Agentic RL: Token-In, Token-Out Done Right

(()=>{const
s=document.currentScript,o=s?.previousElementSibling?.closest(".html-
embed"),c=s?s.previousElementSibling:null;if(!c||!o)return;let r=!1;const t=
()=>{if(r||!c)return;r=!0,c.querySelectorAll("script").forEach(e=>{if(!
(e.type&&e.type!=="text/javascript"&&e.type!=="module"&&e.type!=="")&&e
{const
n=document.createElement("script");Array.from(e.attributes).forEach(a=>n.se
try{(0,eval)(e.text||"")}catch(n){console.error("HtmlEmbed inline script
error:",n)}}),o.classList.add("html-embed--
loaded")};if("IntersectionObserver"in window){const i=new
IntersectionObserver(e=>{e.forEach(n=>{n.isIntersecting&&!r&&
(i.disconnect(),document.readyState==="loading"?
document.addEventListener("DOMContentLoaded",t,
{once:!0}):requestAnimationFrame(()=>{setTimeout(t,0)}))})},
{rootMargin:"100px",threshold:.01});i.observe(o),setTimeout(()=>{r||
(i.disconnect(),document.readyState==="loading"?
document.addEventListener("DOMContentLoaded",t,
{once:!0}):requestAnimationFrame(()=>{setTimeout(t,0)}))},3e3)}else
document.readyState==="loading"?
document.addEventListener("DOMContentLoaded",t,
{once:!0}):requestAnimationFrame(()=>{setTimeout(t,0)})})();

Never re-tokenize what you decoded: the whole trick to correct multi-turn RL.

AUTHORS

Quentin Gallouédec, Kashif Rasul

AFFILIATION

Hugging Face

PUBLISHED

May 28, 2026

Table of Contents

You're training an LLM with RL. Single-turn looks great: clean curves, sane rewards, things converge. But modern models are enhanced with tools, and that's exactly what you want: to train an *agent*.

So you upgrade your training loop to allow the model to call a tool mid-rollout. You start with an easy task, and the curves get weird. Loss occasionally spikes for no obvious reason. And eventually it fails with a shape mismatch error.

What's almost certainly going on: your rollout loop is silently violating the Token-In, Token-Out (TITO) invariant. You parsed the model's response to detect tool calls, then re-tokenized the updated conversation for the next turn. Usually that round-trip gives back the same tokens. Sometimes it doesn't, and the gradient ends up on a sequence the model never sampled. The code doesn't crash, but the math is silently broken and the gradient signal becomes completely unreliable.

Two ways to fix it.

The first is to abstract the chat template behind a per-model interface. For every family you train on, you hand-code a renderer that knows how to format messages, parse completions, and bridge between turns without re-rendering. It's tricky to get right. The `renderers` library does this. It works, and it covers the major open-weights families today. The cost is structural: every new model needs a new hand-coded renderer, and changes to any template propagate as ongoing maintenance.

The second is to design the training around one rule: never re-encode tokens you've decoded. Follow it, and the tricky edge cases vanish. You're left with a single property to check on the chat template: it must be prefix-preserving for tool messages (we'll explain). Turns out the vast majority of templates in the wild already satisfy it. This is Token-In, Token-Out done right, and that's what this post is about.

Train on the model's own tokens

> `tl;dr` RL updates the model on the exact tokens it sampled, and nothing else. Simple now, load-bearing later.

Reinforcement learning, in one breath: you sample a prompt, the model generates a completion, you score the completion, you backprop the gradient through the model's generated tokens.

Single-turn RL loop.

```
sample
prompt

tokenize
prompt

generate
completion

compute
reward

backprop
on
assistant
tokens
```

One detail matters more than it looks. The gradient is computed on the tokens the model generated. That sounds obvious. What else would you train on? It is obvious. Remember it anyway, because you're going to break it sooner than you think.

Multi-turn doesn't change much. The model is allowed to call a tool mid-rollout: it emits a tool call, something on the outside runs the tool, the result is appended back into the conversation, and the model picks up from there. The rollout is just longer now: a few model turns, a few tool turns, a final answer.

Multi-turn RL loop, with a tool call.

```
sample
prompt

tokenize
prompt

generate
completion

execute
tool and
append
result

generate
completion

compute
reward

backprop
on
assistant
tokens
```

The rule carries over: backprop on the tokens the model produced. Not the tool's response (those didn't come from the policy).

The takeaway is small and very specific: in RL, you optimize on the exact tokens the model produced. Right now it reads like a definition. Later in the post, it's the thing that breaks.

Decoding doesn't undo encoding

> `tl;dr` Tokenization isn't reversible: decode a sequence, re-encode the text, and you can land on different tokens.

Going from messages to tokens is mechanical: a chat template renders the messages into a string, then the tokenizer chops that string into integer IDs.

```
>>> from transformers import AutoTokenizer
>>> tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
>>> messages = [
...     {"role": "user", "content": "What's 2+2?"},
...     {"role": "assistant", "content": "4."}
... ]
>>> tokenizer.apply_chat_template(messages, return_dict=False)
[151644, 8948, 198, 2610, 525, 1207, 16948, 11, 3465, 553, 54364, 14817, 13, 1446, 525, 264, 10950, 17847, 13, 151645, 198, 151644, 872, 198, 3838, 594, 220, 17, 10, 17, 30, 151645, 198, 151644, 77091, 198, 19, 13, 151645, 198]
```

Most of the time you don't think about it. You feed messages, you get tokens, the model does its thing.

Multi-turn is where it starts to matter. When the assistant emits tokens, you don't know whether it's about to call a tool until you look. So you decode the generated IDs back into text, parse out the structure, dispatch the call. The pipeline runs *backwards*, conceptually.

The model can generate a response without calling a tool:

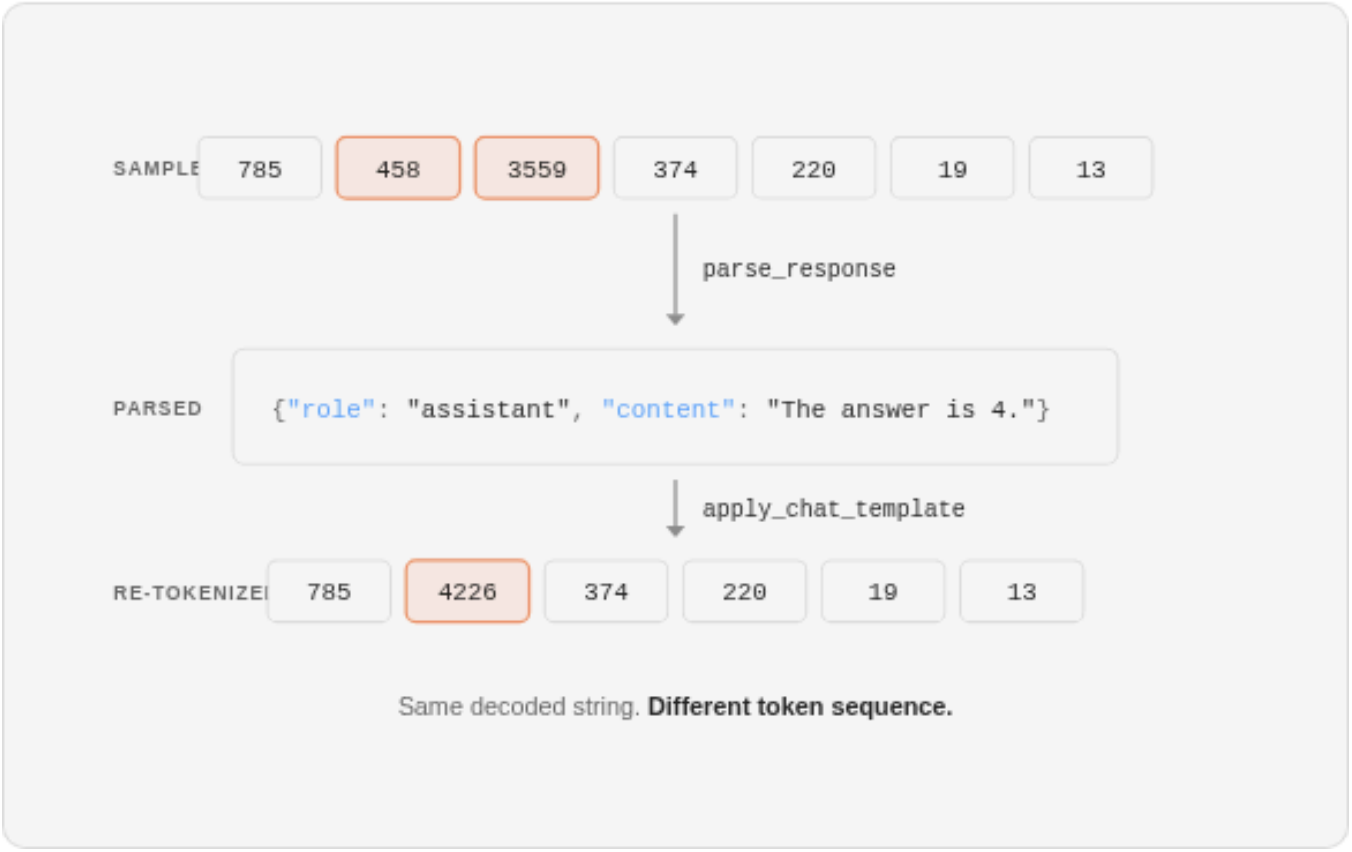
```
>>> output_ids = [19, 13, 151645]  # what the model just generated
>>> tokenizer.parse_response(output_ids)
{"role": "assistant", "content": "4."}
```

or it can call a tool:

```
>>> output_ids = [151657, 198, 4913, 606, 788, 330, 88821, 497, 330, 16370, 788, 5212, 9413, 788, 330, 17, 10, 17, 95642, 151658, 151645]
>>> tokenizer.parse_response(output_ids)
{'role': 'assistant', 'content': '', 'tool_calls': [{'type': 'function', 'function': {'name': 'calculator', 'arguments': {'expr': '2+2'}}}]}
```

Here's the catch. Decoding isn't injective. Multiple distinct token sequences can decode to the same string. Which means: take some tokens, decode them, encode the result back, and you may land on a *different* sequence than the one you started with.

Decode-then-re-encode lands on a different token sequence.



Here, briefly: byte-pair merges aren't stable across token boundaries. Given a string, BPE has one canonical greedy segmentation, but many other valid segmentations exist. Anything you stack on top of that (JSON serialization with negotiable whitespace, argument ordering, boolean casing (`false` vs `False`), how special tokens get re-rendered after a parse) adds more degrees of freedom.

The natural-but-wrong loop

> `tl;dr` Re-rendering the message list every turn drifts the tokens, so you backprop on a sequence the policy never produced.

The natural way to write the loop is the one you'd write on a Friday afternoon. Keep the conversation as a list of messages. Loop over turns. At each turn, render the conversation, generate, parse, append, repeat. When the model finishes, tokenize the whole thing and backprop.

The MITO loop, step by step.

```
sample prompt
while model should generate a
new turn:
   tokenize conversation so far
   generate tokens until model
   stops
   parse the response and
   append
   if there's a tool call:
      execute tool and append
      result
   else:
      stop
compute reward
tokenize the full conversation
compute loss on the tokenized
conversation
backprop on assistant tokens
```

But it's broken in two specific ways.

The first is small but unpleasant. When you tokenize the full conversation at the end, you've lost the per-turn boundaries. The trainer no longer knows which tokens came from the assistant and which came from the tool, and you only want to compute loss on the assistant turns. So you have to *recover* that mapping after the fact: walk the rendered string, find the role markers, figure out which token indices fall inside each assistant turn. Doable, but every chat template does this differently, and you end up writing a small parser per model family. The `renderers` library exists in part to do exactly this. It attaches a `message_indices` array to the rendered ids so each token knows which message it belongs to.

The second is much worse. You broke the rule sooner than you thought: re-tokenizing the conversation at the end can give you a slightly different token sequence than the one the model sampled. Same string, different integer ids. We saw why in the previous section: encode and decode aren't inverses. Consequently, you backprop on these new ids. The gradient targets tokens the policy never produced. The rule from before breaks.

That's the loop everyone writes first. And it's why this post exists.

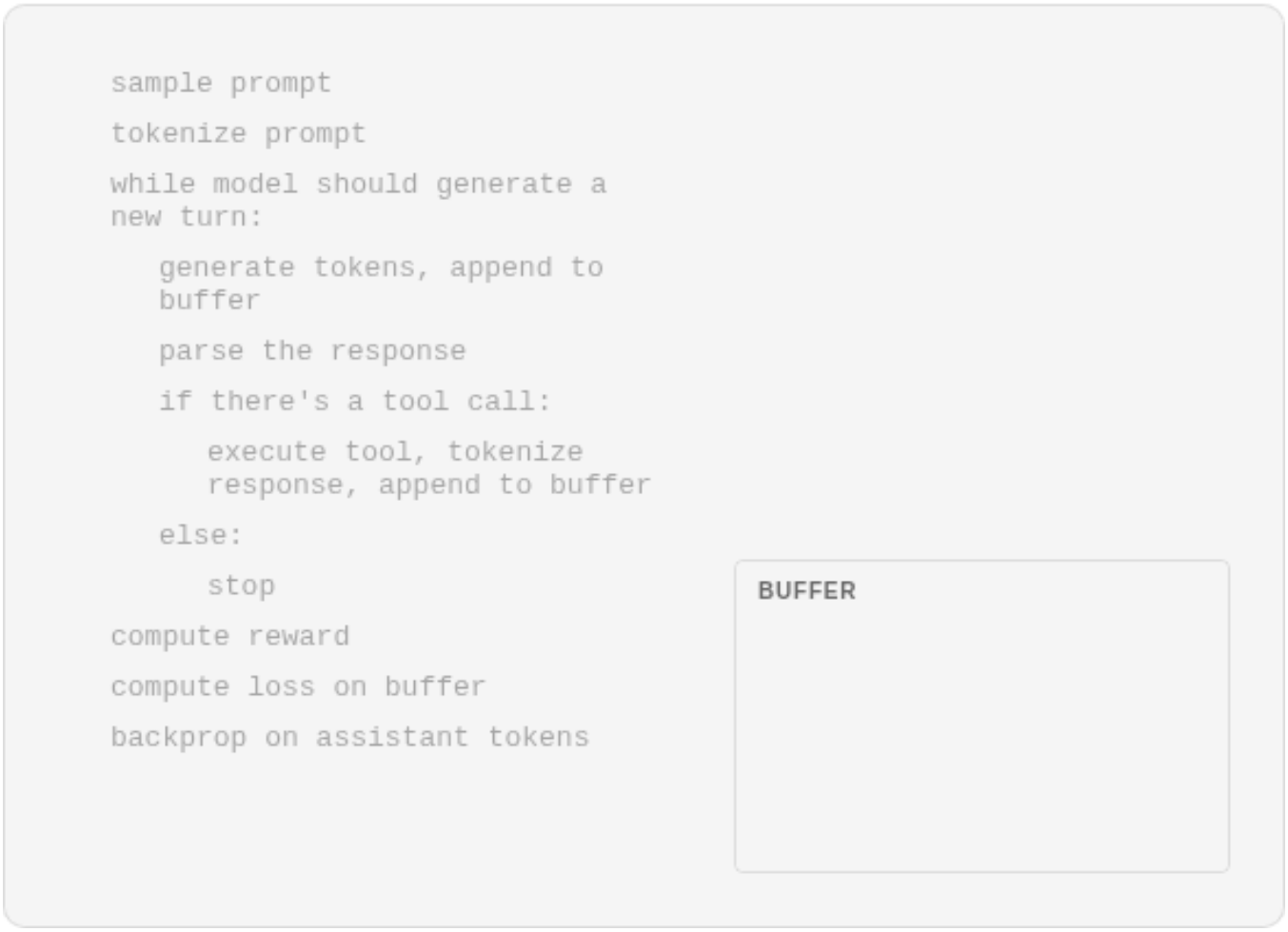
TITO Done Right

> tl;dr Keep the sampled tokens in one buffer, never re-encode them, and both failure modes disappear.

The fix is one rule: never re-encode tokens you've decoded.

The model's sampled tokens go straight into a running buffer, and that buffer is the source of truth. The messages list becomes bookkeeping. We do parse the sampled tokens. We have to, to know whether to dispatch a tool. But the parsed dict is for routing only. It never feeds back into the prompt.

The TITO loop: the buffer accumulates, nothing is re-encoded.



That single change solves both problems from the previous section.

The per-turn boundaries are never lost because they're never recovered. They were known the moment each chunk was appended. The buffer keeps the structure as it grows: these tokens came from the prompt, these from the model, these from the tool, these from the model again. The loss mask is built as you go, not reconstructed afterwards from a re-rendered string.

The token drift is gone for the same reason. The buffer never gets re-encoded. The tokens the policy sampled are exactly the tokens under the gradient. Encoding and decoding are still non-injective. That hasn't changed. But we never *use* the non-injective round-trip. We decode (for tool dispatch), use the result for routing, and throw it away. Nothing decoded ever goes back through `encode`.

The only chat-template operation left in the loop is "tokenize the tool response and append." Everything else is token concatenation.

The tool-response delta

> `tl;dr` The only template operation left: diff two dummy renders for the tool-response tokens, then append them by id.

The TITO loop has one chat-template operation left: tokenize the tool response and append it.

How do you go from the tool result (e.g., `"4"`) to the full wrapped sequence the model expects (`<|im_start|>user\n<tool_response>\n4\n</tool_response><|im_end|>\n<|im_start|>ass`

The way is to use the template only for the tool message. Render the conversation twice (with and without the tool), subtract, and the suffix is exactly the bridge you need to append.

```
>>> from transformers import AutoTokenizer
>>> tok = AutoTokenizer.from_pretrained("Qwen/Qwen2.5-0.5B-Instruct")
>>> messages_prefix = [
...     {"role": "user", "content": "What's 2+2?"},
...     {"role": "assistant", "tool_calls": [
...         {"type": "function", "function": {"name": "calc", "arguments": {"expr": "2+2"}}}
...     ]},
... ]
>>> messages_full = messages_prefix + [{"role": "tool", "content": "4"}]
>>> prefix = tok.apply_chat_template(messages_prefix, return_dict=False)
>>> full = tok.apply_chat_template(messages_full, return_dict=False, add_generation_prompt=True)
>>> delta = full[len(prefix):]
>>> delta
[151644, 872, 198, 27, 14172, 9655, 397, 19, 198, 522, 14172, 9655, 29, 151645, 198, 151644, 77091, 198]
>>> tok.decode(delta)
'<|im_start|>user\n<tool_response>\n4\n</tool_response><|im_end|>\n<|im_start|>assistant\n'
```

That's the entire template-aware part of the loop. The running buffer never sees a re-rendered version of anything the model sampled. It just gets `delta` appended.

The prefix doesn't even have to be a real conversation. Any dummy that ends in an assistant tool call works, since the delta only depends on the tool message and the template's transition logic, not on the prior turns.

The trick has one precondition: the chat template must be *prefix-preserving* for tool messages. Concretely:

```
>>> assert full[:len(prefix)] == prefix
```

If that fails, the subtraction lands on a corrupted suffix. That condition is the subject of the next section.

Prefix preservation

> `tl;dr` It all rests on one property: appending a tool result must extend the render verbatim. Nearly every template already does.

The tool-response delta asks one thing of the chat template, and it's worth stating precisely because the whole loop rests on it. Take any tool messages appended after an assistant tool call. Rendering the conversation *with* them must extend the render *without* them, token for token:

```
render([user, asst_with_tool_call, tool_result])  starts with
render([user, asst_with_tool_call])
```

That is the prefix-preservation property, and the striking thing is how narrow it is. It is required *only* for tool messages. The template is free to do whatever it likes everywhere else (collapse old thinking, rewrite the system prompt, reorder fields) as long as appending a tool result never disturbs bytes it already emitted. User, assistant, and system turns are under no such obligation.

Checking it is a property test, not a proof. Render the prefix, render the extension, compare:

```python
def is_chat_template_prefix_preserving(tokenizer) -> bool:
    dummy_tool_calls = [{"type": "function", "function": {"name": "dummy", "arguments": {}}}]
    messages1 = [
        {"role": "user", "content": "dummy"},
        {"role": "assistant", "content": "", "tool_calls": dummy_tool_calls},
    ]
    messages2 = [
        {"role": "user", "content": "dummy"},
        {"role": "assistant", "content": "", "tool_calls": dummy_tool_calls},
        {"role": "tool", "name": "dummy", "content": "dummy"},
    ]
    ids1 = tokenizer.apply_chat_template(messages1, tokenize=True, return_dict=False)
    ids2 = tokenizer.apply_chat_template(messages2, tokenize=True, return_dict=False, add_generation_prompt=True)
    return ids2[: len(ids1)] == ids1
```

Twelve lines, milliseconds to run, and you can point it at any model the day it ships. So does the property hold in the wild? We ran it across the open-weights families people actually reach for in agentic RL:

family	prefix-preserving for tool messages?
Qwen2.5	✅
Qwen2.5-Coder	✅
Qwen3	❌ (one-line fix below)
Qwen3 Instruct (2507)	✅
Qwen3-VL	✅
Qwen3.5 (think and non-think variants)	✅
Qwen3.6	✅
DeepSeek-V3.1	✅
DeepSeek-R1	✅
DeepSeek-R1-0528	✅
Llama 3.1	✅
Llama 3.2	✅
Llama 4	✅
Gemma 4	✅
Function Gemma	✅
gpt-oss	✅
GLM-4.5	✅
GLM-5	✅
MiniMax-M2.1	✅

Eighteen of nineteen, untouched. The property isn't fragile or rare, it's the quiet default. That's the load-bearing observation for everything that came before: prefix preservation for tool messages is a *weak, narrowly-scoped* condition modern templates satisfy almost by accident, not a demanding one that justifies reimplementing the template per family.

Then there's Qwen3. Easiest way to see what's going on is to render the dummy conversation and inspect the output, before and after appending the tool message:

```
>>> from transformers import AutoTokenizer
>>> tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen3-4B")
>>> dummy_tool_calls = [{"type": "function", "function": {"name": "dummy", "arguments": {}}}]
>>> messages1 = [
...     {"role": "user", "content": "dummy"},
...     {"role": "assistant", "content": "", "tool_calls": dummy_tool_calls},
... ]
>>> messages2 = messages1 + [{"role": "tool", "name": "dummy", "content": "dummy"}]
>>> print(tokenizer.apply_chat_template(messages1, tokenize=False))
# left column below
>>> print(tokenizer.apply_chat_template(messages2, tokenize=False, add_generation_prompt=True))    # right column below
```

apply_chat_template(messages1)	apply_chat_template(messages2)
<|im_start|>user	<|im_start|>user
dummy<|im_end|>	dummy<|im_end|>
<|im_start|>assistant	<|im_start|>assistant
<think>	
</think>	
<tool_call>	<tool_call>
{"name": "dummy", "arguments": {}}	{"name": "dummy", "arguments": {}}
</tool_call><|im_end|>	</tool_call><|im_end|>
	<|im_start|>user
	<tool_response>
	dummy
	</tool_response><|im_end|>
	<|im_start|>assistant

The first one slips an empty `<think>...</think>` block in front of the tool call; the second drops it. The prefix breaks at that exact spot.

One Jinja conditional gates this:

```
{%- if loop.last or (not loop.last and reasoning_content) %}
```

When `reasoning_content` is empty, the `<think>` block renders only on the last assistant turn. Appending a tool result demotes that turn from being last, and the block disappears.

The fix is one line:

```
- {%- if loop.last or (not loop.last and reasoning_content) %}
+ {%- if true %}
```

It costs nothing at inference and restores prefix preservation for training. Qwen3: ✅

Do you need a renderer for this?

> `tl;dr` A per-model renderer guards against re-encoding bugs TITO never has; the lone real requirement is the property from the previous section.

There's a heavier alternative to the loop above. Instead of a ten-line `compute_delta`, you build a renderer: a per-model object that owns the messages-to-tokens boundary. It renders messages, parses completions, and exposes a `bridge_to_next_turn(prev_prompt_ids, prev_completion_ids, new_messages)` that extends the sampled stream byte-for-byte (or returns `None` when it can't prove the extension is safe). One is hand-coded per model family. The `renderers` library ships them for Qwen3, GLM, DeepSeek-V3, Kimi, gpt-oss and a dozen others; `tinker-cookbook` ships its own variant.

The same turn-to-turn step under each design. Both arrive at the same token stream; the difference is where the chat-template logic lives.

Both paths start from the same `(prev_prompt_ids, prev_completion_ids)` and end at the same extended stream. The only difference is *where the template logic lives*: inside a hand-coded per-family object, or inside the one shared `compute_delta` the trainer already calls. The renderer route buys things: a unified API across model families, a `message_indices` array that gives you loss masks by indexing, a bridge that fails loud rather than drifting silently, and the ability to work when you don't control the inference endpoint. If you're plugging into a vendor API that only speaks messages (not tokens), that is not nothing. And the logic lives in Python rather than Jinja: something you can read, test, and step through in a debugger, which the template it replaces (you saw a slice of that Jinja in the Qwen3 fix above) is not.

For RL specifically, most of those are guards against problems TITO never has, and the ergonomic pull rarely gets spent: Python beats Jinja only when you reimplement the template, and TITO doesn't. BPE retokenization drift, canonicalization, JSON whitespace: they only bite a pipeline that re-encodes a string it got from `decode`. TITO never does, so even a non-canonical sample (`[he][llo]` where the canonical encoding is `[hello]`) stays verbatim in the buffer, exactly the tokens under the gradient. It is also why the one property we *do* need can be

checked at the token level rather than the text level: the test runs on a canonical dummy where the two coincide, and the delta is appended by plain id concatenation at an atomic special-token seam (`<|im_end|>` then `<|im_start|>`, neither merges). That property, prefix preservation for tool messages, is the whole and only requirement.

The honest edges

> `tl;dr` Two places reality pushes back: history rewriting genuinely breaks the math, truncation barely registers.

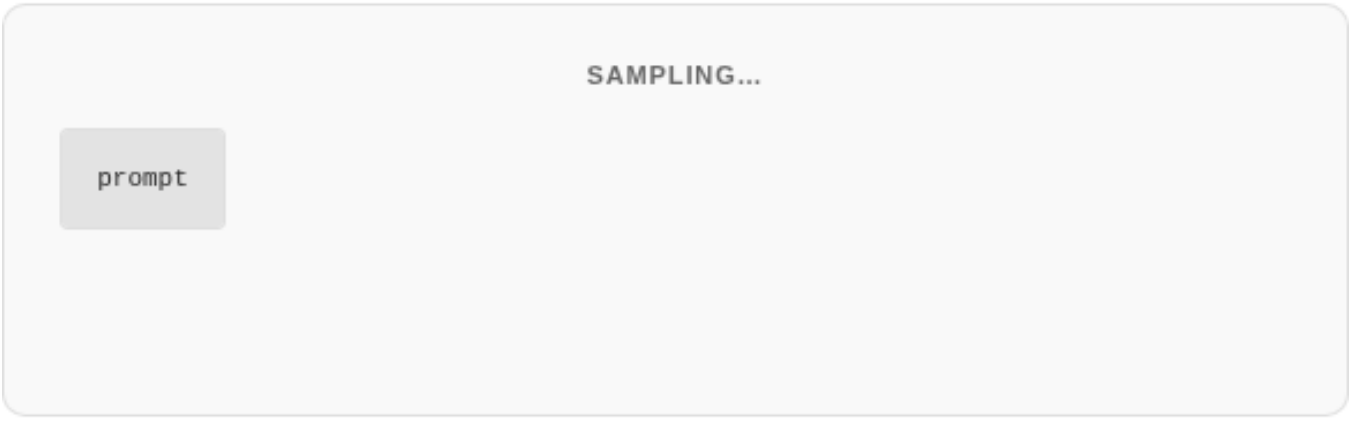
History rewriting

A growing class of agents *edit their own past* as they go. Z.ai's reasoning models ship a `clear_thinking` flag that strips `<think>` blocks from every turn but the last. Long-running coding agents (Claude Code, aider, Codex) compact the conversation when it nears the context limit, replacing dozens of past turns with a short summary. Sub-agent setups go further: a child agent runs, produces a long trace, and only its distilled summary makes it back to the parent. Useful, increasingly standard, and all the same thing under the hood: at some point in the rollout, the tokens that came out of the model are no longer the tokens that go back in.

That breaks TITO at the source. The rule we started with was *you optimize on the exact tokens the model produced.* The "previous turn" the next step is conditioned on never existed as a sampled trajectory, and the PPO/GRPO importance ratio has nothing to say about a Frankenstein prompt the policy never generated. The objective itself is undefined, not just the implementation.

The workaround keeps what you can justify and drops the rest, and it doesn't care *which kind* of history rewriting happened. Pick the last point in the rollout where the past was edited (a compaction, a `clear_thinking` strip, a sub-agent summary, anything) and freeze everything before it as prompt. Loss mask is zero across the frozen part, so prefix preservation and BPE drift no longer apply to it: it's just a prompt the trainer happened to construct in a funny way. Everything after is genuine sampled tokens, still under the gradient, still TITO.

Compaction here as a stand-in for any history rewrite: clear_thinking, sub-agent summarization, anything that replaces past tokens. The mask treats everything up to and including the rewrite as prompt; only what came after carries loss.



The price is the obvious one: the further along the last rewrite lands, the shorter the loss-bearing tail. A long trajectory with periodic rewrites can leave you training on the final few hundred tokens out of tens of thousands sampled.

Truncation

A rollout that hits `max_seq_len` mid-turn ends without its canonical close token. For a renderer, that's a real problem. The bridge anchors on that token to extend the stream byte-for-byte; with it missing, the bridge can't prove a safe extension, returns `None`, and the caller falls back to a full re-render, which re-triggers exactly the drift modes the renderer was built to prevent. The fix is to teach each renderer to recognize a truncated tail and synthesize its own close token there as non-loss context. Another defense, another per-model file, another set of tests.

Under TITO it's a non-event. Generation hits the limit, the buffer ends, the rollout terminates. Mid-reasoning, the dangling `<think>` never needs to close: the tokens are in the buffer as the model produced them, and the loss mask doesn't care that the structure doesn't parse. Mid-tool-call, the parser sees an incomplete block and dispatches nothing; there was no budget left for the tool's result anyway. Nothing on our side of the page, because there was nothing to defend.

The right primitive

You don’t need to reimplement a chat template to train on it, as long as you hold the tokens. (When you don’t, say you’re training against an endpoint that only speaks messages, a renderer is the right and maybe only tool.) You need to know one thing about the template: does appending a tool result extend the render token-for-token? If the answer is yes (it usually does), and you’ve followed the main rule of “never re-encode tokens you’ve decoded,” then your training loop is already correct. The only thing left is to implement the tool-response delta, which is a few lines of code.