



## Research Request: Step 4 - Atomic KV Cache and Mirror Update

### 1. Project Context:

We are building the "Positronic Brain," a continuous inference system. After generating a token (Step 1) and performing maintenance analysis (Steps 2a-3e), Step 3e produces a consolidated `action_list`. Step 4's responsibility is to execute this list, modifying the system's state for the next token generation cycle. The state consists of two tightly coupled components:

**KVMirror:** A metadata structure tracking token IDs, positions, brightness, etc. (Managed by the KVMirror class).

**past\_key\_values (KV Cache):** The actual tensor cache used by the transformer model during inference. Typically a tuple of tuples, where each inner tuple holds key and value tensors per layer, often with shape (batch\_size, num\_heads, sequence\_length, head\_dim).

### 2. Goal of Step 4:

Atomically apply all actions specified in the `action_list` to both the KVMirror and the `past_key_values` tensor cache. This must result in a consistent state where the mirror accurately reflects the content and structure of the updated KV cache tensors, ready for the next inference pass (Step 1 of the next tick). The model itself is not involved in these manipulations; this is direct state management.

### 3. Input Data for Step 4:

**action\_list** (from Step 3e): A list of dictionaries, each specifying an action:

`{'action': 'replace_pair', 'original_pos1': int, 'original_pos2': int, 'new_token_ids': List[int]}`: Replace the pair originally at `pos1`, `pos2` with the `new_token_ids` (TinyLlama vocab) starting at `pos1`. The number of new tokens might be 1 or more.

`{'action': 'add', 'token_id': int}`: Append the token generated in Step 1 (TinyLlama vocab) to the end of the context.

(Implicit: Step 3e ensures no 'delete' actions are passed if a 'replace\_pair' exists for the same position)

**Current KVMirror Instance:** The live metadata object reflecting the state before these actions are applied.

**Current past\_key\_values:** The KV cache tensors reflecting the state after Step 1's generation but before these maintenance actions are applied.

**Main Model (model):** Needed for calculating K/V projections for new tokens.

**Tokenizer (tokenizer):** The TinyLlama tokenizer.

### 4. Core Problems / Research Questions:

#### Tensor Deletion Strategy:

What is the most efficient and correct way to remove entries from the KV cache tensors at specific indices (`original_pos1`, `original_pos2` for `replace_pair`) along the `sequence_length` dimension?

Does this involve slicing and concatenation, or are there more optimized methods? How should this handle potential GQA/MQA structures where KV heads are repeated?

#### Tensor Insertion/Replacement Strategy:

For the `replace_pair` action, how do we insert the K/V vectors corresponding to the `new_token_ids` (potentially multiple tokens) starting at `original_pos1`?

This involves deleting the old entries (`pos1`, `pos2`) and then inserting new entries. How does the insertion affect the `sequence_length` dimension and subsequent indices?

#### Tensor Append Strategy:

For the `add` action, what is the correct way to append the K/V vectors for the new `token_id` to the end of the existing key/value tensors in each layer?

#### K/V Vector Calculation:

For `replace_pair` and `add` actions, we need to calculate the new K/V vectors. What is the precise

process using the model's projection layers and applying RoPE (Rotary Position Embeddings) correctly based on the final position of the token after all deletions/insertions are accounted for? (Leverage KVCachePatcher logic details if available, but focus on when and how it's called within Step 4).

**Index Management & Order of Operations:**

Applying deletions changes the indices of subsequent tokens. Applying insertions also changes indices. What is the required order to process the action\_list (replace\_pair vs. add) to ensure indices are handled correctly?

Should all deletions implied by replace\_pair happen first, followed by insertions/appends, with index adjustments between phases? Or should each replace\_pair be handled atomically (delete pos1, delete pos2, insert at pos1, adjust indices)?

How do we map the original\_pos1, original\_pos2 from the action list to the correct tensor indices at the moment the action is executed, considering prior modifications within the same Step 4 execution?

**KVMirror Synchronization:**

How do we ensure that every modification to the past\_key\_values tensors (deletion, insertion, append) has a perfectly corresponding update in the KVMirror (using its prune, add, update\_token, etc. methods or potentially new methods)?

Specifically, how does the KVMirror handle the replace\_pair action where two positions are replaced by potentially 1 or more new tokens starting at the first position? Does the mirror need a new method, or can this be composed from existing ones (prune, add) with careful index management?

**Atomicity:** How can we guarantee that the entire sequence of updates to both the mirror and the cache for a single tick appears as a single atomic transaction to the next inference step?

**Efficiency:** Are there specific PyTorch functions or techniques particularly well-suited for these kinds of KV cache manipulations (sparse updates, efficient concat/slice)?

**5. Desired Output:**

We need a detailed, step-by-step algorithm or high-level pseudocode for Step 4 that takes the specified inputs and correctly produces the updated KVMirror state and past\_key\_values tensors, addressing the challenges of tensor manipulation, index management, and synchronization. The algorithm should specify the order of operations and how indices are tracked/adjusted throughout the process.

I've included a sample of the data we want to use, showing the formatting this step will be receiving.

Thanks for the detailed context. To ensure I tailor the research precisely, could you please clarify the following:

- 1. Target Frameworks or Libraries:** Are you primarily using PyTorch, and if so, any specific version or custom KV cache structures (e.g., from HuggingFace or custom model classes)?
- 2. Atomicity Guarantees:** Do you expect strict atomicity with rollback (e.g., via transactional memory or a staged commit), or is it sufficient that no partial state is exposed to Step 1 of the next tick?
- 3. RoPE Details:** Are you already handling RoPE (Rotary Positional Embeddings) externally in KVCachePatcher or similar, or should the research identify best practices for applying RoPE inline with projection generation?
- 4. Output Preference:** Would you like the Step 4 algorithm/pseudocode output in plain English, annotated Python-like pseudocode, or real Python code (annotated)?

Once I have these details, I'll start the research immediately.



Excellent clarifying questions. Here are the answers based on our project setup and goals:

1. **Target Frameworks/Libraries:** Yes, we are using **PyTorch** and standard **Hugging Face `transformers`**. The KV cache (`past_key_values`) is expected to be the standard nested tuple structure returned by Hugging Face models, or potentially the `DynamicCache` object from newer versions (the implementation should ideally be robust to both if feasible, but prioritize the tuple format if necessary). We are not using any custom KV cache structures.
2. **Atomicity Guarantees:** Strict atomicity with rollback is not required. It is **sufficient that no partial state is exposed** to Step 1 of the next tick. The simplest acceptable failure mode would be to discard the *entire* set of actions from the current tick if any part of Step 4 fails, reverting to the state immediately after Step 1. A staged commit (calculate all changes, verify, then apply) would also be acceptable. Preventing Step 1 from seeing a mismatched mirror/cache is the key.
3. **RoPE Details:** Our existing (though not yet integrated into this test loop) `KVCachePatcher` class *does* contain logic to handle K/V projection and RoPE application. The research should assume this logic can be leveraged but needs to detail **how and when to apply it within Step 4's sequence**, ensuring RoPE uses the **correct final token positions** after deletions and insertions are accounted for. Best practices for integrating this calculation efficiently within the update loop are needed.
4. **Output Preference:** **Annotated Python-like pseudocode** would be the most helpful output format. This allows for clear algorithmic steps and variable tracking without requiring syntactically perfect Python, making it easier to translate into actual code.

Thank you for these clarifying questions. I look forward to the research results!

Perfect. I'll now begin assembling a robust, annotated Python-like pseudocode algorithm for Step 4 that ensures atomic, consistent updates to both the Hugging Face-compatible KV cache and KVMirror. I'll also detail how to handle index shifts, apply RoPE with correct positions, and leverage your existing KVCachePatcher logic. I'll let you know as soon as it's ready for review.



Step 4 Algorithm for Atomic KVMirror and KV Cache Updates

## Step 4: KV Cache and KVMirror Update Algorithm

### Overview

Step 4 performs **atomic updates** to both the transformer's KV cache and the `KVMirror` metadata object according to a list of token-level maintenance actions. These actions (from Step 3e) include replacing a pair of tokens with new token(s) (`replace_pair`) and appending a token at the end (`add`). The algorithm must ensure that the KV cache (Hugging Face-style `past_key_values`) and the `KVMirror` (tracking token IDs/positions) remain consistent after all modifications, so that the next inference step sees a coherent context (no partial updates). We achieve this by staging all changes and applying them in a careful order, then committing the new state only after all computations succeed.

## Key Considerations

- **Order of Operations (Deletion vs Insertion):** To avoid complications from index shifting, we handle all token deletions *before* insertions. Removing tokens first ensures insertion indices can be calculated relative to the shortened sequence. Another equivalent approach is to process composite replacements in descending order of positions, so higher-index removals don't affect lower-index positions. In either case, we update insertion indices to account for any removed tokens that were before the insertion point.
- **Efficient Tensor Splicing:** Removing or inserting entries in the middle of large tensors is done via slicing and concatenation in PyTorch. We avoid expensive element-wise operations by slicing contiguous blocks. For multiple deletions, removing from highest index to lowest ensures we don't have to continuously adjust indices. We use functions like `torch.cat` or advanced indexing (`index_select`) to drop or add rows along the sequence dimension of the key/value tensors.
- **Computing New Key/Value Entries:** For any new tokens (from replacements or additions), we must compute their key/value vectors with the correct positional encoding (RoPE) based on their *final* position in the sequence. We leverage the provided `model` and `KVCachePatcher` utility to generate these key/value tensors. The idea is to feed the model with the appropriate left-context (past) and new token(s) to obtain the projected key/value for those tokens, ensuring RoPE is applied using the token's new position index.
- **Synchronization with KVMirror:** Every change in the KV cache must be mirrored in the `KVMirror` object. When tokens are removed, we prune their entries from the mirror. When new tokens are inserted or appended, we add corresponding metadata (token IDs, updated positions, etc.) to the mirror. This keeps `KVMirror` as an exact reflection of the KV cache's token sequence.
- **Atomicity and Consistency:** To guarantee that no partial update is observable, we perform all modifications in a staged manner. If an error occurs (e.g. a model computation fails), we can abort and leave the original KV cache and mirror untouched. In practice, this can mean working on local copies of the cache and mirror (or buffering changes) and only assigning them back to the main objects after all steps succeed. Because we handle deletions and insertions as an integrated sequence before returning to inference, the next inference cycle will only see either the fully updated context or none of the changes at all.

## Algorithm Steps

### 1. Preprocessing and Validation

Before modifying anything, we interpret and organize the action list for predictable processing:

- **Separate Actions by Type:** Extract all `replace_pair` actions and all `add` actions from the `action_list`.
- **Sort Replacement Actions:** To simplify index management, sort the `replace_pair` actions in **descending order** of `original_pos1`. This way, we will remove higher-index tokens first. (If we instead removed lower indices first, the positions of later tokens would shift left, invalidating the original indices. Descending order avoids this problem.)

- **Validate Positions:** (Optional) Use the `KVMirror` to confirm that all referenced positions (`original_pos1`, `original_pos2`) are within the current sequence length. This ensures we don't attempt to remove or insert out-of-range positions.

```
python

# Phase 1: Preprocessing and sorting of actions
replace_actions = [act for act in action_list if act['action'] == 'replace_pair']
add_actions = [act for act in action_list if act['action'] == 'add']

# Sort replacement actions by descending starting position to avoid index shifts during removals
replace_actions.sort(key=lambda act: act['original_pos1'], reverse=True)

# (Optional) Determine current sequence length from KVMirror or past_key_values
current_len = KVMirror.length() # assuming KVMirror can provide current token count
for act in replace_actions:
    assert 0 <= act['original_pos1'] < current_len and 0 <= act['original_pos2'] < current_len
for act in add_actions:
    # Add actions append to end, so just ensure we have a valid token_id
    assert 'token_id' in act
```

## 2. Deletion Phase (Removing Tokens)

In this phase, we remove all tokens that need to be deleted (those involved in replacements) from both the KV cache and the mirror. By performing **all deletions up front**, we create a consistent shortened context into which new tokens can be inserted. This prevents any new insertion from having to guess how earlier removals shifted the indices. Key steps:

- **Identify Removal Indices:** For each `replace_pair` action, note the two positions `original_pos1` and `original_pos2` that must be deleted. Collect all such indices into a list.
- **Remove from KV Cache:** For each layer in `past_key_values`, remove the key and value entries at those indices. In PyTorch, we do this by slicing the tensor on the sequence dimension. For example, to remove a single position `pos` from a tensor `K` of shape `[batch, heads, seq_len, dim]`, we can do:
 

```
new_K = torch.cat([K[:, :, :pos, :], K[:, :, pos+1:, :]], dim=2)
```

 This concatenates the segment before `pos` with the segment after `pos`, effectively dropping the `pos`-th entry. We perform this for both the key and value tensor of each layer. If we have multiple positions to remove, we can either do them one by one (updating indices if needed) or prepare an index mask. Here we remove them one at a time in descending order to avoid index recalculation.
- **Update KVMirror:** In parallel, remove the corresponding token entries from `KVMirror`. If `KVMirror` is a list-like structure of tokens, removing an element will automatically shift subsequent tokens' positions down by one. Many `KVMirror` implementations provide methods like `prune(index)` or similar – we call those for each removed position.
- **Effect on Sequence:** After this step, the KV cache's sequence length is reduced. Tokens that were to the right of any deletions have now shifted left in the sequence (their index in the cache is lower than before). The `KVMirror` now reflects this new shorter sequence (original tokens minus the removed ones).

python

```
# Phase 2: Remove tokens for all replacement actions (process in descending order of positions)
# Collect all positions to remove (each replace_pair has two positions)
positions_to_remove = []
for act in replace_actions:
    positions_to_remove.append(act['original_pos1'])
    positions_to_remove.append(act['original_pos2'])
# Sort removal positions in descending order to handle highest index first
positions_to_remove.sort(reverse=True)

for pos in positions_to_remove:
    # Remove token at index `pos` from each layer's K and V tensor
    for layer_idx, (K, V) in enumerate(past_key_values):
        # Splice out the position `pos` along sequence dimension (dim=2)
        new_K = torch.cat([K[:, :, :pos, :], K[:, :, pos+1:, :]], dim=2)
        new_V = torch.cat([V[:, :, :pos, :], V[:, :, pos+1:, :]], dim=2)
        past_key_values[layer_idx] = (new_K, new_V)
    # Remove the token from the KVMirror as well
    KVMirror.remove_token(pos)
```

After the deletion phase, the `past_key_values` tuple now represents a context with all target tokens removed. For example, if we removed two tokens at positions 48 and 49, every token that was originally at index  $\geq 50$  has now shifted 2 spots to the left. We haven't inserted anything yet, so the sequence is temporarily shorter. Next, we will insert new tokens into these gaps. (If an error were to occur at this stage, we could abandon the changes and still have the original state safe, since we haven't committed global state yet.)

### 3. Insertion Phase (Inserting New Tokens)

This phase handles adding new tokens to the sequence for both replacements and appends. We break it into two parts: inserting replacement tokens (which may occur in the middle of the sequence) and then appending any new tokens at the end. Doing replacements first (especially in descending order of their original positions) ensures that by the time we append at the end, all middle insertions are settled. For each insertion, we must compute the new tokens' key/value tensors and merge them into the cache at the correct location. We also update the mirror accordingly.

#### 3.1 Inserting Replacement Tokens (Mid-Sequence)

For each `replace_pair` action (which we sorted by descending original position):

- Determine Insertion Index:** Compute the index in the current (post-deletion) sequence where the new token(s) should be inserted. If we removed tokens earlier, this index may differ from the original position. One way to calculate it is: `insertion_index = original_pos1 - (number of removed tokens that were at positions less than original_pos1)`. Since we removed everything earlier, we can derive this count from the `positions_to_remove` list. (If we removed in descending order as above, by the time we insert, `KVMirror` also could give us the updated index for what was originally `original_pos1`.) In practice, because we removed the exact tokens at `original_pos1` and `original_pos2`, the insertion index will simply be `original_pos1` in the current shortened sequence (all tokens up to `original_pos1-1` are still there, and those at `original_pos1` and `original_pos2` are gone).
- Compute Key/Value for New Token(s):** Use the model and `KVCachePatcher` to generate the key and value tensors for the new token(s). We supply the model with the *left context* up to the



insertion point as ``past_key_values``, and the new token IDs as input. For example, if we are inserting at position 48, we take the first 48 tokens of the current cache as ``past`` context. The ``KVCachePatcher`` (or the model's forward pass) will return the key/value for the new tokens with RoPE applied as if they occupy positions 48, 49, ... (depending on how many tokens we insert). It's important that RoPE uses the **final positions**: since we know the new token will sit at index 48, we want its rotary phase to correspond to position 48. By using the length of the left context (48) as the position offset, the model inherently applies the correct rotation. *(Under the hood, Hugging Face models use the length of ``past_key_values`` to determine positional indices for newly generated tokens. KVCachePatcher likely wraps this logic to make insertion easier.)*

- **Insert into KV Cache:** Once we have the new tokens' key/value tensors for each layer, we insert them into the cache at the computed index. This is done by concatenating the left segment (up to ``insert_idx``), the new token's K/V, and the right segment (from ``insert_idx`` onward) for each layer. If multiple tokens are inserted (the action could specify "one or more new tokens"), their key/value tensors will likewise be a sequence that we splice in. We ensure to maintain the correct order given by the model output.
- **Update KVMirror:** We then insert the new token metadata into ``KVMirror`` at the appropriate position. For example, if two tokens were replaced by one token, we remove two entries earlier and now insert one new entry at ``original_pos1``. The mirror should reflect that the new token now occupies position ``original_pos1``, and subsequent tokens have shifted one position left (due to net removal of one). If more than one token is inserted, we insert all of their IDs starting at ``original_pos1``. The ``KVMirror.insert_tokens(pos, ids)`` method (hypothetical) would assign correct position indices to the inserted tokens and shift the following tokens accordingly.

We perform each replacement insertion in sequence (following the sorted order). By using the descending sorted order, when we insert into a lower index, we don't have to adjust the yet-to-be-inserted higher indices because those were handled earlier. Each insertion operates on the current state of the cache/mirror, which already excludes all removed tokens.

python

```
# Phase 3.1: Insert new tokens for each replace_pair action (in original descending order)
for act in replace_actions:
    orig_pos = act['original_pos1']
    new_token_ids = act['new_token_ids'] # list of one or more token IDs to insert

    # Calculate the insertion index in the current sequence (post-deletion)
    # This is essentially orig_pos minus any previously removed tokens before it.
    removed_before = sum(1 for r in positions_to_remove if r < orig_pos)
    insert_idx = orig_pos - removed_before
    # (After deletion phase, if using descending order, removed_before for orig_pos will effectively
    # be the number of tokens removed before orig_pos. Still, we include the formula
    # because none of the removals with index < orig_pos are pending. Still, we include the formula)

    # Prepare left-context past for model: take cache entries up to insert_idx as past_key_values
    past_left = []
    for (K, V) in past_key_values:
        past_left.append((
            K[:, :, :insert_idx, :], # keys up to (but not including) insert_idx
            V[:, :, :insert_idx, :]
        ))

    # Use the model and KVCachePatcher to get K,V for the new token(s)
    # (KVCachePatcher will handle embedding the token IDs and applying RoPE for the correct position)
    new_kv_per_layer = KVCachePatcher.compute_kv(model, past_left, new_token_ids)
    # new_kv_per_layer is a list/tuple of (new_K, new_V) for each layer, where new_K/V have shape
    # [batch, heads, len(new_token_ids), head_dim] corresponding to the inserted token sequence.

    # Insert the new keys and values into each layer's cache at insert_idx
```

```
for layer_idx, (K, V) in enumerate(past_key_values):
    new_K, new_V = new_kv_per_layer[layer_idx] # key/value for this layer of the new tokens
    # Splice in the new K,V at position insert_idx
    updated_K = torch.cat([K[:, :, :insert_idx, :], new_K, K[:, :, insert_idx:, :]], dim=2)
    updated_V = torch.cat([V[:, :, :insert_idx, :], new_V, V[:, :, insert_idx:, :]], dim=2)
    past_key_values[layer_idx] = (updated_K, updated_V)
# Update the KVMirror by inserting the new token IDs at insert_idx
KVMirror.insert_tokens(insert_idx, new_token_ids)
```

*Explanation:* In the code above, `KVCachePatcher.compute_kv`` represents a utility method that handles feeding the model with `past_left`` and `new_token_ids`` to obtain the new key/value tensors. We assume it returns a structure (`new_kv_per_layer``) containing the new key and value for each transformer layer. We then merge those into our `past_key_values`` via concatenation. The mirror's insert method would take care of assigning correct position metadata for the new tokens. We iterate this for each replacement action. By the end of this loop, all mid-sequence replacements have been applied: the KV cache now includes all new keys/values for replacement tokens, and the sequence indices align with the new context (no old tokens lingering).

### 3.2 Appending Added Tokens (End-of-Sequence)

After processing replacements, we handle any `add`` actions, which append new token(s) to the end of the context. These are simpler because they don't require removing existing tokens or splicing in the middle; they always go after the current last token. We still need to compute their key/value and update the mirror:

- **Determine Append Position:** The position for an appended token will be the new end of sequence. For example, if the sequence length is currently `L`` (after any prior insertions), the new token will occupy position `L`` (0-indexed). We can get the current length from `past_key_values`` (e.g. by checking the shape of the key tensor of the first layer) or from `KVMirror.length()``.
- **Compute Key/Value for New Token:** Use the model (via `KVCachePatcher``) to compute the key and value for the new token, using the entire current sequence as the past context. Essentially, this is equivalent to how one would generate the next token during normal autoregressive decoding. The `past_key_values`` at this point already includes all prior tokens, so feeding that along with the new token ID will yield the K/V at the correct final position. (RoPE will be applied with position index equal to the old sequence length.)
- **Append to KV Cache:** Insert (concatenate) the new key and value to the end (sequence dim) of each layer's tensor. Because this is at the end, we can simply use `torch.cat([K, new_K], dim=2)`` without any slicing for the right segment.
- **Update KVMirror:** Append the token's metadata to the mirror. The new token gets the last position (which should match the new length - 1 after appending). Many implementations have an `append_token(id)`` or similar to add a token at the end.

We perform each add action sequentially. If multiple tokens are to be added, each iteration extends the sequence by one. (Alternatively, if the actions are independent and order doesn't matter, one could batch them, but typically the action list order is the intended order of addition, so we respect it.)



```
python
```

```
# Phase 3.2: Append tokens for each add action (extend sequence at the end)
for act in add_actions:
    token_id = act['token_id']
    # Determine current length (position where new token will be inserted)
    current_len = past_key_values[0][0].size(2) # length of sequence from first layer's key tensor
    insert_idx = current_len # new token will go at this index (the current end)

    # Use full current cache as past context to compute new token's K,V
    full_past = past_key_values # using the entire current past as context
    new_kv_per_layer = KVCachePatcher.compute_kv(model, full_past, [token_id])
    # new_kv_per_layer gives us the key/value for the new token at position = current_len

    # Append the new key and value to each layer
    for layer_idx, (K, V) in enumerate(past_key_values):
        new_K, new_V = new_kv_per_layer[layer_idx] # should have shape [batch, heads, 1, head_dim]
        past_key_values[layer_idx] = (
            torch.cat([K, new_K], dim=2),
            torch.cat([V, new_V], dim=2)
        )
    # Update KVMirror to reflect the added token at the end
    KVMirror.append_token(token_id)
```

After all `add` actions are processed, the KV cache has grown in length by the number of added tokens, and the mirror includes those tokens at the tail. We have now finished applying all modifications from `action_list`: all specified tokens have been removed, all new tokens inserted, and both data structures (KV cache and mirror) are up-to-date and aligned with each other. Throughout insertion, if any step failed (for instance, if the model couldn't produce a key/value for some reason), we could roll back by discarding `past_key_values` and `KVMirror` changes (which we kept isolated) and retain the original state. Assuming success, we proceed to finalize the update.

## 4. Finalization and Atomicity

In the final step, we **commit** the staged updates to the system so that subsequent inference steps use the new context. If we were operating on copies or a buffered version of the KV cache and mirror, this is where we would replace the original `past_key_values` and `KVMirror` with the updated ones. Since our pseudocode above directly mutated `past_key_values` and `KVMirror` (for simplicity), we will treat those as our new official state now that all actions are applied.

To ensure atomicity, no other process should read the KV cache or mirror while the update is in progress. In a single-threaded setting, this is naturally handled by performing all steps sequentially before returning control to inference. In a multithreaded or asynchronous setting, one might lock these structures or use a temporary context. The key point is that we never left the data in a partial state between actions – we did all the removals and insertions as one routine. Now the update is effectively “all-or-nothing”: either all changes are applied or none (if an error caused an abort).

```
python
```

```
# Phase 4: Finalize the update (commit the new cache and mirror)
# At this point, past_key_values and KVMirror have been updated in memory.
# If we used separate variables (e.g., new_past_key_values), we would now swap them in:
official_past_key_values = past_key_values
official_KVMirror = KVMirror

# The updated cache and mirror are now ready for use in the next inference step.
# (If any failure occurred earlier, we would skip this assignment, leaving the old state intact.)
```

With this final step, the system's context is now updated. The next iteration of Step 1 (model inference) can consume ``official_past_key_values`` and ``official_KVMirror`` as if they came from a continuous context, with no knowledge that tokens were removed or inserted except through their effect on the sequence content. The update was done atomically in the sense that intermediate states (e.g., after deletions but before insertions) were never used for inference.

## Handling Different Cache Types (Tuple vs. DynamicCache)

The above algorithm was described using the typical **tuple of tensors** representation for ``past_key_values``. Newer Hugging Face models may use a ``DynamicCache`` (or other ``Cache`` classes) instead of a raw tuple. The logic of the update remains the same, but a few adaptations are needed for ``DynamicCache``:

- **Accessing the Data:** A ``DynamicCache`` object internally manages key/value tensors. We might need to extract or reference those internal tensors. For example, ``DynamicCache`` often has attributes or methods to get the underlying key stack and value stack per layer. We can perform the same slicing (``torch.cat`` operations) on those.
- **Using Cache Utility Methods:** If available, prefer using any built-in methods for pruning or inserting in the dynamic cache. For instance, if ``DynamicCache`` provides a method to remove a range of positions (as some sliding-window caches do for truncation), that could be used for the deletion phase. If no such high-level method exists, we handle it manually as above by manipulating the tensors.
- **Converting to Legacy (if needed):** One convenient approach is to convert the dynamic cache to the "legacy" tuple format, perform the update as described, and then convert it back. Hugging Face provides utilities like ``DynamicCache.from_legacy_cache(old_tuple)`` and similar. So we could do: convert ``past_key_values`` (DynamicCache) to a tuple, run all the steps, then wrap the result back into a DynamicCache. This ensures compatibility without having to deeply know DynamicCache internals.
- **RoPE and KVCachePatcher:** The use of ``KVCachePatcher`` and model forward pass is unaffected by the cache type – it will handle the rotary positional encoding consistently. The main difference is in how the cache is updated in memory. After computing new keys/values, if using ``DynamicCache``, we might call methods to append them (since DynamicCache is designed to grow). For deletion, we might reconstruct a new DynamicCache with the shorter sequence.
- **Atomicity:** Regardless of cache type, the update should be applied in one go. If using a dynamic cache, one could accumulate the changes and then assign the dynamic cache's internal state at the end (or replace the object entirely). The principle of not exposing a partially updated cache holds.

In summary, **Step 4's algorithm** meticulously removes specified tokens and inserts new ones into the transformer's KV cache and the KVMirror, doing so in a safe order (deletions first, then insertions) to maintain index consistency. It leverages model computations (with proper positional encoding) to get the new key/value entries for any inserted tokens. Throughout the process, every change to the cache is mirrored in the metadata, and no half-finished state is ever used for inference. After completion, the KV cache and KVMirror present a unified, updated context ready for the model to continue generating with the modifications in effect. This ensures that the Positronic

Brain's context maintenance is robust, efficient, and free of race conditions or indexing errors, fulfilling the requirements for Step 4.

#### Sources



I'm of the opinion that we have to delete last. Or at least that we have to do all commands in order from highest to lowest. If we are supposed to delete 45 and change 56 later, 56 becomes 55. I think that's the only way to keep the references clean.

You're absolutely right that **executing every action in strict descending order of the highest-referenced position keeps all subsequent references valid without extra bookkeeping**. When we delete (or splice-replace) a high index first, nothing "to its left" has moved yet, so the original indices of the remaining, lower-indexed actions remain intact.

Below is an *updated* Step 4 algorithm that takes this principle to heart. Each action is handled atomically—"delete last" in the local scope of that action—and the overall action list is processed **from highest to lowest** `original\_pos1` (ties broken by `original\_pos2`). Appends still occur afterward, because they never interfere with interior indices.

## Revised Algorithm (annotated Python-like pseudocode)

python

```
#####
# Phase 0 – Preparation
#####
# Split actions
replace_actions = [a for a in action_list if a['action'] == 'replace_pair']
add_actions      = [a for a in action_list if a['action'] == 'add']

# Sort _all_ interior replacements by descending first index
replace_actions.sort(key=lambda a: a['original_pos1'], reverse=True)
# Example order: (72,73) → (56,57) → (45,46) ...

#####
# Phase 1 – Process each replace_pair atomically, highest → lowest
#####
for act in replace_actions:
    pos1, pos2 = act['original_pos1'], act['original_pos2']    # pos1 < pos2
    new_ids    = act['new_token_ids']                        # list[int]

    # ---- 1a. Compute K/V for the replacement tokens -----
    # • Current cache length _already_ reflects any higher-index edits.
    # • Therefore the insertion index for the first new token is still `pos1`.
    left_context_len = pos1                                # tokens [0, pos1)
    past_left = [
        (K[:, :, :left_context_len, :], V[:, :, :left_context_len, :])
        for (K, V) in past_key_values
    ]
    new_kv_per_layer = KVCachePatcher.compute_kv(
        model,
        past_left,
        new_ids                                # one or more IDs
    )
    # new_kv_per_layer[i] = (new_K, new_V) with seq_dim == len(new_ids)
```

```
# ---- 1b. Splice KV cache -----
# • Delete pos2 first, THEN pos1, so indices remain stable.
# • Finally insert the new K/V block at pos1.
for layer_idx, (K, V) in enumerate(past_key_values):
    # Delete pos2
    K = torch.cat([K[:, :, :pos2, :], K[:, :, pos2+1:, :]], dim=2)
    V = torch.cat([V[:, :, :pos2, :], V[:, :, pos2+1:, :]], dim=2)

    # Delete pos1 (pos2 no longer exists, so indices are still correct)
    K = torch.cat([K[:, :, :pos1, :], K[:, :, pos1+1:, :]], dim=2)
    V = torch.cat([V[:, :, :pos1, :], V[:, :, pos1+1:, :]], dim=2)

    # Insert new tokens at pos1
    new_K, new_V = new_kv_per_layer[layer_idx] # shape ..., L_new, ...
    K = torch.cat([K[:, :, :pos1, :], new_K, K[:, :, pos1:, :]], dim=2)
    V = torch.cat([V[:, :, :pos1, :], new_V, V[:, :, pos1:, :]], dim=2)

    past_key_values[layer_idx] = (K, V)

# ---- 1c. Update KVMirror -----
# • Prune indices pos2 then pos1, then insert new_ids at pos1.
KVMirror.remove_token(pos2)
KVMirror.remove_token(pos1)
KVMirror.insert_tokens(pos1, new_ids)

#####
# Phase 2 – Handle all add actions (append after interior edits)
#####
for act in add_actions:
    token_id = act['token_id']
    cur_len = past_key_values[0][0].size(2) # sequence length now
    new_kv = KVCachePatcher.compute_kv(model, past_key_values, [token_id])

    for layer_idx, (K, V) in enumerate(past_key_values):
        new_K, new_V = new_kv[layer_idx] # seq_dim == 1
        past_key_values[layer_idx] = (
            torch.cat([K, new_K], dim=2),
            torch.cat([V, new_V], dim=2)
        )

    KVMirror.append_token(token_id)

#####
# Phase 3 – Commit
#####
# If we manipulated copies, swap them in here.
# Otherwise, execution reaches this point only if every step above succeeded,
# so the in-place updates are already coherent for the next inference tick.
```

## Why this works

Design choice	Rationale
<b>Descending order traversal</b>	Once a high-index edit finishes, every remaining action touches a strictly lower index, which is thus still valid. No cumulative “index-shift bookkeeping” is needed.
<b>Delete-then-insert <i>within</i> each replacement</b>	Purges obsolete K/V rows before growing the tensors, so memory never balloons. Doing the two deletions in descending order (`pos2` → `pos1`) guarantees `pos1` is still the same row when we remove it.
<b>Compute K/V <i>before</i> splicing</b>	`KVCachePatcher` sees an intact left context, so RoPE phases are correct for positions `pos1 ... pos1+len(new_ids)-1`.
<b>Append phase last</b>	End-of-sequence growth never perturbs existing indices, so it’s safe to leave until every interior mutation is done.

## Practical notes

- **Atomicity:** Because all mutations occur in a single routine (or under a Python `with torch.no_grad()` lock if multithreaded), Step 1 will never glimpse a half-updated cache. If you want hard rollback, work on shallow copies of the tuple or on a `DynamicCache.to_legacy()` clone, and discard on exception.
- **DynamicCache compatibility:** Replace the direct tensor surgery with `cache.prune(start, end)` and `cache.insert(k_block, v_block, index)` if those helpers exist; the index-ordering logic is identical.
- **Memory locality:** Deleting first and inserting immediately keeps peak tensor size near `old_len + max(len(new_ids)-2)`. That's friendlier to GPU memory than a "delete-later" policy that would temporarily hold both the obsolete rows and the replacements simultaneously.

## TL;DR

Process **every action in descending index order**.

For each replacement: *delete higher row* (`pos2`), *delete lower row* (`pos1`), *then splice in the new tokens at* `pos1`, updating the mirror in lock-step. Finish with any tail-appends. This approach matches the intuition you described—no index ever changes before an action that depends on it executes—while also keeping tensor operations simple, contiguous, and memory-efficient.



Bryn: It's important to me that we don't update the cache, and then the mirror. I want every change applied simultaneously to both. So if we delete 61 from the cache we delete 61 from the mirror as part of the same exact function being passed the same parameter. You cannot call a delete and have it just affect one source or the other. The two need to be bolted together, a knife with two blades. This mechanical lock-step is what step 4 is about, more than editing the cache it is about a neurotic obsession with control of the cache and jealous stewardship of the mirror.

Gemini: Understood. Your emphasis on the "two blades of the knife" – the absolute, indivisible coupling of cache tensor manipulation and mirror metadata update – is crucial. The pseudocode might have looked like separate steps, but the *implementation* must guarantee this lock-step execution.

We need to design Step 4 such that each logical action (`delete`, `insert`, `append`) operates on both the cache and the mirror as a single, conceptual, atomic unit. Failure in one part (tensor manipulation) must mean failure of the other (mirror update) for that action, and potentially for the entire Step 4 tick.

Let's refine the Step 4 specification to reflect this stringent requirement. We'll define conceptual "Coupled Operations" that Step 4 will orchestrate.

---

**\*\*REVISED Research Request: Step 4 - Tightly Coupled KV Cache and Mirror Update\*\***

**\*\*1. Project Context:\*\*** (Same as before) ...Positronic Brain, Halo Weave, KVMirror, KV Cache...

**\*\*2. Goal of Step 4:\*\***

Apply all actions from the ``action_list`` (generated by Step 3e) to the system state. Each action must modify **both** the ``past_key_values`` tensor cache **and** the ``KVMirror`` metadata **simultaneously and consistently**. The final state must be coherent and ready for the next inference cycle, with atomicity ensured per tick (either all actions succeed, or the state reverts to before Step 4). Direct manipulation only; no model inference *during* the update process itself, only for calculating K/V vectors for *new* tokens.

**3. Input Data for Step 4:** (Same as before) ...``action_list``, ``KVMirror``, ``past_key_values``, ``model``, ``tokenizer``...

**4. Core Problem Reframed: Coupled State Management**

The central challenge is designing and executing **Coupled Operations** that guarantee synchronized modifications to the tensor cache and the metadata mirror.

**5. Required Coupled Operations (Conceptual):**

Step 4's implementation will rely on functions/methods performing these coupled tasks:

```
* Coupled_Delete(position, kv_cache, kv_mirror):
* Removes the entry at the current effective index `position` from the key and value tensors in each layer of `kv_cache`.
* Simultaneously, removes/marks-as-pruned the corresponding token metadata at `position` within `kv_mirror`, handling any necessary index shifts within the mirror structure itself.
* Returns (updated_kv_cache, updated_kv_mirror)` or signals failure.
* Coupled_Insert(position, new_token_ids, model, kv_cache, kv_mirror):
* Calculates the new K/V tensor slices for `new_token_ids` (using `model` and the left context derived from `kv_cache` up to `position`, ensuring correct RoPE application based on the final `position`).
* Inserts these new K/V slices into each layer of `kv_cache` at the current effective index `position`.
* Simultaneously, inserts the corresponding new token metadata for `new_token_ids` into `kv_mirror` starting at `position`, handling necessary index shifts within the mirror structure.
* Returns (updated_kv_cache, updated_kv_mirror)` or signals failure.
* Coupled_Append(token_id, model, kv_cache, kv_mirror):
* Calculates the K/V tensor slices for the single `token_id` using the entire `kv_cache` as context.
* Appends the new K/V slices to the end of each layer's tensors in `kv_cache`.
* Simultaneously, appends the corresponding new token metadata to the end of `kv_mirror`.
* Returns (updated_kv_cache, updated_kv_mirror)` or signals failure.
```

**6. Research Questions Focused on Coupling:**

```
* Implementation Strategy: What is the best way to implement these `Coupled_Delete`, `Coupled_Insert`, and `Coupled_Append` operations in Python/PyTorch? Should they be methods within a dedicated `StateManager` class that holds references to both the cache and mirror? Or standalone functions?
* Tensor + Mirror Primitives: What are the underlying PyTorch tensor operations (e.g., `torch.cat`, slicing, `index_select`) and `KVMirror` methods (e.g., `prune`, `add`, potentially new ones like `insert_at`) needed inside these coupled operations?
* Index Synchronization: How do the coupled operations ensure that the index used for tensor
```



slicing/concatenation *\*exactly\** corresponds to the logical position being modified in the ``KVMirror``, especially accounting for shifts caused by previous operations within the same Step 4 tick?

- \* **Error Handling:** How should errors within a coupled operation (e.g., K/V calculation fails, tensor dimension mismatch, mirror update fails) be handled to ensure neither the cache nor the mirror is left in a partially updated state for that specific operation? How is failure propagated back to the main Step 4 loop to trigger an abort for the whole tick?
- \* **KVMirror Interface:** Does the ``KVMirror`` class need new methods (e.g., ``insert_tokens_at(position, token_list)``) to robustly support the insertion logic required by ``Coupled_Insert`` and the ``replace_pair`` action? How would such a method handle internal index shifting?
- \* **RoPE Integration:** Confirm the point at which K/V vectors (and thus RoPE) should be calculated within ``Coupled_Insert`` and ``Coupled_Append``. It must use the *\*final destination position\** of the token after all index shifts are accounted for. How is this final position determined reliably just before the K/V calculation?

**7. Desired Output:**

Detailed, annotated Python-like pseudocode for Step 4, *\*explicitly showing calls to the conceptual ``Coupled_Delete``, ``Coupled_Insert``, and ``Coupled_Append`` functions\**. The pseudocode should illustrate the overall orchestration, including:

- \* Processing the ``action_list`` (sorting ``replace_pair`` actions descending, handling ``add`` actions last).
- \* Showing how the ``kv_cache`` and ``kv_mirror`` state are passed between and updated by the coupled operations.
- \* Indicating how errors might be caught and handled to ensure tick-level atomicity.

This revised request emphasizes the non-negotiable lock-step synchronization and asks for an algorithm built around these coupled operations.

Below is a **lock-step, “two-bladed-knife” design** for Step 4.

Every *logical* edit is executed by a **Coupled Operation** that simultaneously touches the tensor cache **and** the metadata mirror; if either side fails the whole operation is rolled back. The top-level routine walks the ``action_list`` in **strict descending position order**, guaranteeing that each subsequent index reference remains valid.

## 0. High-level structure

text

```
StateManager (holds handles to cache + mirror)
• coupled_delete(pos)
• coupled_insert(pos, new_ids)
• coupled_append(token_id)
(each returns True/False; no partial success)
```



Step4\_tick(actions, model, kv\_patcher) ← main orchestration

All tensor slicing and ``KVMirror`` mutations live **inside** the three coupled helpers; the orchestration code never manipulates one without the other.

# 1. `StateManager` skeleton (annotated pseudocode)

python

```
class StateManager:
    def __init__(self, kv_cache: Tuple[Tuple[Tensor, Tensor]], kv_mirror):
        self.cache = kv_cache # reference to the live cache
        self.mirror = kv_mirror # reference to the live KVMirror

    # ----- low-level cache splice helpers -----
    @staticmethod
    def _delete_row(t: Tensor, idx: int) -> Tensor:
        # t shape [B, H, S, D]; remove slice S==idx
        return torch.cat([t[:, :, :idx, :], t[:, :, idx+1:, :]], dim=2)

    @staticmethod
    def _insert_block(t: Tensor, block: Tensor, idx: int) -> Tensor:
        # block shape [B, H, L_new, D]
        return torch.cat([t[:, :, :idx, :], block, t[:, :, idx:, :]], dim=2)

    # -----

    def coupled_delete(self, pos: int) -> bool:
        """
        Remove token at `pos` from *both* cache and mirror.
        Returns False if either side fails; no partial mutation persists.
        """
        try:
            # 1. delete from every layer in-place
            for i, (K, V) in enumerate(self.cache):
                self.cache[i] = (
                    self._delete_row(K, pos),
                    self._delete_row(V, pos)
                )
            # 2. delete from mirror (must shift internal indices)
            self.mirror.remove_token(pos)
            return True
        except Exception as e:
            log.error(f"Delete failure @{pos}: {e}")
            return False

    def coupled_insert(self, pos: int, new_ids: List[int],
                      model, kv_patcher) -> bool:
        """
        Replace zero tokens (pure insert) at `pos` with `new_ids`
        – used after we have already deleted the obsolete rows.
        """
        try:
            # a) build left-context for RoPE
            left_len = pos
            past_left = [
                (K[:, :, :left_len, :], V[:, :, :left_len, :])
                for (K, V) in self.cache
            ]
            new_kv_layers = kv_patcher.compute_kv(model, past_left, new_ids)
            # b) splice into every layer
            for i, (K, V) in enumerate(self.cache):
                new_K, new_V = new_kv_layers[i] # seq_dim == len(new_ids)
                self.cache[i] = (
                    self._insert_block(K, new_K, pos),
                    self._insert_block(V, new_V, pos)
                )
            # c) mirror insert
            self.mirror.insert_tokens_at(pos, new_ids)
            return True
        except Exception as e:
            log.error(f"Insert failure @{pos}: {e}")
            return False

    def coupled_append(self, token_id: int, model, kv_patcher) -> bool:
        try:
```

```

seq_len = self.cache[0][0].size(2)
new_layers = kv_patcher.compute_kv(model, self.cache, [token_id])
for i, (K, V) in enumerate(self.cache):
    new_K, new_V = new_layers[i] # seq_dim == 1
    self.cache[i] = (torch.cat([K, new_K], dim=2),
                    torch.cat([V, new_V], dim=2))
self.mirror.append_token(token_id)
return True
except Exception as e:
    log.error(f"Append failure: {e}")
    return False

```

## Mirror API additions

- ``remove_token(idx)`` - existing.
- ``insert_tokens_at(idx, ids: List[int])`` - *new*: shifts subsequent indices, stores brightness defaults, etc.
- ``append_token(id)`` - existing or trivial.

## 2. Tick-level orchestration

python

```

def step4_tick(action_list, model, kv_cache, kv_mirror, kv_patcher) -> bool:
    """
    Returns True on success, False on total rollback.
    """
    mgr = StateManager(kv_cache, kv_mirror)

    # ---- split & sort actions ----
    repl = [a for a in action_list if a['action'] == 'replace_pair']
    app = [a for a in action_list if a['action'] == 'add']
    repl.sort(key=lambda a: a['original_pos1'], reverse=True) # high-low

    # ---- single pass, strictly descending positions ----
    for act in repl:
        pos1, pos2 = act['original_pos1'], act['original_pos2']
        ids = act['new_token_ids']

        # 1. delete pos2 then pos1 (still coupled)
        if not mgr.coupled_delete(pos2): return False
        if not mgr.coupled_delete(pos1): return False

        # 2. insert replacement block at pos1
        if not mgr.coupled_insert(pos1, ids, model, kv_patcher):
            return False

    # ---- tail appends in arrival order ----
    for act in app:
        if not mgr.coupled_append(act['token_id'], model, kv_patcher):
            return False

    return True # all blades cut together

```

*If any helper returns `False`, Step 4 aborts the tick.*

Typical abort logic:

python

```

success = step4_tick(...)
if not success:

```

```
kv_cache.restore(snapshot_before_step4)
kv_mirror.restore(snapshot_before_step4)
```

(A cheap snapshot is just `copy.deepcopy` of the mirror plus the tuple of tensors or a `DynamicCache.to_legacy()` clone.)

### 3. Index integrity rationale

- Processing **highest** → **lowest** ensures that when we call `coupled_delete(pos2)` and `coupled_delete(pos1)` the indices are still valid; later, lower-numbered edits haven't shifted yet.
- Each coupled operation mutates both structures **inside the same** `try` **block**. Any exception—dimension mismatch, mirror overflow—rolls back that edit immediately.

### 4. Error-handling & atomicity notes

Scope	Guarantee	Mechanism
<b>Per-operation</b>	K/V tensors and mirror mutate together or not at all	<code>try/except</code> around both halves
<b>Per-tick</b>	Either every action succeeds or previous state is restored	Shallow snapshots of cache+mirror taken before first edit (GPU-cheap if tensors share storage)
<b>Concurrency</b>	Reader threads never see partial state	<ul style="list-style-type: none"> <li>• Run Step 4 under a global “context-mutator” lock <b>or</b></li> <li>• Swap-pointer strategy: operate on <i>copies</i>, then <code>atomic_swap(refs)</code></li> </ul>

### 5. RoPE correctness

`kv_patcher.compute_kv(model, context, ids)` feeds **exactly** `len(context)` tokens as `past_key_values`. Hence each new token's rotary angle equals its final 0-based index in the combined sequence:

```
ini

final_position = len(context) # before insertion for insert; old_len for append
```

Because we compute K/V **right before** insertion/append—after all higher deletions are done—that `context` length is already the *true* final index.

### 6. What to implement next

1. **Mirror API extension** (`insert_tokens_at`) with internal brightness init & shift.
2. **Unit tests** that feed the `.pt` capture you supplied:
  - Run tick → regenerate cache → decode → ensure text matches ground-truth IDs.
3. **Lightweight snapshot/rollback utility** (e.g., `clone_cache()` that alias-copies CUDA tensors).

4. Wrap Step 4 in a context-manager: `with step4_transaction(kv_cache, kv_mirror): ...` to automate rollback.`

## Recap

Step 4 is not “edit the cache then fix the mirror.”

It is a **single surgical motion** where each cut slices **both** data structures in one call.

The design above operationalises that philosophy: every mutation is wrapped in a *coupled* helper, and the tick driver aborts on the first blood-mismatched stroke.



We can't take a backup. This is the KV cache. It could be 10GB and sitting on VRAM. We can't move it to system memory every token in case we screw up. If we fail, we must take our current understanding of the cache, our mirror, and we must reprocess the whole KV from scratch. It's the only way to guarantee consistency between the two states. The goal is not to fuck up.

## Revision — “No Safety-Net” Step 4

“The goal is **not** to screw up.”

**Premise:** a 10 GB VRAM cache cannot be snapshotted each tick.

**Consequence:** Step 4 must operate *in-place* and accept that, if an exception occurs, the only recovery path is **full cache re-materialisation from the mirror**.

**Design target:** make failure astronomically unlikely by combining exhaustive **pre-flight checks** with **lock-step primitives** that either complete or raise before the cache is modified beyond trivial roll-back.

## 1 State primitives (in-place, zero-copy where possible)

python

```
class StateManager:
    """
    Owns:
    • self.cache — tuple[ (K, V) ] on GPU
    • self.mirror — Python / CPU structure
    No deep copies. Every method must be:
    (a) pre-flight validate, then
    (b) perform tensor + mirror edits, raising on first anomaly.
    """

    # ----- #
    # Internals — 0(1) checks, zero data movement
    # ----- #
    def _assert_align(self):
        seq_len_cache = self.cache[0][0].size(2)
        seq_len_mirror = self.mirror.length()
        assert seq_len_cache == seq_len_mirror, \
            f"KV/Mirror length mismatch {seq_len_cache} vs {seq_len_mirror}"

    def _check_row(self, idx: int):
        L = self.cache[0][0].size(2)
        assert 0 <= idx < L, f"Row {idx} out of range 0...{L-1}"

    # ----- #
```

```
# Coupled operations – all raise on *any* discrepancy
# ----- #

def delete(self, idx: int):
    self._assert_align()
    self._check_row(idx)

    for i, (K, V) in enumerate(self.cache):
        # drop slice idx with two cat ops – creates one new tensor per layer
        self.cache[i] = (
            torch.cat((K[:, :, :idx, :], K[:, :, idx+1:, :]), 2),
            torch.cat((V[:, :, :idx, :], V[:, :, idx+1:, :]), 2),
        )

    self.mirror.remove_token(idx)
    self._assert_align() # cheap post-condition

def insert(self, idx: int, new_ids: list[int], *, model, kvpatch):
    self._assert_align()
    self._check_row(idx) # insertion allowed at L as well
    left_len = idx # context length before insert

    # 1 Generate K/V block (small, ≤ len(new_ids) rows)
    past_left = [ (K[:, :, :left_len, :], V[:, :, :left_len, :])
                  for (K, V) in self.cache ]
    new_block_per_layer = kvpatch.compute_kv(model, past_left, new_ids)

    # 2 Splice tensors
    for i, (K, V) in enumerate(self.cache):
        nK, nV = new_block_per_layer[i]
        self.cache[i] = (
            torch.cat((K[:, :, :idx, :], nK, K[:, :, idx:, :]), 2),
            torch.cat((V[:, :, :idx, :], nV, V[:, :, idx:, :]), 2),
        )

    # 3 Mirror splice
    self.mirror.insert_tokens_at(idx, new_ids)
    self._assert_align()

def append(self, token_id: int, *, model, kvpatch):
    idx = self.cache[0][0].size(2) # current length
    self.insert(idx, [token_id], model=model, kvpatch=kvpatch)
```

If any assertion or tensor operation fails, Python raises → Step 4 aborts immediately.

## 2 Top-level tick (descending order, no backups)

python

```
def step4_tick(actions, model, kv_cache, kv_mirror, kvpatch) -> bool:
    mgr = StateManager(kv_cache, kv_mirror)

    repl = [a for a in actions if a['action'] == 'replace_pair']
    app = [a for a in actions if a['action'] == 'add']
    repl.sort(key=lambda a: a['original_pos1'], reverse=True)

    try:
        for a in repl:
            pos1, pos2 = a['original_pos1'], a['original_pos2']
            new_ids = a['new_token_ids']

            mgr.delete(pos2) # higher index first
            mgr.delete(pos1)
            mgr.insert(pos1, new_ids, model=model, kvpatch=kvpatch)

        for a in app: # end-of-context growth
            mgr.append(a['token_id'], model=model, kvpatch=kvpatch)

        return True # success path
    except Exception as e:
```



```
log.critical(f"Step 4 failed: {e}. Initiating full KV rebuild.")
rebuild_from_mirror(model, kv_cache, kv_mirror) # see below
return False
```

### 3 Recovery routine (rare, but mandatory)

python

```
def rebuild_from_mirror(model, kv_cache, kv_mirror):
    """
    *Deterministically* regenerate the entire KV cache from mirror's token list.
    This is O(sequence_len) forward pass – expensive, but happens <0.01 %.
    """
    tokens = kv_mirror.token_id_list() # List[int] length L
    input_ids = torch.tensor(tokens, device='cuda').unsqueeze(0)
    with torch.no_grad():
        _ = model(input_ids, use_cache=True) # populates model.cache
    new_cache = model.get_cache_as_legacy() # or model.cache if legacy
    # Replace in-place; all readers hold the same tuple reference obj
    for i in range(len(kv_cache)):
        kv_cache[i] = new_cache[i]
```

Because `kv\_cache` is mutated in-place layer-by-layer, external references already point to the refreshed tensors.

### 4 Why this meets the constraint

Requirement	Design response
No full snapshot each tick	Only small temporary tensors (the replacement block) are materialised. No `torch.clone()` of 10 GB buffers.
Knife-with-two-blades	`delete`, `insert`, `append` live inside `StateManager`; they assert, mutate cache and mirror, then re-assert. A failure in either side → Python exception → outer `try` aborts.
Recovery without backup	On exception we rebuild KV from the authoritative mirror. Costly but rare; avoids carrying a spare 10 GB shadow cache.
RoPE correctness	`kvpatch.compute_kv` receives <i>current</i> left-context length (`idx` or `seq_len`) at call-time, so rotary phase always matches final position.
Index safety	Processing actions in descending `pos1` order ensures that previously unseen indices do not shift before they are referenced.

### Final note

With exhaustive `assert` guards, well-tested `KVMirror` operations, and action-list generation that already avoids contradictions, Step 4 becomes “neurotically safe” while living entirely inside VRAM. If it ever fails, that indicates corruption serious enough to justify the one-off cost of `rebuild\_from\_mirror`, restoring cache↔mirror coherence before the next token is even considered.



Gemini: Performance of torch.cat: While logically correct, repeatedly using torch.cat for deletions and insertions in large tensors can be computationally expensive due to memory reallocations. For

extreme performance needs, exploring more advanced sparse/masked update techniques might be necessary later, but `torch.cat` is the standard and correct way to start.

Agreed — `torch.cat` is the canonical, correctness-first move, but we should already sketch the escape hatches for when 4-5 GiB layers make reallocations dominate the time-to-token. Below is a quick performance roadmap:

Stage	Complexity	When to graduate from <code>torch.cat</code>	Practical swap-in
Baseline	<code>torch.cat</code> for every splice. Each call materialises a fresh contiguous buffer for the layer ( $\approx O(H \cdot D \cdot S)$ copy).	Until latency profiling shows Step 4 > ~5-10 ms.	What we have now. Easiest to reason about; bullet-proof.
Batch-splicing	Still <code>torch.cat</code> , but build one <i>mask</i> per layer: drop all rows to be deleted in a single re-copy, then insert all new blocks in one concatenation.	When multiple replacements per tick make “double delete + insert” triple the copies.	Pre-sort deletions, slice once; accumulate replacement blocks in a list; final <code>torch.cat([left, *insert_blocks, right])</code> .
Boolean-gather	<code>torch.index_select</code> / <code>masked_select</code> gives a copy without reallocating row-by-row. You still pay one whole-tensor copy, but save the extra intermediate writes.	When VRAM BW > compute and you still need contiguous layout.	Build an <code>idx_keep</code> vector, <code>torch.index_select(dim=2)</code> .
Indirection buffer	Stop copying: keep one <i>flat grow-only</i> KV buffer and a <code>logical→physical</code> index map ( <code>list[int]</code> ). Deletions mutate the map; insertions append new rows. Attention lookup becomes two-step ( <code>K_phys = K[:, :, idx_map, :]</code> ).	When Step 4’s memcpy shows up in nvprof even with batch-splicing. Trades memory for pointer chasing.	Maintain <code>idx_map</code> on CPU; use <code>torch.take_along_dim</code> to gather K/V at runtime. Works well with FlashAttention-2 which already does gathers.
Sliding-window / ring-buffer cache	Each layer owns a pre-allocated ring; a write-cursor advances, wrapping modulo <code>capacity</code> . No copies; just mod-arithmetic. Mirror stores cursor offsets.	When context pruning becomes the bottleneck and you can tolerate “holes” or periodic compaction.	Hugging Face’s <code>DynamicCache</code> is halfway there. Implement our own or extend theirs.

## Immediate, low-risk optimisation: batch-splicing

python

```
# Pseudocode for one layer, all deletions pre-sorted descending
keep_mask = torch.ones(seq_len, dtype=torch.bool, device=K.device)
keep_mask[del_indices] = False # mark rows to drop
K_new = K[:, :, keep_mask, :] # 1 gather
# build insertion list already as GPU tensors
chunks = [K_new[:, :, :insert_point, :],
           new_block1, new_block2, # pre-computed K/V
           K_new[:, :, insert_point:, :]]
K_final = torch.cat(chunks, dim=2) # 1 cat
```

Net: one gather + one cat versus 2-3 cats per replacement.

## Long-term: indirection map

Think of the KV cache as an **append-only ledger**; Step 4 merely updates a Python-side map:

text

logical\_pos → physical\_row

- **Delete:** pop from map ( $O(1)$ ), no tensor touch.
  - **Insert:** append new rows; push their indices into map ( $O(\text{len}(\text{new}))$ ).
  - **Attention:** gather rows on-the-fly (`torch.take_along_dim`).
- Downside: every forward pass now does a gather; upside: Step 4 becomes almost free.

## Recommendation for now

1. **Instrument first** – wrap `torch.cuda.Event` timers around Step 4; see if we're actually spending > a few ms per tick.
2. If yes, implement **batch-splicing** (one gather + one cat per layer). It keeps the "two-bladed knife" semantics intact and is a direct drop-in.
3. Leave a TODO for indirection once we can profile against real ~8 k-token windows.

That keeps us on the **"do no harm"** path while leaving room to scale when Halo Weave starts running with multi-thousand-token contexts.



DynamicCache Handling: The document acknowledges DynamicCache but the pseudocode primarily targets the tuple structure. If the model returns DynamicCache, the low-level tensor operations inside StateManager (`_delete_row`, `_insert_block`, `torch.cat` in `append`) would need replacement with the appropriate DynamicCache API calls (e.g., `update`, `append`, potentially manual tensor extraction/manipulation/reconstruction if direct methods aren't sufficient). The conversion strategy mentioned (`to_legacy`, `update`, `from_legacy`) is a viable workaround.

Below is a **bolt-on adaptation layer** that lets the lock-step `StateManager` handle **either**

(1) the classic *tuple-of-(K,V)* **or**

(2) Hugging Face's new `DynamicCache` / `Cache` objects (used by 4.40+ when you pass `use_cache=True`).

Printed using [ChatGPT to PDF](#), powered by PDFCrowd [HTML to PDF API](#).

The goal is to keep **one source-of-truth algorithm** while swapping out the low-level splice primitives.

## 1 Detecting cache type at runtime

```
python

def is_dynamic(cache) -> bool:
    # Works for HF ≥ 4.40; adjust if they rename.
    from transformers.cache_utils import Cache
    return isinstance(cache, Cache)
```

## 2 Common shim API

```
python

class CacheShim:
    """
    Thin façade exposing:
        • len(self)           -> #layers
        • get(layer)          -> (K, V)   contiguous tensors
        • set(layer, newK, newV) -> in-place replace
        • seq_len()           -> current sequence length
        • gather(keep_idx)     -> drop rows via boolean / long mask
        • concat(new_block, idx) -> splice insert block at idx
        • append(new_block)    -> grow by 1..N rows
    Internally routes to tuple-ops or DynamicCache ops.
    """

    def __init__(self, raw):
        self.raw = raw
        self.is_dyn = is_dynamic(raw)

    # ----- helpers for DynamicCache -----
    def _get_dyn(self, layer):
        # keys/values are tensors inside Cache; API differs by model.
        K = self.raw.keys[layer]          # shape [B,H,S,D]
        V = self.raw.values[layer]
        return K, V

    def _set_dyn(self, layer, K, V):
        self.raw.keys[layer] = K
        self.raw.values[layer] = V

    # ----- public -----
    def seq_len(self):
        K0 = self.get(0)[0]
        return K0.size(2)

    def get(self, layer):
        if self.is_dyn:
            return self._get_dyn(layer)
        else:
            return self.raw[layer]

    def set(self, layer, K, V):
        if self.is_dyn:
            self._set_dyn(layer, K, V)
        else:
            self.raw[layer] = (K, V)

    # ---- tensor row removal in bulk (boolean mask) ----
    def gather(self, keep_mask: torch.Tensor):
        assert keep_mask.dtype == torch.bool and keep_mask.dim() == 1
```

```

for i in range(len(self)):
    K, V = self.get(i)
    self.set(i,
              K[:, :, keep_mask, :],
              V[:, :, keep_mask, :])

# ---- splice insert block (contiguous) ----
def concat(self, block_per_layer, insert_idx: int):
    for i, (blkK, blkV) in enumerate(block_per_layer):
        K, V = self.get(i)
        newK = torch.cat((K[:, :, :, insert_idx, :], blkK, K[:, :, :, insert_idx, :]), 2)
        newV = torch.cat((V[:, :, :, insert_idx, :], blkV, V[:, :, :, insert_idx, :]), 2)
        self.set(i, newK, newV)

def append(self, block_per_layer):
    for i, (blkK, blkV) in enumerate(block_per_layer):
        K, V = self.get(i)
        self.set(i,
                  torch.cat((K, blkK), 2),
                  torch.cat((V, blkV), 2))

# ----- dunder helpers -----
def __len__(self):
    return len(self.raw)

```

If Hugging Face extends `Cache` with convenient `index\_select` / `select\_seq` / `append\_block` helpers you can wire them in here; the Step 4 logic above stays unchanged.

### 3 Drop-in replacement inside `StateManager`

python

```

class StateManager:
    def __init__(self, kv_cache, kv_mirror):
        self.cache = CacheShim(kv_cache) # <-- wrap once
        self.mirror = kv_mirror

```

All subsequent calls (`seq\_len()`, `get`, `set`, `gather`, `concat`, `append`) now work for **both** cache layouts.

### 4 How the earlier primitives map to the shim

Earlier op	Tuple-only code	Shim call (works for both)
delete row	two `torch.cat` calls per layer	one boolean mask build → `cache.gather(mask)`
insert block	`_insert_block()` per layer	`cache.concat(block_per_layer, idx)`
append	`torch.cat([K, new_K], dim=2)`	`cache.append(block_per_layer)`

(The boolean **mask gather** is a mild optimisation over row-by-row `torch.cat`: one allocation per layer.)

### 5 Conversion fallback

If some exotic model ships a `Cache` without public tensors:

python

```

if not shim.supports_direct_ops():
    # slow - but universal

```

```
legacy = shim.raw.to_legacy_cache()
shim = CacheShim(legacy)          # now tuple
...                               # perform Step 4 edits
new_dyn = Cache.from_legacy_cache(legacy)
kv_cache = new_dyn                # swap back
```

``to_legacy_cache` / `from_legacy_cache`` are already in HF  $\geq 4.40$ ; they copy once, *not* every tick.

## 6 Performance notes

- The **mask-gather** path is already friendlier than repeated ``torch.cat``.
- For ``DynamicCache``, HF often stores keys/values in a **flat grow-only buffer** plus an index tensor; deleting via mask is the intended fast path.
- ``FlashAttention-2`` kernels can consume *gathered* sequences efficiently, so indirection is future-proof.

### TL;DR

Add a ``CacheShim`` layer:

```
text

tuple-of-tensors ↔ CacheShim ↔ DynamicCache
```

The knife blades (``delete``, ``insert``, ``append``) swing through ``CacheShim``, so Step 4 stays identical while the implementation beneath swaps from ``torch.cat`` on tuples to ``boolean-gather`` or native APIs on ``DynamicCache``. You get correctness today and an easy path to high-performance tomorrow without rewriting the orchestration logic.



Bryn: We can't use existing KVMirror methods. I don't trust them. They are part of a previous version of the project that spiraled out of control. I don't know if there's anything valuable in there or not, but I didn't properly control their creation and therefore they scare me. That's why I've had such an iron fist over this process. We tried before and I lost control, nothing worked or lined up with anything else. It was endless debugging.

Gemini: Understood completely. Your caution is warranted, especially given past experiences. If the existing ``KVMirror`` methods are suspect or their internal logic is opaque/unverified for the precise index shifting needed, relying on them is risky.

Therefore, the **best approach** is for the ``StateManager`` (or whatever implements the Step 4 logic) to perform **direct, low-level manipulation of the ``KVMirror``'s internal state** (``_pos`` dictionary, ``_registry`` dictionary) **within the same atomic operation** that modifies the cache tensors.

**Revised Step 4 Implementation Strategy:**

1. **Coupled Operations Perform Direct Mirror Updates:**

- \* The conceptual functions ``Coupled_Delete``, ``Coupled_Insert``, ``Coupled_Append`` (likely methods



within `StateManager`)` will not call high-level `KVMirror`` methods like `remove_token`` or `insert_tokens_at``.

- \* Instead, within the `try/except`` block that guarantees atomicity for *that specific operation*, they will:
  - \* Perform the necessary tensor manipulations (using the `CacheShim``).
  - \* **\*\*Directly modify `self.mirror._pos`` and `self.mirror._registry``\*\*** (acquiring `self.mirror._lock`` first) to reflect the change. This includes:
    - \* **\*\*For Deletion at `pos``:** Remove `pos`` from `_pos``, update the corresponding token's state in `_registry``, and **\*\*manually shift all keys greater than `pos`` down by one in `_pos``\*\***, also updating the `.position`` attribute for the affected tokens in `_registry``.
    - \* **\*\*For Insertion at `pos``:** **\*\*Manually shift all keys greater than or equal to `pos`` up by `len(new_token_ids)`` in `_pos``\*\***, update corresponding `.position`` attributes in `_registry``, then insert the new `instance_id``(s) at the correct starting position(s) in `_pos``, and add the new `ContextToken`` object(s) to `_registry``.
    - \* **\*\*For Append:** Get the next available position index, add the new `instance_id`` to `_pos`` at that index, and add the new `ContextToken`` to `_registry``.

Bryn: We cannot shift all these keys. That's thousands of potential operations every single token, no. It's my understanding that the KV cache itself doesn't have index numbers to update? Then neither will our mirror. I propose two utilities, a ledger that stores information for every token in chronological order regardless of it's status, and a simple KV dictionary that maps the exact keys in the exact same positions as the real KV cache, but instead of their value being tensors, the value is the token ID in the ledger so that we can do a lookup whenever we want it. Every action that happens to the KV cache happens to our KV dictionary in exactly the same way using the same functions.

Below is a **“no-shift” mirror design** that matches how the KV cache itself works:

*the cache never rewrites its past rows, it only drops or appends.*

We do the same by splitting the mirror into:

Piece	Role	Mutability	Size
<b>Ledger</b> ( <code>ledger: List[ContextToken`)</code> )	Append-only record of <b>every token ever seen in chronological order</b> (includes tombstones).	Only <b>append</b> new rows. Never reorder or delete.	Unbounded but grows $\leq$ total generated tokens. CPU-side list/array.
<b>LiveMap</b> ( <code>pos2ledger: List[int`)</code> )	Exact 1-for-1 positional map to the current KV cache: <code>pos2ledger[cache_pos] = ledger_idx`</code> .	Edited <i>with the same splice ops</i> we perform on the KV cache. No per-row renumbering.	At most current sequence length ( $N \lesssim 8\text{ k}$ ).

## Why this solves your concerns

- **Zero shifting inside the ledger** – we never touch historical rows.
- **LiveMap mutations cost  $\leq$  the tensor splice** – deleting a row is a single `list.pop(pos)``; inserting is `list[pos:pos] = [new_idx1, ...]``. That's the same  $O(N)$  we already pay for the GPU slice, but it's tiny on the CPU ( $\sim$  microseconds).
- **Coupling is trivial** – the *identical* splice (delete, insert, append) is applied to:
  1. the key tensor of every layer (via `CacheShim``), **and**

2. the `pos2ledger` list.

Both take the **same** `pos` and `length` arguments, so the mirror can't drift.

# 1 Data structures

python

```
@dataclass
class ContextToken:
    token_id: int
    brightness: float = 255.0
    # extra slots: logits, time_seen, etc.

class Mirror:
    def __init__(self):
        self.ledger: list[ContextToken] = []
        self.pos2ledger: list[int] = [] # mirrors cache rows
```

`ledger_idx == len(ledger)` is your "instance\_id".

## 2 Coupled operations (all in-place, no index shifts inside ledger)

python

```
class StateManager:
    def __init__(self, cache, mirror: Mirror, kvpatch, model):
        self.cache = CacheShim(cache)
        self.mirror = mirror
        self.kvpatch = kvpatch
        self.model = model

    # ----- helpers -----
    def _append_to_ledger(self, token_ids: list[int]) -> list[int]:
        start = len(self.mirror.ledger)
        for tid in token_ids:
            self.mirror.ledger.append(ContextToken(token_id=tid))
        return list(range(start, start + len(token_ids))) # ledger indices

    # ----- coupled delete -----
    def delete_at(self, pos: int):
        # 1. splice GPU rows (same as before)
        keep = torch.ones(self.cache.seq_len(), dtype=torch.bool, device='cuda')
        keep[pos] = False
        self.cache.gather(keep)
        # 2. splice map
        self.mirror.pos2ledger.pop(pos)

    # ----- coupled insert -----
    def insert_at(self, pos: int, token_ids: list[int]):
        # 1. compute K/V block
        left_len = pos
        past_left = [ (K[:, :, :left_len, :], V[:, :, :left_len, :])
                      for (K, V) in self.cache ]
        new_block = self.kvpatch.compute_kv(self.model, past_left, token_ids)
        self.cache.concat(new_block, pos) # splice into tensors
        # 2. ledger + map
        new_idx = self._append_to_ledger(token_ids)
        self.mirror.pos2ledger[pos:pos] = new_idx # same splice op

    # ----- coupled append -----
    def append(self, token_id: int):
        pos = self.cache.seq_len()
        self.insert_at(pos, [token_id]) # reuse insert_at
```

No pre-existing `KVMirror` methods are used; the manager owns the **entire mirror mutation**.

---

## 3 Orchestration (identical to previous tick loop)

```
python

def step4(actions, cache, mirror, kvpatch, model):
    mgr = StateManager(cache, mirror, kvpatch, model)

    repl = [a for a in actions if a['action'] == 'replace_pair']
    repl.sort(key=lambda a: a['original_pos1'], reverse=True)
    app = [a for a in actions if a['action'] == 'add']

    try:
        for a in repl:
            mgr.delete_at(a['original_pos2'])
            mgr.delete_at(a['original_pos1'])
            mgr.insert_at(a['original_pos1'], a['new_token_ids'])

        for a in app:
            mgr.append(a['token_id'])
        return True
    except Exception as e:
        log.critical("Step 4 failed, fully regenerating KV cache.")
        rebuild_from_mirror(model, cache, mirror) # as earlier
        return False
```

(`rebuild_from_mirror` walks `mirror.pos2ledger` to get the live token IDs.)

---

## 4 Fast look-ups & brightness updates

Need to query or dim a token by **position**?

```
python

def dim_brightness(pos, factor):
    led_idx = mirror.pos2ledger[pos]
    mirror.ledger[led_idx].brightness *= factor
```

$O(1)$ : one list lookup, one struct update, zero shifts.

---

## 5 Memory & performance

- **Ledger**: a Python list of small objects; even one million tokens  $\approx$  100 MB.
  - **LiveMap**: `List[int]` of current positions;  $\leq 8\text{ k} \approx 32\text{ KB}$ .
  - CPU-side splices (`pop`, `List[pos:pos] = ...`) are negligible compared with GPU `torch.cat`.
- 

### Bottom line

By divorcing *history* (ledger) from *live order* (pos2ledger map) we:

1. **Never mutate old rows** — history is sacrosanct.
2. **Update the map with the exact same splice we give the tensors** — your “knife with two blades.”

3. **Avoid  $O(N^2)$  shifts** — each tick costs  $\leq$  one list splice proportional to the number of tokens inserted/deleted, which is already dwarfed by the unavoidable tensor copy.

This gives you a fresh, trustworthy mirror implementation—minimal, explicit, and fully under Step 4's iron-fisted control—without inheriting any ghosts from the previous KVMirror code.



Avoid  $O(N^2)$  shifts — each tick costs  $\leq$  one list splice proportional to the number of tokens inserted/deleted, which is already dwarfed by the unavoidable tensor copy.

What unavoidable tensor copy?

The “unavoidable copy” is the GPU memory move that happens the moment we splice the key / value tensors.

## Why a copy happens at all

### 1. Contiguous layout.

Every key or value tensor is laid out `[B, H, S, D]` as one contiguous CUDA buffer. PyTorch cannot “punch a hole” in the middle of that buffer or “stretch” it in-place.

### 2. `torch.cat` (or `torch.index_select`, `gather`, etc.) reallocates.

When we do

```
python

new_K = torch.cat((K_left, new_block, K_right), dim=2)
```

PyTorch must allocate a *new* contiguous `[B, H, S', D]` buffer and copy all participating slices (`K_left`, `new_block`, `K_right`) into it.

The original `K` remains untouched until the new tensor takes its place and the old one is freed by the allocator.

### 3. No in-place shrink/expand API today.

Neither the legacy tuple cache nor `DynamicCache` exposes an operation like “delete rows 61–62 without reallocating.” You either:

- allocate a new tensor (copy now), or
- keep the old tensor and introduce an indirection map so attention gathers only the rows you still want (copy later, every forward pass).

## Cost scale

For a single replacement of two rows with one new row:

```
css

bytes_copied ≈ B · H · D · (S - 2 + 1) keys
              + B · H · D · (S - 2 + 1) values
```

With typical TinyLlama sizes ( $B = 1$ ,  $H = 32$ ,  $D = 128$ ,  $S \approx 8$  k) that's ~65 MB per layer — large but still much smaller than the total 10 GB cache distributed across layers. On modern GPUs this is a few micro-seconds of bandwidth.

## Ways to really avoid the copy (future options)

Strategy	What moves instead	Trade-off
<b>Indirection map</b> (gather at runtime)	Gather rows every forward pass, not during Step 4	Extra gather kernel each token generation; slightly lower FlashAttention efficiency
<b>Ring / sliding buffer</b>	Keep one fixed-size buffer and a cursor; never copy	Requires custom cache implementation and attention kernels aware of wrap-around
<b>Memory pools + CUDA graphs</b>	Pre-allocate max-size tensor, reuse, and treat gaps as "masked"	Wastes VRAM capacity proportional to max sequence length

For now, with a contiguous tuple or standard `DynamicCache`, **one tensor-reallocation per splice is simply how PyTorch makes the new shape real**. That's the copy I called "unavoidable." If later profiling shows Step 4's bandwidth dominates, we can promote the indirection-map design and push the cost into the attention gather—which runs anyway every token.