ChatGPT

Improving the Template Editor with Block Modification Tracking

Overview and Goals

The **Create/Customize Template Editor** should let users freely edit the composed prompt preview while correctly tracking changes to individual prompt "blocks" (role, context, constraints, etc.). The goals are: (1) detect when the user edits a block's text in the `EditablePromptPreview`, (2) mark that block as modified in the UI, (3) preserve modifications even if the user switches or adds/removes other blocks, and (4) on save, create new block entries for modified blocks instead of corrupting the original references. We'll achieve this by extending the editor's state management (in hooks like `useTemplateDialogBase`) and leveraging utilities in `metadataUtils.ts` for diffing and state comparison.

Detecting Block Edits in `EditablePromptPreview`

When the user edits the full prompt content via the `EditablePromptPreview` component, we need to identify which section(s) correspond to metadata blocks and how they differ from the original backend content. We can introduce a helper like `getModifiedBlocks()` (e.g. in `utils/prompts/metadataUtils.ts`) that takes the current `PromptMetadata`, the edited `finalPromptContent` string, and the `blockContentCache` (which holds original text for each block ID [1]). This function would:

- **Parse the final prompt content into sections per metadata block.** We know the preview builder prefixes each block with a label (e.g. *"Ton rôle est de ..."* for role, *"Le contexte est ..."* for context, etc.) and separates sections with double newlines [2] [3]. Using these known prefixes or splitting by `\n\n`, the function can isolate each metadata block's text. For multiple items like constraints/examples, it would find each occurrence of the prefix (e.g. "Contrainte:") in order.
- **Compare with original block text.** For each metadata entry, retrieve the original text from `blockContentCache` by its block ID [1] (or from the `MetadataItem.value` if it was a custom value with no blockId). If the edited text differs from the original (beyond trivial whitespace), record it in a `modifiedBlocks` map as `blockId -> newContent`. Blocks that are unchanged are omitted.

Using this, we can detect, for example, that the user altered the **context** section versus the original context block's content, or edited a specific **constraint** line. This `getModifiedBlocks` logic runs when the user finishes editing the preview (e.g. on pressing the preview's "Save" button or on blur in simple edit mode). In practice, we'd integrate this into the `onFinalContentChange` handler. For example, in the Basic/Advanced editor components' internal handler, before calling `setFinalPromptContent`, we compute modifications:

```
const mods = getModifiedBlocks(metadata, newContent, blockContentCache);
for (const [id, newText] of Object.entries(mods)) {
```

```
  updateBlockContent(Number(id), newText);
}
setFinalPromptContent(newContent);
```

Here `updateBlockContent` is the existing action that updates the editor state's `modifiedBlocks` for a given block ID [4]. By doing this, we immediately sync any changed block text into the `modifiedBlocks` state. Blocks whose text now matches the original would simply not appear in `mods`; we can further ensure any stale entries are removed (e.g. if a user reverted a change back to original, we'd want to clear that block's modification).

UI Indication of Modified Blocks

With `modifiedBlocks` state populated, we should provide visual feedback in the editor that certain blocks have been edited. A pattern can follow how the editor indicates unsaved main content changes. For example, the **AdvancedEditor** already shows a badge "Unsaved changes" with a pulsing dot when the main content textarea has uncommitted edits [5]. We can introduce a similar indicator for each metadata block card (perhaps a small "(modified)" label or an icon) whenever its block ID is present in the `modifiedBlocks` map.

Concretely, in the `MetadataCard` component that renders each metadata block selection: after the user edits a block, that block's ID will be in `modifiedBlocks`. We could append a label to the block's title or content preview. For example, if a **Role** block was modified, the card for Role could show "Role (modified)" or highlight the content snippet differently. We can check `if (modifiedBlocks[blockId])` in `MetadataCard` when rendering the selected block's preview snippet (see where it currently displays the first 60 chars [6]) and conditionally style it (e.g. orange text or an asterisk) to denote it has unsaved modifications. This real-time feedback makes it clear which sections deviate from the original template blocks.

For multiple-value metadata (constraints/examples), each item in the list could similarly be marked if its `blockId` is in the modified set. This might involve rendering a small marker next to the item text or using italics. The key is that the UI should clearly distinguish edited blocks before saving, without relying on memory alone.

Isolating Changes per Block Type

It's crucial that editing one block's content doesn't unintentionally override or reset others. Our state management should **isolate modifications by block** so that switching selections or toggling metadata doesn't wipe out unrelated edits. Two scenarios illustrate this:

- **Switching a block selection after editing another**: Suppose the user modified the Context text, then decides to choose a different Role from the dropdown. We must ensure the context edit persists. In our implementation, since the Context's blockId didn't change, its entry remains in `modifiedBlocks` and will continue to override the context portion. We should avoid any logic that clears all modifications on such changes. Instead, handle changes *surgically*: when a user changes a metadata selection via `updateSingleMetadataValue`, remove only the modification tied to the previous block of that type (if any). For example, if the Role was block #12 (and maybe had a mod

entry) and the user selects a new Role (#18), we can drop the entry for #12 from `modifiedBlocks` (since that block is no longer in use). This prevents "stale" modifications from accumulating. However, do **not** touch the Context modification in this case – it stays intact. An effective way is to filter `modifiedBlocks` whenever metadata changes: compute the set of all block IDs currently referenced in `metadata` (all single metadata IDs plus all `item.blockId` in constraints/examples), and purge any keys from `modifiedBlocks` that are not in that set. This ensures only active blocks retain their overrides.

- **Adding or removing metadata blocks**: If the user adds a new constraint or removes one, we again want to persist unrelated edits. Adding a block (via `addMetadataItem`) will cause a new blank item to appear; this shouldn't affect existing `modifiedBlocks` at all. Removing a block (via `removeMetadataItem` or removing a secondary metadata type) should trigger removal of any mod entry for that block's ID (since it's no longer part of the prompt). We might integrate this cleanup in the hooks – for instance, inside `useTemplateDialogBase`, after calling `setState` for a metadata update, call another state update to prune `modifiedBlocks` as above. By scoping changes to only the affected block, we avoid "wholesale resets" of the editing session state.

Under the hood, the preview recomposition logic already merges modifications with original content. The `useBlockManager` hook's `effectiveBlockMap` combines the cached block content with any overrides in `modifiedBlocks` [7]. This means as long as a modification remains in state, it will continue to be applied to the final prompt preview even after other changes. In practice, after a user switches blocks, the `buildFinalPromptContent` will rebuild the prompt using the new block's content for that section, but still using the edited text for any other sections that have a modification [8] [9]. Thus, context edits persist while role changes, etc. The isolated diff approach ensures consistency: only the intended block text gets replaced when you pick a new block, and any edited text for untouched sections carries over.

Avoiding State Resets for a Stable Editing Session

To support a "diff-aware" session, we must prevent unnecessary re-initialization of the editor state that could erase the user's in-progress changes. Some improvements are already in place – for example, the template editor uses persistent component state and avoids remounting on tab switches (see how the Tabs no longer use dynamic keys to prevent full unmount/remount [10]). We should continue this pattern by not reconstructing the entire prompt content from scratch on every minor action, but rather updating it incrementally with respect to modifications.

One strategy is to utilize the metadata's `values` fields to temporarily hold edited text for blocks. The `PromptMetadata.values` stores custom text for single metadata when no blockId is used or when overriding content. We can leverage the utility `applyBlockOverridesToMetadata` to inject current `modifiedBlocks` into the metadata structure [11]. This function copies the metadata and, for each single-type block ID present in an overrides map, it sets that text into `metadata.values[type]` (and similarly updates `constraints[i].value` for any constraint item with an override [12]). By calling this after edits, the metadata itself carries the modified text. This is useful for two reasons:

1. It provides a source of truth for the edited content in case the UI re-renders or the user toggles into Advanced mode. For instance, the Advanced editor could read from `metadata.values.context`

to show the modified context text in a text area, even though a blockId is still selected – effectively treating it as a custom override until saved.

2. It guards against losing the edits if `finalPromptContent` is rebuilt. Even if we regenerate the preview from `metadata` and `blockContentCache`, if we have applied overrides to metadata, the builder will use those (since the preview generation can resolve metadata to content using either block content or provided custom values). In fact, our preview builder could be adjusted to call a similar "resolve" function that prefers `metadata.values` content for a block if present. This way, **no full reset** occurs – the local changes are folded into the state that the preview uses.

Additionally, ensure that when the user toggles between Basic and Advanced tabs, any pending edits are committed so they aren't lost. The code already synchronizes the main prompt content on tab switch (e.g., committing pending `content` changes in AdvancedEditor on unmount [13] [14]). We should do the same for `finalPromptContent` changes – but since we apply them immediately, the state should already be up to date. The key is not to throw away the `modifiedBlocks` or reset `finalPromptContent` unless the user explicitly cancels changes. The `discardFinalContentChanges` action should revert to the baseline content and clear modifications [15] only when the user chooses to cancel edits.

Saving Templates with New Block IDs

When the user saves the template (e.g. clicks "Save Template" or "Create Template"), we must persist any modified blocks as new records in Supabase and update the template to reference them. This prevents losing the user's customizations and avoids altering the original block definitions. The implementation for this can largely reuse the logic already being prototyped in the create dialog. In `useCreateTemplateDialog.handleComplete`, we see that if there are modifications, the code iterates through each `baseHook.modifiedBlocks` entry [16], fetches the original block data, then creates a **new block** with the modified content and a title suffixed "(Modified)" [17]. Each new block is marked as unpublished and linked to the original via a parent ID. We should adopt this approach for both **new templates and editing existing ones** (currently the code only does it for new templates when `!isEditMode`, but editing a template should likely do the same to avoid changing shared blocks).

After creating the new blocks, update the template's metadata to point to them. A helper function `updateMetadataWithNewBlocks` is used to swap out the IDs in the PromptMetadata [18]. This function goes through each single metadata field and replaces the ID if it matches one of the original IDs, and likewise updates each MetadataItem in constraints/examples [19] [20]. By running this, `finalMetadata` will map to the newly created block IDs. We then save the template with `finalContent` (the full prompt text, including user edits) and the updated metadata. The result is the saved template's `metadata` accurately references the new modified blocks in the database, while the original blocks remain untouched.

For maintainability, it's wise to encapsulate this "save with modifications" logic in a reusable utility or hook, rather than duplicating it for create vs edit. We could move `updateMetadataWithNewBlocks` into `metadataUtils.ts` and perhaps have a `persistModifiedBlocks(modifiedBlocks)` function that creates the blocks and returns the mapping. In fact, `useBlockManager` already has a similar `createBlocksFromModifications()` [21] [22] we can leverage. In a unified approach, the Template Editor's save handler could call something like:

```
const newBlockMap = await blockManager.createBlocksFromModifications();
const finalMeta = newBlockMap ? updateMetadataWithNewBlocks(metadata,
newBlockMap) : metadata;
```

This consolidates the block creation path. Ensuring this runs for both template creation and template editing modes will handle modifications consistently. The net effect is that any block the user edited gets a new ID (with "(Modified)" title) upon saving, and future loads of this template will pull in the user's custom block content via that new ID. The UI can even consider indicating these in the library as user-specific blocks.

Integrating with Existing Hooks and Utilities

To implement the above, we will make targeted enhancements to the existing hooks and utils:

- `EditablePromptPreview` **& Editors**: Adjust the `onFinalContentChange` usage in `BasicEditor` and `AdvancedEditor` to intercept changes. As described, incorporate `getModifiedBlocks()` to update state before setting the final content. The editors already have access to `metadata`, `blockContentCache`, and the `updateBlockContent` action via context (`useTemplateEditor()` provides these [23] [24]). This means the editor can call `updateBlockContent` for each changed block ID. After this, call `setFinalPromptContent(newContent)` to update the preview text. This ensures that `TemplateDialogState.modifiedBlocks` is up to date **in real-time** with the preview.

- `useTemplateDialogBase` **(or new diff logic hook)**: Add logic to keep `modifiedBlocks` in sync with metadata. A simple approach is an effect watching `state.metadata`. Whenever the metadata's block selections change, reconcile the `modifiedBlocks`:

```
useEffect(() => {
  const activeIds = new Set<number>();
  // collect all block IDs in current metadata
  Object.values(state.metadata).forEach(val => {
    if (typeof val === 'number' && val !== 0) activeIds.add(val);
    else if (Array.isArray(val)) {
      val.forEach(item => { if (item.blockId)
activeIds.add(item.blockId); });
    }
  });
  // remove mods for blocks no longer active
  setState(prev => {
    const prunedMods = {...prev.modifiedBlocks};
    for (let id of Object.keys(prunedMods)) {
      if (!activeIds.has(Number(id))) delete prunedMods[id];
    }
    return { ...prev, modifiedBlocks: prunedMods };
```

```
  });
}, [state.metadata]);
```

This ensures, for example, when a user removes a "constraint" item or changes a selection, we drop obsolete modifications. We might also invoke `applyBlockOverridesToMetadata(state.metadata, state.modifiedBlocks)` here to update `metadata.values` for any modified blocks (so the metadata carries the edited text as discussed). This can be done prior to building the preview or when switching to Advanced mode, so that if the user expands a metadata card after editing it in preview, the card could show the overridden text in a textarea (particularly if blockId is 0 or if we treat it as a custom value).

- `metadataUtils.ts`: Implement the `getModifiedBlocks(metadata, finalContent, blockContentCache)` utility as described. This can use the same prefix logic the preview builder uses. We can utilize `METADATA_CONFIGS` to get the label for each type (e.g. `METADATA_CONFIGS['role'].label` might be "Role" or a localized prefix). In practice, since the preview text includes the full phrase ("Ton rôle est de"), we might hardcode or internationalize how to locate that segment. Even a simpler approach is to reconstruct what the **original** metadata-only preview text would be (using `buildMetadataOnlyPreview`) and then do a diff between that and the edited metadata-only portion of `finalContent`. However, doing a full textual diff is complex; identifying block-by-block differences is more maintainable.

- **Reusing Patterns**: Where possible, use existing patterns like how custom single metadata values are handled. Notably, `updateSingleMetadata` clears `metadata.values[type]` when a block is selected [25] and `updateCustomValue` sets blockId to 0 and stores text in `values` when user inputs a custom string [26]. In our context, an edited block is conceptually akin to a "custom" value for that template (until saved). We can take advantage of this by, for example, setting `metadata.values.context` to the edited text if context was modified. This way, the Advanced Editor's fields (if any) can show it as an override. The `applyBlockOverridesToMetadata` we discussed does exactly this mapping of `modifiedBlocks` onto metadata values [11] and has already been written for persisting overrides across editors. We should call it whenever `modifiedBlocks` changes or before rendering advanced metadata inputs. This pattern ensures long-term maintainability by treating block modifications similarly to custom metadata values – a familiar concept in the codebase.

By implementing the above strategy, the Template Editor will become far more robust and user-friendly. Edits in the preview will be tracked per-block, visually indicated, and never lost due to UI updates. The saving logic will cleanly separate original vs user-modified content by creating new blocks in the backend (with clear lineage via `parent_block_id`). Overall, these changes isolate concerns: **diff management** in the front-end state and **block versioning** on save, using well-structured hooks and utilities. This sets a foundation for maintainable growth – for instance, if later we allow in-line editing of metadata cards themselves, we can reuse the same `modifiedBlocks` mechanism and diff logic to handle those changes uniformly. The end result is a template editor that behaves intuitively for users (no surprising resets), while keeping the underlying data consistent and versioned.

Sources: The implementation details are informed by the project's current code for prompt building and template dialogs, e.g., block content caching [1] and merge logic [7], state management of modifications

[4], UI patterns for change indicators [5], and the existing block creation on save for modified content [16] [17] which updates template metadata with new block IDs [18] [19]. These provide a solid starting point to extend the editor's functionality as described.

---

[1] [7] [8] [9] [21] [22] useBlockManager.ts

https://github.com/quentinbragard/jaydai-chrome-extension-frontend/blob/3676a93ccaa7eefc072a81ae81cc7d926d8e4e0d/src/hooks/prompts/editors/useBlockManager.ts

[2] [3] promptPreviewUtils.ts

https://github.com/quentinbragard/jaydai-chrome-extension-frontend/blob/3676a93ccaa7eefc072a81ae81cc7d926d8e4e0d/src/utils/templates/promptPreviewUtils.ts

[4] [15] useTemplateDialogBase.ts

https://github.com/quentinbragard/jaydai-chrome-extension-frontend/blob/3676a93ccaa7eefc072a81ae81cc7d926d8e4e0d/src/hooks/dialogs/useTemplateDialogBase.ts

[5] [13] [14] index.tsx

https://github.com/quentinbragard/jaydai-chrome-extension-frontend/blob/3676a93ccaa7eefc072a81ae81cc7d926d8e4e0d/src/components/dialogs/prompts/editors/AdvancedEditor/index.tsx

[6] MetadataCard.tsx

https://github.com/quentinbragard/jaydai-chrome-extension-frontend/blob/3676a93ccaa7eefc072a81ae81cc7d926d8e4e0d/src/components/prompts/blocks/MetadataCard.tsx

[10] index.tsx

https://github.com/quentinbragard/jaydai-chrome-extension-frontend/blob/3676a93ccaa7eefc072a81ae81cc7d926d8e4e0d/src/components/dialogs/prompts/TemplateEditorDialog/index.tsx

[11] [12] [25] [26] metadataUtils.ts

https://github.com/quentinbragard/jaydai-chrome-extension-frontend/blob/3676a93ccaa7eefc072a81ae81cc7d926d8e4e0d/src/utils/prompts/metadataUtils.ts

[16] [17] [18] [19] [20] useCreateTemplateDialog.ts

https://github.com/quentinbragard/jaydai-chrome-extension-frontend/blob/3676a93ccaa7eefc072a81ae81cc7d926d8e4e0d/src/hooks/dialogs/useCreateTemplateDialog.ts

[23] [24] index.tsx

https://github.com/quentinbragard/jaydai-chrome-extension-frontend/blob/3676a93ccaa7eefc072a81ae81cc7d926d8e4e0d/src/components/dialogs/prompts/editors/BasicEditor/index.tsx