields a handful of `modifyTextElement` diffs (or one if not segmented). - Content reordering: yields `relocateGroup` or an equivalent remove/add pair. - Added section: one addElement (big subtree). - Removed section: one removeElement. - Style changes: an addAttribute/removeAttribute or small replaceElement cluster. Understanding these helps to possibly **simplify the presentation**: we can decide to merge, group, or label combined operations (like grouping a relocate's outcome or multiple modifies in one element) to make the diff easier to comprehend. The goal is to present the *sequence of changes* as something meaningful: e.g., "Moved section 3 above section 2" or "Added a new paragraph and updated the preceding heading's text."

4. Implementation Lessons from Production

Examining real applications using diffDOM (and similar DOM diff tools) reveals several lessons for handling diff operations in production:

- **Filtering and Noise Reduction:** Production systems often implement filtering to avoid overwhelming users with trivial changes. For instance, *Fidus Writer's* collaborative editor might ignore diffs in certain metadata elements or always skip diffs for auto-generated timestamps. diffDOM provides hooks like `preDiffApply` and `filterOuterDiff` for this purpose. A real example: a developer set `filterOuterDiff` to treat nodes with `data-summary` attribute as equal if the summary is same, thereby **skipping inner diffs**. This prevented a flurry of changes inside a frequently updating element (like a dynamic word count) from showing up. Lesson: Identify sections of the DOM or attributes that don't need user-facing diffs (e.g., hidden elements, non-significant whitespace, ad banners in a saved page diff, etc.) and filter them out or coalesce them.

- **Consolidating Redundant Ops:** While diffDOM tries to avoid redundant operations, in practice some sequences can be merged for clarity. One production approach is *post-processor consolidation*. For example, if diffDOM gave: addElement (empty container) followed by addTextElement (text inside it), one can merge that into a single addElement with text in it (which is essentially reconstituting the fact that a whole element with text was added). Typically, diffDOM would have already included text in the element object if it was part of initial insertion, but if something caused separate ops (maybe if the text node diff considered separately), merging them simplifies the output. Another consolidation: consecutive text mods could be merged as one edit. The *GADIE refactoring* (as hinted in the prompt) likely involves consolidating diffDOM output – possibly an internal name for a system that groups diff operations (like GADIE might stand for some diff engine or UI component).

- **Performance optimizations:** Real-world usage of diffDOM on large documents (100+ KB HTML) has uncovered performance issues. Projects have addressed this by:

- Limiting diff depth or complexity (using diffDOM options `maxDepth`, `maxChildCount` to prevent extremely slow comparisons in huge DOMs).
- Avoiding diffing large identical sections by hashing or early checks (not a built-in diffDOM feature, but an app can short-circuit diff if two sections have the same hash, skip diffing inside).
- In collaborative editing (like *Operational Transform or CRDT* systems), diffDOM might be too slow for real-time on very large docs, so they limit its usage to smaller sections or do batched updates.
- If diffDOM is used in a server or background context (e.g., comparing two versions of a web page), systems like *web-monitoring-diff* have options to choose a simpler diff if content is too large or treat it as mostly-changed.

- On the rendering side, if you have thousands of diff markers to insert, update the DOM efficiently: e.g., using a single tree walk to apply all highlights. A practical tip is to use the `route` info: you can traverse the original DOM tree once, and whenever your traversal path matches the next diff's route, apply it, rather than doing separate `getElementByPath` for each diff. Alternatively, leverage diffDOM's own apply on a clone of the DOM with custom handlers (like instead of actually applying, intercept each action to decorate the clone's nodes with markup).

- **Handling Large Documents:** Production diff viewers often allow **collapsing sections** of the diff. For example, if a large section has many small changes, an option to collapse that section with a summary "12 changes hidden. Click to expand." helps usability. This isn't directly provided by diffDOM, but you can implement it by inspecting diff routes: if many diffs share a common ancestor, maybe give an option to collapse that ancestor's diff. For instance, if 50 diffs all occur inside a `<div id="section5">…</div>`, you might show a one-line summary for section5 and let the user expand to see details. This was inspired by how some code diff tools collapse unchanged lines – here we collapse changed sections if too verbose.

- **Real-world catches (bugs encountered):** A lesson from the Matrix/Element integration: initially, they might not have expected diffDOM to produce `relocateGroup` (especially if using an older version). If using a diffDOM version without move support, they saw remove+add pairs instead. Once diffDOM introduced moves, the rendering code needed to handle a new `action`. In one case, a developer noted *"diffDOM's algorithm is mostly correct, but it lacks a move operation…"* – they patched it. Now that `relocateGroup` exists, ensure your code doesn't ignore it. Similarly, some GitHub issues point out edge cases:

- diffDOM sometimes treated nodes with the same content but in different places as moved when they weren't logically the same node (if content was identical and unique, it could mis-match). The use of `uniqueDescriptors` is meant to minimize false moves by requiring a unique signature. In production, if you do encounter a spurious relocate (like it thought a paragraph moved, but really you deleted one paragraph and added a different one that happens to have similar text), you might want to double-check. Possibly an app could compare element IDs or other metadata to confirm a move or override diffDOM's decision if it seems off.

- Ordering of attributes in diff output: diffDOM doesn't consider attribute order significant (per HTML rules), so it doesn't issue diffs for reordering attributes. But if you diff XML or XML-like HTML where order might matter, that could be a catch – diffDOM might ignore a meaningful order change. Most apps don't consider that a problem, just a note.

- **Applying diffDOM in collaborative settings:** Tools like *CryptPad* and *Matrix* have used diffDOM diffs to send changes over network (since it can serialize to JSON easily). One lesson: diffDOM diffs must be applied to the exact same initial DOM state they were generated from. If not, `apply` will fail or produce wrong results. So production systems include checks or versioning. For example, Matrix's editor might diff an original message to edited message and then try to apply that diff to the HTML currently rendered. If the message content changed by something else in the meantime (like an auto-format), the apply could fail (diffDOM returns `false` on conflict [2]). So one should have a fallback: either re-generate a diff with the latest base or just replace content wholesale if patch fails.

- **Preprocessing content for better diffs:** Some production pipelines **normalize HTML** before diffing to reduce noise:

- Removing insignificant whitespace text nodes (or conversely, adding a consistent whitespace so diffs don't catch random differences in indentation).
- Sorting attributes or stripping expected random attributes (like transient `data-reactid` attributes in React-rendered HTML) so they don't appear as diffs.
- In *wiki* or documentation diffs, sometimes they remove scripts or style sections that aren't relevant to user content changes.

- Ensuring both DOMs are comparable (e.g., diffDOM might treat a self-closing tag vs explicit closing tag differently if the DOM parser outputs differently – usually not, but by feeding actual DOM nodes via `nodeToObj` you avoid that).

- **Integration with text diff for readability:** Many production systems end up combining DOM diff with text diff. For instance, *Element (Matrix)* uses diffDOM to identify changed nodes, but then uses a text diff library on those nodes' text content to generate a nicer display of changes. The lesson is that diffDOM gives a structural diff, but human-friendly output often requires a second pass for fine details (especially in text and attributes). Another example is *visual-dom-diff* by Teamwork, which effectively does its own diff and then wraps changes in `<ins>/<del>` tags for display. They chose to highlight at text-level granularity. This suggests that using diffDOM's output directly is great for applying changes or programmatic patching, but for *visual diff* you might enhance it with string diffs for the content changes.

- **Undo/redo of diffs:** diffDOM supports `undo` by generating inverse diffs. Some apps have leveraged this for collaborative editing undo stacks. If you apply a diff and store it, you can undo it with `dd.undo()`. The takeaway: the diff operations are reversible. In a viewer context, this is not directly needed, but it assures that representing changes as these operations is "lossless". For an editor, you could apply diff ops for an incoming change and undo if needed. One must ensure not to mix up diffs out of order though.

- **Dealing with moves in production:** As mentioned, older diffDOM had no explicit move op, so apps either wrote their own detection or accepted remove/add. Now with `relocateGroup`, one lesson is to present moves meaningfully. Some UIs (e.g., *track changes* in Word or Google Docs) do not explicitly show moves – Word, for instance, just shows it as deletion in old spot and insertion in new spot, possibly with comments that content moved. But newer tools (like GitHub's rich text diff) highlight moved lines. It's still a UX question whether to emphasize moves or treat them as a combination of deletion/addition. At least diffDOM gives the info; how to use it might depend on user preference. If moves are frequent (like reordering sections of a document), giving them a special treatment helps users follow reordering without reading content and guessing.

- **Case study – Fidus Writer:** As diffDOM's origin, Fidus Writer likely employs it for collaborative editing. They would have encountered issues like diff size in large docs and ensuring diffs apply correctly in real-time. Possibly they implemented chunking of diffs or queuing changes. A likely lesson: if one user is making a lot of rapid changes, diffDOM on each keystroke might be too slow for real-time. Throttling diff generation (e.g., generate diffs at intervals or batch multiple small changes) can be necessary. Also, merging consecutive diffs to send over network (like bundling changes) might be considered, though diffDOM is quick for small changes, large ones needed caution.

- **Stack Overflow insights:** Community discussions often revolve around handling diffDOM's quirks. For example, one question might be how to handle a scenario where diffDOM treats a node replacement unexpectedly. The consensus was often to inspect the diff and adjust. E.g., if diffDOM output seems overly granular or weird, sometimes adjusting the diffDOM options (like disabling valueDiffing if not needed, or enabling `innerHTML` diffing mode for certain nodes) can yield more expected results.

- **Comparison with other tools:** Projects that tried alternatives (like *DaisyDiff* or custom DOM diffs) found diffDOM mostly robust, but lacking in documentation for the diff format. So a production guide (like this one) is valuable. We saw an architecture doc comment: *"Its diffing algorithm is mostly correct, but it lacks a move operation..."* – which has since been addressed. Also, others have noted diffDOM can sometimes produce large diffs, hence interest in compressing diffs (the `compress` option turns the keys into numbers to shrink JSON size). If sending diffs over slow networks (e.g., collaboration in low-bandwidth), using `options.compress = true` might be a good idea to reduce payload.

In essence, production use of diffDOM teaches us to:

- **Fine-tune what changes to show** (filter trivial ones, emphasize meaningful ones).
- **Optimize diff creation and application** for large content (limit scope, batch changes).
- **Augment structural diffs with content diffs** for best visual clarity.
- **Be prepared for edge cases** (like moves, or diffs failing to apply due to mismatches).

- **Use diffDOM's features** (hooks, options) to integrate smoothly (for example, use `preDiffApply` to skip an operation if you decide it's not user-relevant – diffDOM allows returning true in `preDiffApply` to short-circuit an op).

By applying these lessons, the dual-pane diff viewer can be both efficient and user-friendly, focusing on the changes that matter and handling big documents or tricky operations gracefully.

5. Gotchas and Edge Cases

Even with a robust diff algorithm, there are several **gotchas and edge cases** to be aware of when using diffDOM operations:

- **Misidentified Moves:** While diffDOM strives to detect relocations, it isn't foolproof. It relies on unique signatures (like tag + id/class + content). If two identical elements exist and one gets removed while another similar one is added elsewhere, diffDOM **might incorrectly flag a relocate** (thinking the element moved, when actually one was deleted and a different one inserted). For example, if you have two identical paragraphs A and B, and you delete A and add a new paragraph C elsewhere with text identical to A, diffDOM could see it as A moved to C's position. This is rare (because if they're identical, they're not unique, so the algorithm likely won't treat them as the same unique element – it would then not do a relocate to avoid ambiguity). But it's a scenario to test. The gotcha is mainly: not every remove+add of similar content is labeled as relocate, and if it is labeled, double-check it's truly the same logical item. **Workaround:** If your content has stable IDs, consider including them (diffDOM will treat elements with same id as likely the same node, which can help correct matching in moves).

- **Route Interpretation:** The `route` arrays in diffDOM can be confusing at first. Remember that the route is relative to the **root element you diffed** (often the body or a specific container). A route like `[3, 2]` means: go to the 4th child of root, then the 3rd child of that element. Off-by-one misunderstandings are common. Also, **after applying some diffs, routes of subsequent diffs could shift** if you were applying sequentially to a live DOM. For instance, if you remove child 0 of a parent, then an `addElement` later referring to route `[parent, 5]` now might actually be index 4 in the updated DOM (because one was removed). However, diffDOM's generated diffs are in an order that is meant to be applied sequentially to maintain validity. It adjusts insertion routes by computing them on the fly during apply for adds. If you apply out-of-order or try to interpret all routes on the original DOM, you might get mismatches. **Lesson:** Use diffDOM's apply order or update your reference structure as you interpret each diff. In read-only visualization (where we don't actually remove nodes from the original), you should interpret routes against the original DOM for removals, but for additions you need to be careful since the route includes the position *in the new DOM*. diffDOM itself handles this by a special case for add routes. Our viewer can sidestep this by not actually removing nodes from the left DOM at all – thus, for highlighting, we may compute positions differently. It's an edge case if one tries to auto-align content strictly by numeric indices. So be cautious with using `route` for mapping nodes in a static diff view; sometimes searching by structural position or id may be safer.

- **Text Node Merge/Split Oddities:** The DOM may split text into multiple nodes unexpectedly (especially if there are entity references or if innerHTML was used differently). If one side's parsing yields one text node and the other yields two (e.g., `<p>Hello  world</p>` might parse

with two text nodes around ` ` in some cases), diffDOM could output an addText/removeText pair where logically nothing "changed" except how whitespace was tokenized. This is a corner case often due to parsing differences. In production, one might normalize the DOM (e.g., ensure no adjacent text nodes by merging them) before diffing to avoid such false diffs. Similarly, a slight difference in whitespace can trigger a modifyText that may seem inconsequential (like one extra space or newline). These appear as diffs even though content might render the same. **Solution:** possibly trim or collapse whitespace when appropriate, or at least detect if `oldValue` and `newValue` are same when trimmed (to ignore if only trailing spaces differ, if that's not meaningful for the application).

- **Empty Text Nodes and Insignificant Nodes:** DiffDOM will treat an empty text node (`""`) as a node. Sometimes HTML contains empty text nodes or purely whitespace nodes for formatting/indentation. If one DOM has them and the other doesn't, you'll get addText or removeText diffs that literally contain `value: " "` or `""`. These are usually noise. For instance, pretty-printed HTML vs minified HTML differ in whitespace text nodes. Real systems often **ignore diffs that are solely empty or whitespace text** unless whitespace is meaningful (in preformatted text). So it's a gotcha to either configure diffDOM (no direct option for that aside from maybe filtering in post) or handle it in the UI (don't display a diff that says "removed `\n`"). Some diff viewers hide whitespace changes by default with an option to show them.

- **Attribute Ordering and Case:** Attributes in HTML are case-insensitive (names), but in XML they might be case-sensitive, and order doesn't matter. diffDOM likely ignores order altogether. But if you have two DOMs where attributes appear in different order, diffDOM will output no diff (because logically the element is the same). This is correct for HTML. Just a note in case someone expects a diff – it's not a bug, just how HTML works. Another attribute quirk: boolean attributes like `checked` or `disabled` might appear in one as just `checked=""` and in the other as `checked="checked"` or just `checked` with no value. diffDOM typically normalizes those to presence/absence rather than string differences. But if it doesn't, you might see a modifyAttribute from `checked="checked"` to `checked=""` which is essentially no real change (just how the HTML was serialized). Usually diffDOM uses `.removeAttribute()` or `.setAttribute()` which doesn't care about the specific value for boolean (any value means true). This is minor, but if you see an attribute diff that toggles between "" and the attribute name, it might be an artifact of how the HTML was generated. Possibly filter it out if both represent the same truthy value.

- **Similar Content Treated as Change vs Move:** There's a threshold in the algorithm: how far it searches for a possible move. If an element moves very far or to a very different context (e.g., moved from one parent to a completely different parent), diffDOM might not detect it as relocate. It tends to only consider moves within the same containing parent (relocateGroup is designed for reordering siblings). If a node was cut from section A and pasted into section B, that is conceptually a move, but diffDOM will see a remove in A and add in B – because relocateGroup doesn't cover cross-parent moves. **Edge case:** moving content across the DOM hierarchy is shown as deletion + insertion. Our viewer should thus handle those as separate ops (and perhaps we could optionally hint "moved to a different section" if we can deduce by content matching, but diffDOM itself won't label it a move). So not all real moves are caught by diffDOM – only sibling reorders. This is important: if users drag an element from one container to another, you will see it as removed and added, not as a single "move" op. Don't be confused by that – it's by design. The gotcha is simply understanding relocateGroup's scope.

- **modifyValue vs modifyAttribute (value):** This can be a subtle edge: If an `<input>` had a `value` attribute originally and the value changed, sometimes diffDOM might produce `modifyAttribute` for the `value` attribute *and* a `modifyValue`, or one of them. Actually, the diffDOM code has a special case: when setting an input's value via attribute diff, it also updates `.value` property. They differentiate between initial attributes and user input. For example, if the HTML changed the `value` attribute, you get `modifyAttribute(name="value")`. If the HTML didn't change but the user input did, you get `modifyValue`. But consider a case: original `<input value="X">`, new `<input value="Y">` (user changed it and it's saved in HTML) – diffDOM might just do modifyAttribute for value. If original had no value attr and new has one (user input saved), that could be addAttribute. Meanwhile, if a user change isn't reflected in attribute, it's modifyValue. So, sometimes the "same" change (the text in a field changed) could be represented differently depending on how the DOM was captured. This is an edge nuance: be prepared that changes to form fields might come as either attribute diffs or value diffs or both. The viewer can treat them similarly. But make sure not to double-count. If diffDOM ever gave both (perhaps if one side had an attribute and other only property?), that would be weird but just in case: filter out duplicate representation. Usually it will be one or the other.

- **Route length changes due to concurrent inserts/removes:** If multiple siblings are inserted or removed at a location, later operations' routes might refer to positions in the new DOM structure. For example, if diff contains: removeElement at [5] then another removeElement at [6], those are in original indexing. Once you remove index 5, what was originally index 6 becomes index 5. But diffDOM likely computed them on original independently, and when applying sequentially it adjusts because after the first removal, it will call getFromRoute and find the next. If you were to apply out-of-order or not sequentially update an index mapping, the second remove might end up removing the wrong element. This is only if you misuse the diff (like applying in wrong order). It's a gotcha in writing a custom applier or viewer alignment: always process in given order. If our viewer does all highlighting after computing all diffs, we shouldn't actually remove nodes from the left DOM at all (so their indexes remain original). That's fine for display. But if we try to simulate the DOM changes in the viewer's DOM to line things up (like physically removing nodes from left pane to collapse space), we'd have to update subsequent route references. Safer not to do that. Instead, leave the DOM intact and just mark removals visually.

- **Diff Application Failures:** If you use `dd.apply(fromDOM, diff)` and it returns false, something went wrong (element not found at route, etc.). In a pure viewing scenario we're not applying to mutate, but if someone tries to test the diff by applying it to ensure it works, failures often mean either the DOMs weren't truly in sync or a bug. One known case is diffDOM sometimes returning false when trying to apply a diff with relocation if the node moved doesn't exist at expected place. Could be due to prior operations messing it. Always ensure you apply on the exact original DOM state. It's more a developer gotcha than user-facing.

- **Edge cases where diffDOM output "seems wrong":** A few have been reported:

- In some older versions, diffDOM had trouble with certain self-closing tags or comment placements (e.g., a comment node at end of parent might have led to odd route calcs). These are mostly fixed, but keep an eye on comments at edges.
- If the from and to DOM have different namespace contexts (like one part is SVG), diffDOM handles namespaces but ensure you treat created elements properly if applying. For viewing, if you highlight

an SVG element, adding spans or ins tags might not be valid in that XML namespace. So highlight via CSS rather than adding foreign elements inside an SVG node. That's a viewer gotcha: don't insert HTML tags into an SVG DOM, or it'll break. Instead, maybe highlight the SVG element by overlaying an HTML element or using a filter effect.

- If diffing tables, sometimes moving rows around can be tricky because browsers might insert `<tbody>` implicitly. If one HTML had no explicit tbody and the other does, diffDOM might think a structure changed (like an extra container added). Typically, diffDOM normalizes that by using DOM parsing (browsers always produce a tbody). But if one side was parsed differently, you could get an addElement for `<tbody>` when comparing innerHTML strings. Best to feed actual DOM nodes to diffDOM (which handle those consistently).

- **Understanding `group` in relocateGroup:** The diff contains a `group` property (likely an array of virtual nodes that were moved). In our earlier analysis, we noted apply doesn't really need it (it physically removes and reinserts nodes). But `group` exists for completeness (and for JSON serialization – so the diff can be standalone). If you inspect a relocate diff, you'll see something like `{ action: "relocateGroup", route: [2], from: 5, to: 2, groupLength: 1, group: [ {node representation} ] }`. The `group` content might be the moved node's data. This can be used if you wanted to display exactly what moved (though you could also locate the node via route). Just know it's there. If you inadvertently apply a relocate diff as separate remove/add operations (not using diffDOM's relocate logic), you might double-remove the nodes. So handle it as one atomic thing if you manually apply.

- **Diff size and memory:** If two DOMs are vastly different, the diff array might be huge (in extreme cases, size ~ number of nodes). This can consume memory. If you attempt to JSON.stringify a very large diff, that string might be big. diffDOM's `compress` option helps by using numbers for keys and not repeating strings for actions. If we were sending this diff to a client or storing it, it's useful. For our viewer, since we directly highlight, we might not need to store the whole diff as JSON, but if we do (say for debugging), be mindful of size.

- **Algorithmic heuristics:** diffDOM may treat certain changes as modifications vs replacements depending on content. If an element's tag changed but children are identical, diffDOM does `replaceElement`. If children changed a lot too, it might still do replace vs building a list of diffs inside – it has some heuristics. If you find a case where you expected more granular diffs but got a replaceElement, it might be because diffDOM decided it's too different (or tag name changed). Conversely, sometimes it might do many diffs where a human would think "just replace the whole thing". This is subjective. Recognize that diffDOM's output is one possible diff, not always the only way. For instance, if an entire subtree was replaced with completely new content of roughly same size, diffDOM might still try to match some bits and produce a series of modifications, which could be more confusing than a simple "replace parent node". There's an open question of whether to collapse such diffs. A heuristic: if an element has more than X% of children changed, maybe showing it as wholly replaced is easier. But diffDOM doesn't do that by default (except via maxChildCount perhaps). So a gotcha is "over-detailed diff". The fix in presentation might be to detect and group as mentioned.

- **Out-of-Order Diff Application in UI:** If building an interactive viewer where a user can toggle applying diffs one by one (like stepping through changes), applying them in real DOM sequentially

will actually mutate the left DOM toward the right. That's fine, but then your left pane is no longer original state midway – it's partially updated. That can confuse the highlighting logic if not careful. If implementing a step-through view, better to not literally apply changes to the actual displayed DOM nodes, but rather simulate by revealing/hiding prepared highlighted elements. Alternatively, re-diff on each step (too slow). Usually diff viewers just show all diffs at once or allow jumping, not incrementally applying them visually. So perhaps not a big issue, but something to consider if you ever wanted to animate the diff application.

In conclusion, handling diffDOM in production requires guarding against subtle mismatches and making decisions on what changes matter. Many of these gotchas can be mitigated by:

- Preprocessing the DOMs (normalize whitespace, ensure consistent parsing).
- Postprocessing the diff (filter noise like whitespace-only changes, merge logically connected ops).
- Thoroughly testing typical content so you know how diffDOM expresses those changes.
- Being prepared to adjust if diffDOM's output changes with new versions (e.g., a future diffDOM update might introduce new action types or tweak move detection).

By anticipating these edge cases, the diff viewer will be more robust. When something "looks wrong" in the diff, it often traces back to one of the above issues – understanding them allows quick fixes (like ignoring a weird diff or combining it with another). And because diffDOM's output is designed to be easily interpretable, we have the flexibility to refine the presentation without losing fidelity to the underlying changes.

---

1 jquery - Generating a diff of two DOM elements (rather than two strings) in javascript - Stack Overflow
https://stackoverflow.com/questions/9468979/generating-a-diff-of-two-dom-elements-rather-than-two-strings-in-javascript

2 GitHub - fiduswriter/diffDOM: A diff for DOM elements, as client-side JavaScript code. Gets all modifications, insertions and removals between two DOM fragments.
https://github.com/fiduswriter/diffDOM