ChatGPT

Hover-Based UI in Production-Grade Editors

Overview of Hover/Selection UI Challenges

Modern rich text editors (Notion, Google Docs, Medium, Slate, Quill, TipTap/ProseMirror, etc.) often display contextual UI elements (icons, toolbars, tooltips) when you **hover over annotated text or select text**. Implementing these in an **active edit mode** (when the contenteditable is focused) is tricky. The UI must appear at the right position and time without disrupting the **caret (text cursor)** or the user's typing. Key challenges include: ensuring the overlay doesn't capture focus or block text selection, handling cross-browser quirks (especially in Safari), debouncing the appearance (e.g. slight delay), and positioning the element (often above the text line). Below, we discuss common implementation patterns and how real-world editors solve these problems.

Approaches: Overlays vs. Inline Decorations

1. DOM Overlay Elements: Many editors render hover toolbars or icons as separate DOM elements absolutely positioned *above* or *beside* the text content. These overlays are often appended outside the contenteditable (for example, as a sibling to the editor node or even to the document body) so they do not alter document flow [1]. By using methods like `getBoundingClientRect()` or editor-specific APIs (e.g. ProseMirror's `view.coordsAtPos`), the overlay can be positioned to align with a specific text position or selection [2]. This approach is used for floating "bubble" menus (à la Medium) and hover tooltips. It prevents the UI element from becoming part of the editable text, thus avoiding any impact on typing or layout.

2. Inline Widget Decorations: Another approach (especially in ProseMirror) is to use **widget decorations** – essentially ephemeral inline DOM nodes – to mark a position in the document and inject an element there. These widget elements can be styled as absolute/fixed overlays. If no explicit CSS `left/top` is given, the widget will sit inline at its text position [3]. This is a simple way to attach an icon or label at a specific spot in the text. However, widget decorations have limits: if you need the tooltip to float *above* selected text or overflow outside the editor (e.g. into a margin or beyond a scroll container), a purely inline approach won't suffice [4]. In such cases, manual positioning with an external element is needed.

Most production editors use a hybrid: simple cases (like an icon at the start of a hovered block) might use a widget decoration, whereas complex popovers (like formatting toolbars) use overlays anchored via coordinates. For example, ProseMirror's documentation notes *"the easiest [tooltip] is to insert widget decorations and absolutely position them... [but] if you want to position something above the selection... then decorations are probably not practical"* [5].

Plugin-Based Event Handling (ProseMirror/TipTap)

In editors based on ProseMirror (including TipTap), the robust way to track hover state in edit mode is to use a **plugin with event hooks**. ProseMirror's plugin system allows intercepting DOM events via the `handleDOMEvents` property. For instance, one can capture `mouseover`/`mousemove` events on the editor and determine which node or text span is under the cursor. Marijn Haverbeke (ProseMirror's author) suggests using `handleDOMEvents` (listening for `"mouseenter"` or `"mouseover"`) combined with `view.posAtDOM(event.target)` or `view.posAtCoords(...)` to locate which document position or node is being hovered [6]. This avoids modifying the actual selection – *"you don't want to select something just by hovering… Maybe create a state field that tracks the paragraph that's being hovered over."* [7].

Stateful hover tracking: The plugin can maintain a piece of state (a *plugin state field*) to record the currently hovered node/position. On each mouse move, if the hovered node changes, the plugin can update its state and trigger a UI update (e.g. show or move the hover icon). This way, hover state is part of the editor's state updates, keeping it in sync with the view. One developer reported success with this method: on `mouseover`, dispatch a transaction with metadata of the hovered position, and use a plugin's `state.apply` to update a field and redraw a custom UI component [8].

TipTap example: TipTap (built on ProseMirror) encourages this approach via extensions. For example, a TipTap user implemented Notion-style node hover buttons (a plus and menu in the left gutter) by adding a ProseMirror plugin in an extension. In the plugin's `props.handleDOMEvents.mouseover`, they compute the hovered block's DOM rect and then position an external `tools` `<div>` next to it [9] [10]. This overlay is appended to the editor's container (`element.appendChild(tools)`) and shown when hovering a block. (In that proof-of-concept, they simply displayed a "TOOLS" label; a real implementation would style a button icon.) Another responder noted one could also use a decoration widget inside the node to show extra controls [11] – but the event-driven overlay is often more flexible.

ProseMirror's own **Selection Tooltip** example uses a plugin with a *view* component: it creates a tooltip element in the plugin's `view()` constructor and appends it as a sibling to the editor content [1]. Then in the plugin's `update()` lifecycle, it checks the editor state and repositions that tooltip. In the example, when text is selected, the tooltip is shown above the selection and displays the character count [12] [13]. This demonstrates the plugin approach for a selection-based bubble. The tooltip's position is updated using `coordsAtPos` for the selection endpoints, computing a horizontal center and a vertical offset just above the selected text line [2]. The technique ensures the tooltip moves with content changes and hides when the selection is empty [14].

Advantages: Using an editor plugin or extension to manage hover UI means the logic is tightly integrated with editor state. It can respect editor focus and won't miss events (which might happen if you rely on a separate document-level listener that gets disconnected when the editor is focused). Indeed, internal research notes for a TipTap project observed that in edit mode the editor might be "consuming mousemove events" or not bubbling them [15]. The recommended fix was precisely to handle events inside the editor (via a plugin or capturing listener) rather than only at the document level [16]. By hooking into ProseMirror's event system, you ensure your hover logic runs even when the editor is focused.

Contextual UI in Various Editors

Different editors converge on similar patterns for these UIs:

- **Notion:** Notion's editor (which is block-based) shows a hover menu for each block – a grab handle (six dots) and a "+" button for new blocks – when you move the mouse near the block. This is implemented by overlaying controls in the left margin of the content. Likely, each block element in the DOM has an associated hidden control that becomes visible on hover (via CSS `:hover` or via a dynamic event that inserts the control). The key is that the control is absolutely positioned and does not push the text. Notion ensures you can still place the caret in the text freely; the hover icon appears *outside* the text area. (Notion also shows inline comment icons: when text has a comment, hovering it shows a small speech-bubble icon at the end – again an absolutely positioned icon that appears on hover.)

- **Google Docs:** Google Docs uses a complex layout, but similarly, comment anchors and suggestion markers are rendered in a separate layer. When you hover over text with a comment, Docs shows a small comment icon in the margin. This icon is not part of the HTML text flow – it's drawn in an overlay that aligns with the text's position on the page. By using an overlay, Google Docs ensures the icon doesn't obstruct the caret. Clicking the icon opens the comment thread, but the editing cursor remains in place in the document text. (Internally, Google Docs likely uses a mix of contenteditable and canvas, but the overlay principle still applies – UI is drawn on top.) The approach guarantees that typing is unaffected by the presence of these hover icons.

- **Medium:** Medium's famous editor introduced the idea of a *floating formatting toolbar* that appears when you select text. Medium's implementation is proprietary, but it's known to rely on a contenteditable base. The selection toolbar is an overlay that appears above the selected text ("little bubble above the selection" [17]). It likely uses the selection's bounding rect to position itself. This toolbar only appears on selection (not just hover), which avoids interfering with a moving cursor. Medium also had a feature where hovering a link showed a small tooltip with the URL and options to edit/remove – again done with an overlay near the link. In all cases, the design is such that the overlay doesn't shift any text (it floats above) and typically disappears or moves out of the way as soon as you continue typing.

- **Slate (React-based):** Slate is a framework where you manage rendering via React. The common approach for a hovering or selection toolbar in Slate is to use a React Portal or absolutely positioned component. For example, Slate's **Hovering Toolbar** example renders a `<div>` for the menu and uses the DOM `Selection` API to get the range's `getBoundingClientRect()`. The toolbar component is usually rendered outside the contenteditable (e.g. in a portal to the `<body>` or a parent) and its style.left/top is set based on the selection rect. This is done on each selection change. The Slate team demonstrates centering a menu above the selection and only showing it when the selection is not collapsed (i.e., there is selected text) [18] . The logic often involves `window.getSelection()` and listening for Slate's selection change events to trigger a re-position. Because Slate's content is still a normal DOM (contenteditable), the underlying strategy (absolute overlay + selection rect) is similar to ProseMirror's. The difference is that in React you might use state hooks and effects (as one blog post does) to update the toolbar position on events like `mouseup` or on Slate's onChange. The result is a contextual menu that appears on selection, stays in position even if the editor is focused, and doesn't eat the user's input events (since it's a separate layer).

- **Quill:** Quill editor has a "Bubble" theme which implements a Medium-like floating toolbar. Quill's architecture uses modules and themes to achieve this. In the Bubble theme, when text is selected, a tooltip toolbar is shown near the selection (with formatting buttons). Quill's tooltip is rendered within the editor container but positioned above the text. One known detail is that Quill constrains the tooltip within the editor's boundaries. For example, if a link tooltip would go out of the editor frame at the top, Quill will adjust its positioning (or you can configure a `bounds` option to limit where the tooltip can appear) [19]. This prevents the bubble from getting cut off. The Quill docs explicitly mention it supports *“a floating tooltip theme (‘Bubble’)”* for the UI [20]. Under the hood, Quill’s tooltip is likely a fixed element overlay that gets its position updated on selection change. Quill also shows link preview/editing tooltips: when you hover or click a link in the text, a small tooltip appears with the URL and an edit option. This is implemented as part of the Link module, using a similar overlay approach.

- **TipTap:** TipTap provides ready-made extensions like **BubbleMenu** and **FloatingMenu** which abstract the logic for these overlays. The BubbleMenu extension attaches a floating menu element that follows the text selection. Notably, TipTap’s BubbleMenu uses the **Floating UI** library under the hood for positioning [21] – this helps handle cross-browser differences and automatically flips or shifts the menu to stay visible if near edges. The extension’s API lets you provide an `element` (the menu DOM node) and it handles showing/hiding it. TipTap also debounces its updates: by default it waits 250ms before re-positioning or showing the menu on state changes [22]. This debounce prevents jitter if the selection is changing rapidly (e.g., user still selecting text or moving caret). You can configure or disable this delay, but the default exists to ensure performance and a smoother UX. TipTap’s design is focused on “edit mode” usage, so they ensure the menu appears only in appropriate conditions (you can specify a `shouldShow` callback to e.g. only show for certain node types) [23] [24].

In summary, **all these editors use overlays or decorations combined with event hooks to show contextual UI without disrupting editing**. The implementations differ (custom plugins, React components, external libraries), but the core idea is the same.

Ensuring the UI Doesn’t Interfere with Typing

A crucial aspect is that these hover or selection toolbars must **not block the caret or text interactions**:

- **No layout shift:** By absolutely positioning the element (or using fixed positioning), the text does not reflow when the UI appears. For example, ProseMirror’s tooltip sets `position: absolute` and uses bottom/left offsets to sit above text [13]. The underlying text remains static, so the caret stays at the same position in the text line.

- **Pointer events and focus:** Typically, the overlay should not steal focus when it appears. In many implementations, the overlay is placed outside the contenteditable and might even be marked with `contenteditable="false"` just in case. For instance, a sample React implementation for a floating toolbar wraps the menu in a `<div contentEditable={false}>` so that the browser doesn’t treat it as part of the editable text [25]. This means clicking the toolbar’s buttons won’t place a text cursor in the toolbar element. Additionally, the toolbar container is often given `user-select: none` CSS [26] so that dragging the mouse over the toolbar doesn’t accidentally select UI text instead of editor text.

- **Preventing focus loss:** If the user clicks an icon or button in the hover UI, you generally don’t want the editor to blur (lose focus). A common trick is to intercept the mouse down on the button and `preventDefault()` – this way, the focus does not shift to the button even though it was clicked.

The action can be handled (e.g. apply formatting or open a dialog), and then the editor can explicitly retain or restore focus (e.g., call `editor.view.focus()` in ProseMirror after handling). Many editors ensure that after you click a formatting option, the text cursor is still in place so you can continue typing immediately. In cases where the overlay is purely informational (like a hover preview), they often set `pointer-events: none` on the tooltip, so that moving the mouse over it doesn't even register as a hover on the tooltip – you can click "through" it to the text if needed. That said, if the tooltip contains interactive controls, `pointer-events: none` is not used on the whole element (otherwise you couldn't click the buttons!). Instead, they carefully manage event handling. For example, Medium's toolbar only appears on text selection (when the user likely has finished selecting), and disappears when you start typing again or click off, thereby minimizing chance of conflict.

- **Safari and cross-browser quirks:** Safari (especially iOS Safari) has a number of contenteditable quirks. One known issue is that on iOS, even if you overlay something or use a node selection, the blinking caret might still be shown underneath [27]. Also, WebKit might handle focus and pointer-events differently. To avoid Safari issues with caret placement, editors often avoid covering the exact spot of the caret with any element. Positioning the hover UI **above** the text line (instead of directly over the text) is one way – e.g., the bubble menu sits a few pixels above the selected text, not on top of it. Additionally, testing on Safari might reveal a need to adjust z-index stacking contexts. For instance, if an editor or its container has a stacking context, an overlay might need a very high z-index to truly appear on top (the internal research plan hypothesized checking z-index on focus [28]). In practice, setting a sufficiently high z-index on the tooltip (and ensuring the editor doesn't create a new stacking context that hides it) is important [29]. Chrome and Firefox are generally straightforward, but each browser may have slight differences in how selection bounding rect is reported (e.g., multi-line selection rect might just cover the first line in Firefox). Using a well-tested library (like Floating UI or Popper.js) can abstract away many cross-browser positioning issues. TipTap's adoption of Floating UI is one example of leveraging a third-party for consistent behavior [21].

- **Debounce and timing:** To preserve the **typing UX**, the hover UI typically does not pop up the instant you brush the text with your cursor. A small delay is introduced to ensure the user actually meant to hover. UX guidelines often use about **300ms delay** for tooltips to appear [30]. In practice, this can be done with `setTimeout`: on `mouseenter`, schedule the icon/tooltip to show after (say) 300ms, and if a `mouseleave` happens before that, cancel the timeout. This prevents flicker when the mouse moves quickly over the text. Similarly, when hiding the UI, some implementations hide it immediately on mouse out, while others might also delay hide slightly to allow moving the mouse onto the tooltip (if it's interactive) without it vanishing. In the context of text selection, many libraries debounce updates to avoid jitter as selection changes. For example, TipTap's BubbleMenu waits 250ms on selection changes [22]. This ensures that if you are still selecting text or moving the caret, the menu doesn't continually flash or reposition on every single character change – it updates when things have settled, giving a smoother feel.

- **Positioning above the text line:** Most hover UIs are placed just above the text to avoid covering it. The ProseMirror example calculates `bottom = (container.bottom - selection.top)` to position the tooltip *above* the selection's top coordinate [31]. Many implementations also horizontally center the tooltip relative to the selection or hovered element. In the sample code above, they averaged the left coordinates of the selection start and end to find a center point [13]. Some add a slight offset (e.g. +5px up) so there's a small gap between the text and the tooltip. This attention to positioning ensures the UI is visible and clearly associated with the text, yet not occluding the text itself. If the overlay has an arrow pointer (like in the example code from the Medium article, they added a small arrow) [32], it further indicates which text it refers to.

- **Preserving caret and scroll:** Another consideration is that the presence of a floating element should not block text selection or scrolling. By using `pointer-events: none` on non-interactive parts (e.g., the arrow or background of a purely informational tooltip), the user can still highlight text through it. And because the overlay is not part of the editor's scrollable content, when the document scrolls, you must update its position or hide it. Many editors simply hide the hover UI on scroll, or recompute positions on a scroll event, to avoid it floating in an incorrect spot. This is part of cross-browser compatibility too: e.g., ensure that on Safari iOS (where scrolling can also blur focus at times), the overlay is handled appropriately.

Cross-Browser Reliability

Production editors test these features across browsers and electron environments:

- Using standard web APIs (DOM events, `getBoundingClientRect`, `selectionchange`, etc.) usually works in Chrome, Firefox, and modern Edge without issue. The differences come in mobile vs desktop (touch events vs mouse) and Safari's idiosyncrasies. It's recommended to use high-level libraries or well-documented techniques to manage these. For instance, the example floating toolbar for selection used a combination of `mousedown`/`mouseup` and the `selectionchange` event to detect when the user finished selecting text [33] [34], because on touch devices `selectionchange` is the only reliable event (there's no mouseup). This kind of logic ensures the UI appears at the right time on all input types.

- When implementing your own, be aware of browser bugs. As an example, a developer implementing a custom hover tooltip noted that on Chrome it worked fine, but on Safari some events behaved differently – they suspected the editor "might be preventing event propagation or handling it differently when focused" [35]. The solution was to attach events in the capture phase or within the editor's own handlers [16]. So, for cross-browser reliability, you sometimes need to experiment with event phases or fallback strategies (e.g., if `mouseover` isn't firing as expected, perhaps use `mousemove`).

- **Electron** (which uses Chromium under the hood) should behave like Chrome. One thing to consider in Electron apps is that modals or separate windows might handle focus differently, but for an in-page editor, the behavior is essentially the same as Chrome. If anything, ensure that your z-index stacking isn't disrupted by any Electron context (e.g., if your editor is inside a scrollable view with its own coordinate space).

- **Testing in Safari and iOS:** It's important to test in Safari desktop and on iOS Safari, because contenteditable + overlay can reveal issues like: the toolbar might not appear at the correct spot if the virtual keyboard is up (on iOS), or tapping the overlay might not work unless you call `preventDefault` on touch events. Many editors include specific tweaks for iOS (for example, disabling certain hover features on touch devices, or using a long-press instead). If hover icons are not critical on mobile, some simply disable them there.

In conclusion, the common pattern is: **listen for the relevant event (hover or text selection), use the editor's APIs to identify where it happened, and render a floating element at that location with minimal delay**. All the mentioned editors ensure the floating UI is an **enhancement that does not disrupt typing** – achieved by keeping it out of the content flow (`position:absolute`), not taking focus (use `contenteditable=false` and event tricks), and by intelligently managing its appearance (with debounced timing and smart positioning). By studying open-source examples – ProseMirror's tooltip demo, TipTap's BubbleMenu, Slate's hovering menu, Quill's bubble theme – one can find well-tested patterns to implement a reliable hover-based UI in any rich text editor environment [2] [22]. The result is a polished UX: icons and toolbars that conveniently appear when needed, across all modern browsers, without ever "getting in the way" of the user's writing.

Sources:

- ProseMirror Tooltip example (Medium-style floating menu) [3] [2]
- ProseMirror discuss forum – handling hover with plugins and `handleDOMEvents` [6] [7]
- ProseMirror discuss – Tiptap extension for Notion-like hover buttons [9] [11]
- TipTap BubbleMenu docs – debouncing and Floating UI positioning [22] [21]
- Medium-style selection toolbar implementation (N. Singh, 2025) [36] [25]
- PatternFly UX guideline on tooltip hover delay [30]
- Quill documentation (Bubble theme and tooltips) [20]

---

[1] [2] [3] [4] [5] [12] [13] [14] [17] [18] [31] ProseMirror tooltip example
https://prosemirror.net/examples/tooltip/

[6] [7] [8] Handling mouse events for nodes - discuss.ProseMirror
https://discuss.prosemirror.net/t/handling-mouse-events-for-nodes/1813

[9] [10] [11] Displaying buttons when hovering over a node - discuss.ProseMirror
https://discuss.prosemirror.net/t/displaying-buttons-when-hovering-over-a-node/5952

[15] [16] [29] [35] edit-mode-findings.md
file://file-HktAXrCVamgnd4SJJ51iTv

[19] Link tooltip cut off by edge of editor · Issue #360 · slab/quill - GitHub
https://github.com/slab/quill/issues/360

[20] Customization - Quill Rich Text Editor
https://quilljs.com/docs/customization

[21] [22] [23] [24] bubble-menu.mdx
https://github.com/centricconsulting/agent_c_framework/blob/8c6d62bfd24c88fc607f30e004e031d367b19924/src/realtime_client/ref/tiptap_docs/editor/extensions/functionality/bubble-menu.mdx

[25] [26] [32] [33] [34] [36] How to Show a Floating Toolbar on Text Selection
https://code.nkslearning.com/blogs/how-to-show-a-floating-toolbar-on-text-selection_683b57ce63482ebacfb8

27 iOS Safari does not hide the caret on non visible selections. E.g. ...
https://discuss.prosemirror.net/t/ios-safari-does-not-hide-the-caret-on-non-visible-selections-e-g-nodeselection/3024

28 edit-mode-research-plan.md
file://file-3osJ6QVNNiuxM8e8fxpcii

30 Tooltip - PatternFly
https://www.patternfly.org/components/tooltip/design-guidelines