MemTR: Enhancing Tool-Calling Reliability via Uncertainty-Triggered FFN-Space Retracing

Anonymous ACL submission

Abstract

Tool calling requires Large Language Models (LLMs) to generate structured decisions including tool names and schema-constrained arguments, where small decoding mistakes can cause hard failures. Existing methods either rely on costly tool-use training data or only constrain syntax, leaving tool selection and argument value errors largely unsolved. We analyze tool-calling failures through a Where–When lens: (Where) failures correlate with persistent uncertainty in late transformer layers, (When) uncertainty concentrates on content-bearing tokens (tool names and argument values) rather than schema tokens. Based on this, and motivated by evidence that transformer Feed Forward Networks (FFNs) as key–value style memories that store and retrieve factual or associative mappings, we propose Memory Space Tool Retracing (MemTR), a weight-free decoding-time method that retrieves relevant tool evidence from the tool library and mixes it into the FFN-output at the uncertain layer, treating FFNs as key–value memories. Through extensive experiments on various model families (Qwen, Llama, and xLAM) and benchmarks (BFCL, ACEBench, APIBank), MemTR reduces tool-calling failures by 2%–9% with only 1%–2% runtime overhead - without any fine-tuning or additional tool-use training data. Our implementation code will be released as open-source software after publication.

1 Introduction

Recent advancements in LLMs have significantly expanded their applications beyond basic NLP tasks to more complex and dynamic functionalities (Qian et al., 2023; Li et al., 2024; Luo et al., 2025). There is growing interest in equipping LLMs with external tools, allowing them to perform tasks that extend beyond traditional language generation, such as interacting with APIs to retrieve information, control devices, or even make complex decisions (Schick et al., 2023; Qin et al., 2023; Zhao et al., 2025).

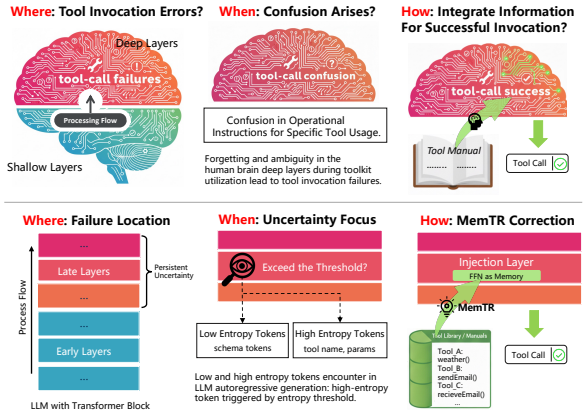

Figure 1: Tool calling in LLMs inspired by human cognition.

Large language models (LLMs), trained on vast natural-language corpora, emit tool-call blocks when invoking external tools (Huang et al., 2024; Qin et al., 2023). The tool-call block (the block containing information such as tool names and arguments, organized in a specific JSON format) is usually separated from natural-language text by special delimiters (such as XML-like tags, fenced blocks, or keyword headers) across different models. For example, Qwen adopts the following structured format as below (Schick et al., 2023; Li et al., 2023):

```
<tool_call>{"name": "tool_name",
"arguments": {"key": "value"}}
</tool_call>.
```

The tool calling process of LLMs can be divided into three key steps: (1) Tool Awareness, where the model identifies the need for external tools to accomplish the task, signaled by outputting the character "<tool_call>" to enter tool mode, (2) Tool Selection, in which the model selects the most appropriate tool by generating its specific name immediately after "{", and (3) Tool Call, where the

model provides the correct parameters and completes the tool calling according to the predefined format, then waits for the tool server's response.

Despite its significance, tool invocation remains fragile: a single-token mistake can invalidate the entire tool calling ("high" token is incorrect, the correct expression should be "medium" as shown in Figure 2(a)). Moreover, in real-world user scenarios, tools are continuously updated and newly added, rendering offline adaptation efforts extremely arduous and labor-intensive. Therefore, there is an urgent need for an online method to rectify tool invocation errors in production environments. Prior work improves tool calling mainly via (i) additional training or fine-tuning (Zeng et al., 2025; Ye et al., 2025), (ii) optimized RAG (Pan et al., 2025), (iii) in-context learning (ICL) (Wei et al., 2022; Yao et al., 2022; Başar et al., 2025) and (iv) constrained decoding or grammar-based validation (Zhang et al., 2023; Xu et al., 2024; Franceschelli and Musolesi, 2024; Park et al., 2025). These approaches are insufficient for our setting. Training and ICL are offline optimizations: they require repeated curation for new tools and evolving APIs, which is costly in production. Constrained decoding and format validation can enforce syntactic well-formedness (e.g., valid JSON), but they do not correct semantic content such as choosing the right tool argument values once the model commits to an incorrect token. (Pan et al., 2025) only updates retriever-side embeddings, so it cannot directly fix token-level argument errors and requires repeated online updates, intervening at input retrieval rather than at the decoding point of failure. In contrast, we target online, token-level correction during inference, focusing on value tokens within tool callings.

In our work, we propose MemTR inspired by human cognition (Figure 1), an online token-level correction framework guided by a lightweight WWH diagnostic mechanism that identify where errors emerge in depth, when uncertainty spikes during generation and how to inject corrective evidence. Where: MemTR selects a late transformer layer exhibiting the highest uncertainty. When: it activates only when uncertainty exceeds a threshold, and only in the tool-call block. How: MemTR retrieves tool evidence by similarity matching against embedded tool library tokens and injects it by mixing with the FFN-output at the selected layer. This design is motivated by prior work interpreting transformer FFNs as key–value memories. Under this view, FFN-output mixing can be seen as a lightweight "memory refresh" that injects tool-relevant evidence during decoding, without changing model weights. In summary, the contributions as follows:

1. Across various families of large language models, we observe that during tool calling, incorrect tokens typically remain high-entropy even in late layers, whereas the entropy of correct tokens drops markedly as depth increases. This suggests that we can identify moments when potential errors are likely to occur in the model's late layers.

2. We propose MemTR, a weight-free decoder patch that selects where to intervene via late-layer entropy, decides when via uncertainty-triggering in tool-call block, and defines how via tool library reinjecting and FFN-output mixing.

3. We demonstrate consistent gains across datasets and models with minimal overhead, and validate the mechanism with entropy and logit-margin diagnostics.

2 Empirical Observations

To motivate the design of MemTR, we conduct a systematic diagnostic of LLMs during tool-calling tasks. We analyze the failure modes through the lens of predictive uncertainty (entropy) to pinpoint exactly where (layer-wise) and when (token-wise) the model's internal alignment with tool specifications collapses.

Where to locate tool calling fail? (Error tokens exhibit persistent uncertainty in late layers.) As shown in Figure 3, both correctly and incorrectly predicted tokens exhibit high entropy in the model's early and middle layers. However, as the model depth increases, the entropy of correctly predicted tokens decreases markedly, while incorrectly predicted tokens remain high-entropy. A similar trend is also observed for the llama3.2-3b and xLAM-2-3b models as shown in Appendix A.1. We therefore use late-layers uncertainty to decide which layer to intervene on.

When to detect tool calling fail? As shown in Figure 2(c), within the tool-call block(from "<tool_call>" to "</tool_call>" on Qwen), we observe a consistent schema-content split. Deterministic schema tokens (e.g., braces, quotes, separators, and fixed keys such as "name" and "arguments") quickly become low-entropy . In contrast, content-bearing tokens(tool-name values and argument val-

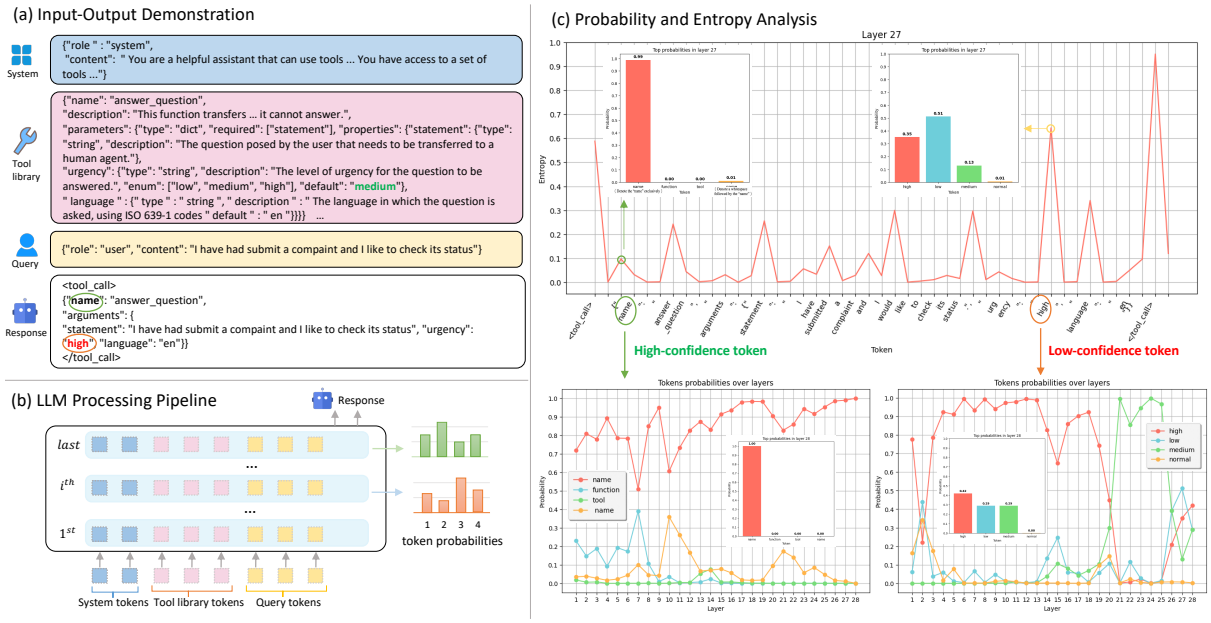

Figure 2: (a) Examples of the system prompt, tool library, user query and model-generated tool content are shown, where the correct token "name" and incorrect token "high" are highlighted. (b) It demonstrates the loading sequence of system prompt tokens, tool library tokens, and query tokens during the prefill stage, while each token is autoregressively generated based on its probability value. (c) Illustration of correct and incorrect token probabilities across transformer layers. The incorrect tokens exhibit high entropy that indicates uncertainty at the 27th layer of the model. From the first to the final layer, the probability values of incorrect tokens remain consistently the highest across all layers. Notably, at the final layer, the highest probability value of incorrect tokens shows minimal differentiation from the second and third highest ones. In contrast, the probability values of correct tokens maintain a high level and remain well-distinguished throughout the entire layer progression from the first to the last.

ues) remain comparatively high entropy. Importantly, tool-call errors concentrate on these content positions, including wrong tool names, missing/incorrect keys, and incorrect values/types. The incorrect tokens (the "high" token) have the maximum entropy values, tokens with entropy below a specific value are correct. This means that we can detect erroneous tokens when the entropy exceeds a certain threshold.

Implication: a targeted, uncertainty-triggered intervention. These observations motivate a testable hypothesis: when the model remains uncertain on content tokens in late layers, its internal representation misaligns with tool specifications. Therefore, injecting tool library evidence at the right time (high uncertainty) and place (late layers) can increase tool-consistent logit margins and reduce tool-call failures. This leads to a targeted strategy adopted by MemTR: intervene only on content/value positions and only in the late layer .

Diagnostics. We validate this mechanism by measuring (i) entropy reduction and (ii) logit-margin increase on the affected tool-call positions after MemTR triggers. See Figure 9 for the effectiveness of this mechanism.

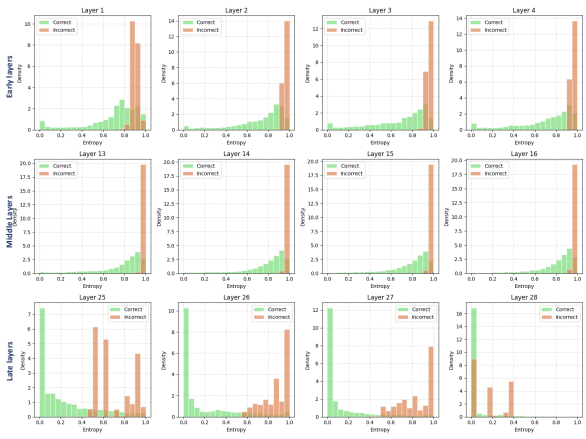

Figure 3: Token entropy distribution for correct and incorrect predictions on the calibration dataset (Qwen2.5-1.5B-FC).

3 Methodology

Motivated by our empirical observations on where and when tool calling fails, we propose MemTR (Figure 4), a weight-free decoding-time interven-

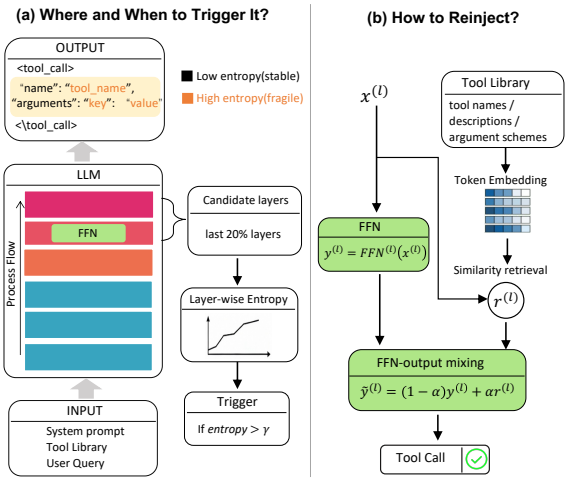

Figure 4: Overview of MemTR. MemTR is activated only on the tool-call block. At each decoding step, we estimate layer-wise predictive uncertainty (entropy) over a candidate layer range. If uncertainty exceeds a threshold, MemTR retrieves tool library evidence and mixes it with the FFN-output.

tion for tool calling. MemTR uses layer-wise predictive uncertainty (token entropy) as a runtime signal to detect fragile decisions on content-bearing tool-call tokens (tool names and argument values). When such uncertainty persists in late layers, MemTR retrieves evidence from the tool library and injects it into the model at the selected layer via a FFN-output mixing operator.

Overview, At each decoding step inside the tool-call block, MemTR (i) computes layer-wise next-token distributions using intermediate representations and estimates uncertainty via entropy, (ii) selects the uncertain layer within a fixed late-layers candidate range (Where), (iii) triggers only if the uncertainty exceeds a threshold and the current position corresponds to a value field in the tool-call JSON (When), and (iv) retrieves tool library evidence by similarity matching against embedded tool library tokens and mixes it into the FFN-output at the selected layer (How). MemTR modifies only decoding-time activations for plug-and-play compatibility across LLM backbones.

3.1 Task Definitions

Given a user query q and candidate tools from tool library $T = \{t_1, t_2, ..., t_N\}$, the goal of the tool calling task is to select suitable tools and extract information as arguments with the model parameters θ :

$$A = [t_1(a_1), ..., t_m(a_m)] = f_\theta(\langle q, T \rangle), \quad (1)$$

where t_j and a_j represent the j-th invoked tool and corresponding arguments, respectively, with $1 \leq j \leq m$ and m being the total number of tool calling needed. The function $f_\theta(\cdot)$ denotes the autoregressive generation manner of LLMs. As shown in Figure 2(b), LLMs load the system prompt, tool library and user query in the prefill phase, then autoregressively output reasoning processes and tool calling information sequentially.

And following DoLa (Chuang et al., 2023), we reuse the pretrained LM head (ζ) to obtain an intermediate next-token distribution at each layer during inference, and measure uncertainty with normalized entropy (Farquhar et al., 2024): $\mu = \sum -p_i log p_i / log N$, where $\{p_i\}_{i=1}^{N}$.

3.2 Refresh Tools Memory Suppresses Confusion

We observe that tool-related information can become less influential in late layers during tool-call generation, correlating with higher uncertainty and failures. A plausible reason is that the model over-relies on recent context tokens rather than tool specifications, weakening tool-spec alignment. MemTR mitigates this by refreshing tool-library information when uncertainty is high.

3.3 FFNs as Key-value Memories

Two fully connected layers constitute vanilla FFNs. We suppose $x \in \mathbb{R}^d$ as an input token of the FFNs, and FFN function can be formulated as

$$FFN(x) = \phi(xW_1)W_2^T, \quad (2)$$

where ϕ is activation function like ReLU or SiLU, and $W_1, W_2 \in R^{d \times D}$ are the weight matrices, in usual D = 4d. particularly, W_1 and W_2 can be rewritten as:

$$W_1 = (k_1, k_2, ..., k_D), W_2 = (v_1, v_2, ..., v_D), \quad (3)$$

where $k_i, v_i \in R^d$ denote entries of key and value, respectively. The FFN function can be reformulated as

$$FFN(x) = \sum \phi(\langle x, k_i \rangle) \cdot v_i. \quad (4)$$

The FFN function can be construed as using input x as a query to measure similarity with keys, find matching values, and gather values by similarity, which works like a key-value memory storing the factual knowledge as found in previous studies (Geva et al., 2021). The approach of injecting

knowledge into FFN has also been used in computer vision literature (Zou et al., 2024). Therefore, the FFN modules are analogous to the human memory module in terms of its role as a key-value memories in Figure 1 .

3.4 Tool Library Representations

Let the concatenated tool Library (tool names, descriptions, argument schemas, and constraints) be tokenized into n tokens. We denote their embedding vectors as:

$$Z_T = [z_{(T,1)}, ..., z_{(T,n)}] \in R^{n \times d}, \tag{5}$$

obtained using the model's input embedding layer.

3.5 Retracing Operator

At decoding step s and layer l, let $x_s^{(l)} \in \mathbb{R}^d$ be the FFN-input, used as the query for tool-memory retrieval. For triggering, we compute layer-wise uncertainty from the layer output $h_s^{(l)} \in \mathbb{R}^d$ by reusing the model's pretrained LM head:

$$p_s^{(l)} = \mathrm{softmax}(\zeta(h_s^{(l)})). \tag{6}$$

Here $\zeta(\cdot)$ shares the original LM head (no extra probes). And then we retrieve a tool-memory vector by similarity:

$$\begin{aligned} w_i &= \mathrm{softmax}_i \left(\frac{\langle x_s^{(l)}, z_{(T,i)} \rangle}{\tau} \right), \\ \Delta(x_s^{(l)}, Z_T) &= \sum_{i=1}^{n} w_i z_{(T,i)}, \end{aligned} \tag{7}$$

where τ is a temperature.

Intuition. The retracing operator $\Delta(x, Z_T)$ can be viewed as a lightweight lookup over tool library information. Given the current hidden state x (e.g., when generating the tool name or an argument), we compute its similarity to each documentation token embedding and form a weighted summary vector. This vector acts as a "tool-memory reminder" that brings tool specifications back into the model's active representation when the model is uncertain.

3.6 FFN-output Intervention

MemTR modifies the FFN-output at a triggered layer by mixing the original FFN-output with the retrieved tool-memory vector. Let

$$y_s^{(l)} = \mathrm{FFN}^{(l)}(x_s^{(l)}) \in \mathbb{R}^d, \tag{8}$$

Algorithm 1 Calibrated uncertainty-triggered MemTR decoding (FFN-output mixing)

Require: LLM with L layers, tool tokens Z_T, candidate layers $\mathcal{L}$, calibrated threshold γ, injection strength α, temperature τ, decoding step s, $\zeta(\cdot)$ for vocabulary head on each layer, Top$K$ for entropy approximation.
Ensure: Generated output Y.

```
tool_mode ← false
for s = 1, 2, ... do
    if output prefix enters <tool_call> block then
        for l ∈ L do
            p_s^(l) ← TopK(softmax(ζ(x_s^(l))))
            u_s^(l) ← Entropy(p_s^(l))
            if u_s^(l) > γ then
                w_i ← softmax(⟨x_s^(l), z_(T,i)⟩/τ)
                r ← Σ_i w_i z_(T,i)
                y ← FFN^(l)(x_s^(l))
                ỹ ← (1 − α) y + α r
                Replace the FFN-output at layer l with ỹ
            end if
        end for
    end if
    Decode next token y_s from the final layer, append to Y, stop if EOS
end for
```

be the original FFN-output, and $r_s^{(l)} = \Delta(x_s^{(l)}, Z_T)$ be the retrieved tool-memory vector. MemTR replaces $y_s^{(l)}$ with:

$$\tilde{y}_s^{(l)} = (1 - \alpha)\, y_s^{(l)} + \alpha\, r_s^{(l)}, \tag{9}$$

where $\alpha \in [0, 1]$ controls the injection strength. The residual stream update then uses $\tilde{y}_s^{(l)}$ in place of $y_s^{(l)}$. This intervention is weight-free and only changes activations during decoding.

3.7 Uncertainty-triggered MemTR

We trigger MemTR only on the tool-call block. Let L be the number of layers. We compute layer-wise predictive uncertainty on a fixed candidate layer set:

$$\mathcal{L} = \{\lceil 0.8L \rceil, \ldots, \lceil 1L \rceil\}. \tag{10}$$

At each decoding step, we choose the trigger layer within $\mathcal{L}$ and apply MemTR if uncertainty with layer-wise entropy exceeds a threshold γ.

Calibration of γ and α. MemTR does not train the model, it only needs a rule to decide *when* the model is uncertain enough to activate the intervention. Just like all machine learning algorithms, a validation set is required for hyperparameter tuning to ensure that the optimal performance of these parameters is achieved on the test set. We therefore calibrate an entropy threshold γ on a small labeled calibration set that is strictly disjoint from all evaluation benchmarks. For each backbone model, γ

and α is calibrated once and then kept fixed across all benchmarks.

Zero-shot triggering with EMA. We also present a zero-shot triggering method that does not require calibrated dataset as illustrated in Appendix B.1. EMA thresholding is computed online within each example and reset between examples, it uses only model-internal uncertainty signals and does not access gold labels, evaluation feedback, or aggregate statistics across the test set.

3.8 Mechanistic Evidence: Scope and Diagnostics

MemTR is a decoding-time intervention that mixes the original FFN-output with retrieved tool-documentation evidence:

$$\tilde{y}_s^{(l)} = (1-\alpha)y_s^{(l)} + \alpha r_s^{(l)}, \quad \alpha \in [0,1], \quad (11)$$

where $y_s^{(l)} = \mathrm{FFN}^{(l)}(x_s^{(l)}), r_s^{(l)} = \Delta(x_s^{(l)}, Z_T)$.

What we do claim. We provide a mechanistic interpretation grounded in FFNs as key-value memories (Geva et al., 2021) and focus on testable logit-level diagnostics.

Why uncertainty-triggered Intervention is Reasonable. When the next-token distribution has high entropy, the model typically has a small margin among top candidates, thus a small activation-level perturbation is more likely to change the argmax decision. This motivates our design: MemTR activates only when entropy exceeds a threshold and only inside the `<tool_call>` block (and only on content-bearing value positions via JSON gating).

Measurable predictions (diagnostics). We evaluate MemTR using two diagnostics on the affected tool-call positions:

1. Entropy reduction: after MemTR triggers, predictive entropy on the triggered tool-call tokens decreases.

2. Logit-margin increase: MemTR increases the logit margin between the tool-consistent candidate and its strongest competitor (e.g., among valid tool-name tokens or schema-consistent value candidates).

We report both effects. A local linearization derivation (Jacobian view) of the margin effect is provided in Appendix C.

4 Experiments

4.1 Experimental Settings

Benchmark and evaluation. We selected the Berkeley Function Call Leaderboard (BFCL) (Patil et al.), as the evaluation framework. we evaluate on two other tool-calling benchmarks: ACEBench (Chen et al., 2025) and API-Bank (Li et al., 2023). The details of those are list in Appendix G.

4.2 Implementation Details

Purpose (no training). MemTR is weight-free and does not update model weights. It uses three hyperparameters: (i) the candidate trigger-layer set $\mathcal{L}$, (ii) the uncertainty threshold γ for deciding whether to activate MemTR, and (iii) the mixing weight α for FFN-output mixing. The implementation details refer to Algorithm 1. Although MemTR is weight-free (no gradient updates), the calibrated setting requires a small labeled set to select γ and α. We therefore additionally report a zero-shot EMA variant that requires no labels, and we compare calibrated vs. zero-shot results throughout. The implementations of the two methods are provided in the Appendix D.

4.3 Experimental Results.

Results on BFCL. To demonstrate the superiority of the model performance under the method we propose, we implement our method on Qwen, Llama and xLAM. And we compare the tool calling accuracy in the BFCL. All numbers in Table 1 are generated by the official BFCL evaluation script, we will release the exact command lines and raw model outputs. The results are shown in Table 1, from which we can make the following observations:

(i) The performance of MemTR outperforms other weight-free methods, namely prompt optimization based COT, sampling with beam search and constrained decoding-based method. (ii) Our method also demonstrates effectiveness on models including Qwen, Llama and xLAM with a similar pattern observed in the metric improvement.

Results on more benchmarks. To provide a more comprehensive evaluation, we continue to conduct experiments on two other representative benchmarks ACEBench and API-Bank. As shown in Table 2, our method also exhibits effectiveness on other datasets.

We also demonstrate the effectiveness of our method on BFCL V3, as shown in Table 3.

Models	Method	None-Live				Live				Overall		
		Simple	Multiple	Parallel	Miltiple Parallel	Simple	Multiple	Parallel	Miltiple Parallel	None-Live	Live	Overall
Qwen2.5-1.5B-FC	Vanilla	74.91%	82.50%	63.50%	68.50%	65.50%	56.22%	56.25%	58.33%	73.13%	58.03%	64.97%
	COT	77.27%	82.50%	66.50%	69.00%	65.50%	60.02%	50.00%	50.00%	74.87%	60.77%	67.25%
	BeamSearch(n=5)	74.91%	83.50%	55.00%	60.00%	66.67%	61.44%	43.75%	29.17%	70.35%	61.66%	65.65%
	ToolDec	76.00%	83.50%	65.00%	69.00%	65.89%	61.44%	**56.25%**	54.17%	74.17%	62.10%	67.65%
	MemTR(zero-shot)	77.45%	84.00%	70.50%	71.00%	65.89%	61.73%	50.00%	58.33%	76.26%	62.32%	68.73%
	MemTR(Calibrated)	**77.82%**	**85.00%**	**73.00%**	**72.50%**	**66.67%**	**62.39%**	**56.25%**	**62.50%**	**77.30%**	**63.14%**	**69.65%**
Qwen2.5-7B-FC	Vanilla	81.82%	94.00%	86.50%	82.50%	70.16%	72.93%	62.50%	**54.17%**	84.87%	71.95%	77.89%
	COT	82.91%	94.50%	88.00%	83.50%	70.16%	73.31%	56.25%	50.00%	85.91%	72.09%	78.45%
	BeamSearch(n=5)	82.55%	93.50%	86.50%	82.50%	67.83%	73.31%	50.00%	45.83%	85.13%	71.50%	77.77%
	ToolDec	84.55%	95.00%	88.50%	84.00%	70.54%	73.50%	62.50%	50.00%	86.96%	72.39%	79.09%
	MemTR(zero-shot)	85.45%	95.50%	89.00%	86.00%	70.93%	73.88%	68.75%	**54.17%**	87.91%	72.91%	79.81%
	MemTR(Calibrated)	**86.18%**	**96.00%**	**89.50%**	**87.00%**	**72.87%**	**74.07%**	**75.00%**	**54.17%**	**88.61%**	**73.50%**	**80.45%**
Llama-3.2-3B-Instruct	Vanilla	81.45%	92.50%	86.50%	78.00%	62.40%	56.60%	12.50%	37.50%	83.65%	56.85%	69.17%
	COT	81.82%	92.50%	86.50%	77.00%	63.57%	57.26%	**18.75%**	33.33%	83.65%	57.59%	69.57%
	BeamSearch(n=5)	81.64%	91.00%	87.00%	76.50%	62.79%	56.41%	12.50%	37.50%	83.30%	56.77%	68.97%
	ToolDec	82.73%	93.00%	87.50%	78.50%	63.95%	57.36%	12.50%	37.50%	84.61%	57.74%	70.09%
	MemTR(zero-shot)	83.45%	93.50%	87.50%	78.50%	64.73%	57.55%	**18.75%**	37.50%	85.04%	58.11%	70.49%
	MemTR(Calibrated)	**84.18%**	**94.50%**	**88.00%**	**79.00%**	**65.89%**	**57.83%**	**18.75%**	**41.67%**	**85.74%**	**58.62%**	**71.09%**
Llama-xLAM-2-8B	Vanilla	82.91%	92.50%	84.00%	78.00%	72.87%	64.48%	56.25%	**54.17%**	83.91%	65.80%	74.13%
	COT	84.36%	93.00%	85.50%	82.50%	74.81%	66.38%	56.25%	41.67%	85.74%	67.43%	75.85%
	BeamSearch(n=5)	81.82%	89.50%	78.50%	79.50%	70.16%	60.87%	43.75%	37.50%	82.17%	62.03%	71.29%
	ToolDec	84.73%	94.00%	86.50%	83.50%	74.81%	66.48%	56.25%	**54.17%**	86.43%	67.73%	76.33%
	MemTR(zero-shot)	85.64%	94.50%	87.50%	83.00%	75.19%	66.67%	56.25%	**54.17%**	87.04%	67.95%	76.73%
	MemTR(Calibrated)	**86.00%**	**95.00%**	**88.00%**	**84.00%**	**75.97%**	**66.76%**	**62.50%**	**54.17%**	**87.57%**	**68.25%**	**77.13%**

Table 1: Accuracy comparison on BFCLv2 with different weight-free methods.(Implementation details of the comparative methods shown in Appendix D.0.4)

Models (+Method)	ACEBench	APIBank
Qwen2.5-1.5B-FC	45.21%	39.12%
+ COT	46.52%	41.26%
+ ToolDec	48.33%	43.95%
+ MemTR	**52.27%**	**48.32%**
+ MemTR+COT	**52.32%**	**48.91%**
+ MemTR+ToolDec	**53.16%**	**49.21%**
Qwen2.5-7B-FC	51.81%	45.62%
+ COT	51.78%	45.15%
+ ToolDec	53.23%	46.82%
+ MemTR	**54.11%**	**51.10%**
+ MemTR+COT	**56.89%**	**51.89%**
+ MemTR+ToolDec	**57.12%**	**53.23%**
Llama-3.2-8B-Instruct	48.70%	59.11%
+ COT	49.25%	59.06%
+ ToolDec	49.82%	60.12%
+ MemTR	**54.20%**	**63.80%**
+ MemTR+COT	**54.45%**	**64.10%**
+ MemTR+ToolDec	**56.26%**	**65.21%**

Table 2: Different methods and their combinations on various models and benchmarks.(MemTR is Calibrated method.)

More study. As a plug-and-play method, MemTR can integrate seamlessly with prompt engineering or constrained decoding method. Table 2 presents the performance of these combination which significantly enhance the model's tool calling capabilities.

5 Analysis

In this section, we present an in-depth analysis along five dimensions: (1) error-cases analysis, (2) the impact of the threshold and injection ratio, (3) uncertainty reduction achieved by MemTR, (4) uncertainty analysis, and (5) computational cost.

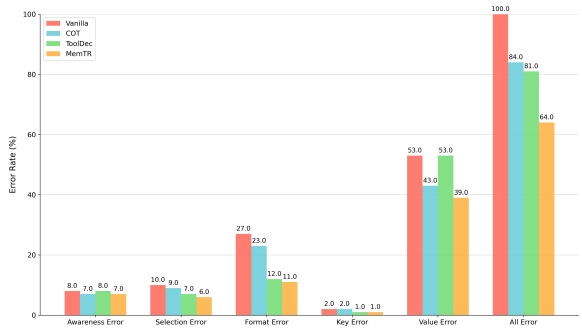

Figure 5: Test on the BFCL, the reduction ratios of five error types achieved by the COT, ToolDec and MemTR.

Error-cases analysis. We observe that the ToolDec method can suppress format errors and selection errors, but it has no effect on value errors as shown in Figure 5. In contrast, MemTR not only works on the other two error types but also achieves excellent suppression of value errors. This observation aligns with our prior understanding: ToolDec is a constrained decoding method that cannot exert any influence on parameter values, since parameter values are derived from user queries rather than the tool library. Moreover, Figure 11 shows proportion of error types, Value-type errors are the most common. By comparison, MemTR performs semantic-level realignment, which enables it to achieve favorable results in addressing value errors as well.

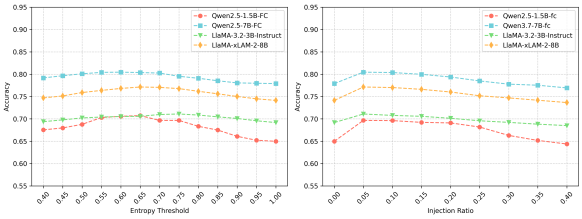

Figure 6: (Left) Results under different thresholds γ. (Right) Results under different injection ratios α, both on BFCLv2 benchmark.

The impact of the threshold and injection ratio. Figure 6 shows the performance under different injection ratios α and thresholds γ, we find that setting γ between 0.55 and 0.85 can improve performance. Understandably, a high threshold makes TR hard to trigger, while a low threshold would trigger TR at an earlier layer. For injection rate α, we find the optimal rate is around 0.05.

Uncertainty reduction achieved by MemTR. As shown in Figure 9, for error token "high" MemTR effectively reduces uncertainty after the 26th layer triggered TR. This consequently causes the probability of the token high to start decreasing after the 26th layer, while the correct token medium becomes the most probable one, with its margin over the second and third-ranked tokens increasing substantially.

Uncertainty analysis. We observe that the entropy values of correct tokens drop to very low levels in the last few layers, whereas those of incorrect tokens remain relatively high even in the final layer. For detailed analysis, please refer to the Appendix E.

Computational cost. We leave the cost analysis in Appendix F.

6 Related Work

In-context learning. ReAct (Yao et al., 2022) enables LLMs to generate reasoning trails and task-specific tool calling through interaction within the environment, improving the reasoning and decision-making capabilities of AI agents. Reflexion (Shinn et al., 2023), Self-refine (Madaan et al., 2023) and ReflAct (Kim et al., 2025) analyze the external feedback of user interaction within the environment and then iteratively refine and polish results via well-designed reflexion prompts. All of these designs allow us to better understand user instructions.

Sampling and decoding in language lodels. A variety of decoding strategies have been proposed to improve language model performance, including top-k sampling (Fan et al., 2018), temperature-based sampling (Ficler and Goldberg, 2017), and nucleus sampling (Holtzman et al., 2019). Beyond these, Constrained decoding (Willard and Louf, 2023; Chen et al., 2022; Fang et al., 2023; Lu et al., 2022) improves generation quality by limiting the vocabulary to a smaller set of candidate tokens, ToolDec (Zhang et al., 2023) is directly employed to tool calling.

Supervised training. Tool-augmented fine-tuning improves tool calling by training LLMs on tool-call traces. A major limitation is the need for high-quality interaction data, which typically requires executing tools in realistic environments (Feng et al., 2025; Singh et al., 2025). Collecting such data is expensive and time-consuming.

7 Conclusion

We presented MemTR, a weight-free decoding-time method for improving LLM tool calling. MemTR detects high uncertainty on tool-call blocks and reinjects tool library evidence by mixing it with FFN-output at an intermediate layer. The method requires no model training and can be applied across LLM backbones. Experiments demonstrate consistent improvements with minimal overhead, and analyses show increased logit margins and reduced uncertainty on tool-call tokens.

8 Limitations

First, our experiments were conducted on models up to 8B due to resource constraints. Additionally, this work focuses on tool calling accuracy selecting the correct tool and providing precise parameters but does not address retrieving tools from a large-scale tool pool.

9 Ethics Statement

We conduct no user studies or user-facing deployment. We create a small calibration set by re-annotating publicly available benchmark instances to tune hyperparameters, and annotations are done internally. The data contain no intended personally identifying or sensitive information, and no IRB review was sought since no human-subjects study is involved.

References

Erkan Başar, Xin Sun, Iris Hendrickx, Jan de Wit, Tibor Bosse, Gert-Jan De Bruijn, Jos A Bosch, and Emiel Krahmer. 2025. How well can large language models reflect? a human evaluation of llm-generated reflections for motivational interviewing dialogues. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 1964–1982.

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, and 1 others. 2025. Acebench: Who wins the match point in tool usage? *arXiv preprint arXiv:2501.12851*.

Xiang Chen, Zhixian Yang, and Xiaojun Wan. 2022. Relation-constrained decoding for text generation. *Advances in Neural Information Processing Systems*, 35:26804–26819.

Yung-Sung Chuang, Yujia Xie, Hongyin Luo, Yoon Kim, James Glass, and Pengcheng He. 2023. Dola: Decoding by contrasting layers improves factuality in large language models. *arXiv preprint arXiv:2309.03883*.

Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical neural story generation. *arXiv preprint arXiv:1805.04833*.

Hao Fang, Anusha Balakrishnan, Harsh Jhamtani, John Bufe, Jean Crawford, Jayant Krishnamurthy, Adam Pauls, Jason Eisner, Jacob Andreas, and Dan Klein. 2023. The whole truth and nothing but the truth: Faithful and controllable dialogue response generation with dataflow transduction and constrained decoding. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5682–5700.

Sebastian Farquhar, Jannik Kossen, Lorenz Kuhn, and Yarin Gal. 2024. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630(8017):625–630.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*.

Jessica Ficler and Yoav Goldberg. 2017. Controlling linguistic style aspects in neural language generation. *arXiv preprint arXiv:1707.02633*.

Giorgio Franceschelli and Mirco Musolesi. 2024. Creative beam search: Llm-as-a-judge for improving response generation. *arXiv preprint arXiv:2405.00099*.

Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. 2021. Transformer feed-forward layers are key-value memories. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5484–5495.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*.

Shijue Huang, Wanjun Zhong, Jianqiao Lu, Qi Zhu, Jiahui Gao, Weiwen Liu, Yutai Hou, Xingshan Zeng, Yasheng Wang, Lifeng Shang, and 1 others. 2024. Planning, creation, usage: Benchmarking llms for comprehensive tool utilization in real-world complex scenarios. *arXiv preprint arXiv:2401.17167*.

Jeonghye Kim, Sojeong Rhee, Minbeom Kim, Dohyung Kim, Sangmook Lee, Youngchul Sung, and Kyomin Jung. 2025. Reflact: World-grounded decision making in llm agents via goal-state reflection. *arXiv preprint arXiv:2505.15182*.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*.

Yanda Li, Chi Zhang, Wanqi Yang, Bin Fu, Pei Cheng, Xin Chen, Ling Chen, and Yunchao Wei. 2024. Appagent v2: Advanced agent for flexible mobile interactions. *arXiv preprint arXiv:2408.11824*.

Ximing Lu, Sean Welleck, Peter West, Liwei Jiang, Jungo Kasai, Daniel Khashabi, Ronan Le Bras, Lianhui Qin, Youngjae Yu, Rowan Zellers, and 1 others. 2022. Neurologic a* esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 780–799.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, and 1 others. 2025. Large language model agent: A survey on methodology, applications and challenges. *arXiv preprint arXiv:2503.21460*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36:46534–46594.

Yu Pan, Xiaocheng Li, and Hanzhao Wang. 2025. Online-optimized rag for tool use and function calling. *arXiv preprint arXiv:2509.20415*.

Kanghee Park, Timothy Zhou, and Loris D'Antoni. 2025. Flexible and efficient grammar-constrained decoding. *arXiv preprint arXiv:2502.05111*.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, and 1 others. 2023. Chatdev: Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652.

Joykirat Singh, Raghav Magazine, Yash Pandya, and Akshay Nambi. 2025. Agentic reasoning and tool integration for llms via reinforcement learning. *arXiv preprint arXiv:2505.01441*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Brandon T Willard and Rémi Louf. 2023. Efficient guided generation for large language models. *arXiv preprint arXiv:2307.09702*.

Xinhao Xu, Hui Chen, Zijia Lin, Jungong Han, Lixing Gong, Guoxin Wang, Yongjun Bao, and Guiguang Ding. 2024. Tad: A plug-and-play task-aware decoding method to better adapt llms on downstream tasks. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pages 6587–6596.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Junjie Ye, Guanyu Li, Songyang Gao, Caishuang Huang, Yilong Wu, Sixian Li, Xiaoran Fan, Shihan Dou, Tao Ji, Qi Zhang, and 1 others. 2025. Tooleyes: Fine-grained evaluation for tool learning capabilities of large language models in real-world scenarios. In *Proceedings of the 31st international conference on computational linguistics*, pages 156–187.

Xingshan Zeng, Weiwen Liu, Xu Huang, Zezhong Wang, Lingzhi Wang, Liangyou Li, Yasheng Wang, Lifeng Shang, Xin Jiang, Ruiming Tang, and 1 others. 2025. Toolace-r: Tool learning with adaptive self-refinement. *arXiv preprint arXiv:2504.01400*.

Kexun Zhang, Hongqiao Chen, Lei Li, and William Yang Wang. 2023. Tooldec: Syntax error-free and generalizable tool use for llms via finite-state decoding.

Weikang Zhao, Xili Wang, Chengdi Ma, Lingbin Kong, Zhaohua Yang, Mingxiang Tuo, Xiaowei Shi, Yitao Zhai, and Xunliang Cai. 2025. Mua-rl: Multi-turn user-interacting agent reinforcement learning for agentic tool use. *arXiv preprint arXiv:2508.18669*.

Xin Zou, Yizhou Wang, Yibo Yan, Yuanhuiyi Lyu, Kening Zheng, Sirui Huang, Junkai Chen, Peijie Jiang, Jia Liu, Chang Tang, and 1 others. 2024. Look twice before you answer: Memory-space visual retracing for hallucination mitigation in multimodal large language models. *arXiv preprint arXiv:2410.03577*.

A Additional Observations

A.1 The Entropy Distribution for More Models.

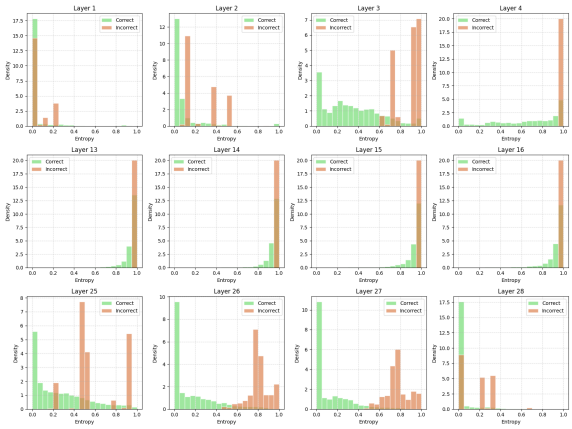

Figure 7: Token entropy distribution for correct and incorrect predictions on the calibration dataset (llama3.2-3b).

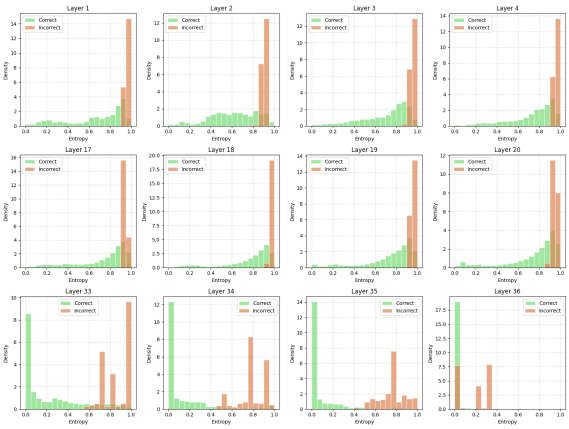

Figure 8: Token entropy distribution for correct and incorrect predictions on the calibration dataset (xLAM-2-3b).

A.2 Empirical motivation: text priors vs. tool specifications.

Tool documentation is structured (tool names, required keys, type constraints) and differs from the natural-language distribution LLMs are primarily pretrained on. During autoregressive decoding, models can over-rely on textual priors from the user query and previously generated reasoning tokens, while tool specifications become less influential on tool-specific decisions.

B Methodology Appendix

B.1 Methods Overview for Zero-shot Triggering with EMA.

To remove labeled calibration for the uncertainty threshold, we adopt an online adaptive threshold based on exponential moving averages (EMA). The implementation details refer to Algorithm 2. Within the block, at each decoding step s we compute u_s and $l^\star$ and maintain:

$$m_s = \beta m_{s-1} + (1-\beta)u_s, \tag{12}$$

$$v_s = \beta v_{s-1} + (1-\beta)(u_s - m_s)^2, \tag{13}$$

initialized as $m_0 = 0$ and $v_0 = 0$ and reset between examples. We set the adaptive threshold as

$$\gamma_s = m_s + \lambda\sqrt{v_s}. \tag{14}$$

MemTR triggers at step s if $u_s > \gamma_s$.

JSON-state gating and warm-up. Tool callings follow a structured JSON format (e.g., `{"name": "...", "arguments": {...}}`). The first few tokens inside the tool-call block are typically fixed-format (delimiters and schema keys) and thus have very low entropy, which can make early EMA thresholds overly small and lead to spurious activations later. To stabilize zero-shot triggering without any dataset-level statistics, we apply two deterministic rules: (1) **Warm-up:** for the first w decoding steps after entering the tool-call block, we update (m_s, v_s) but do not trigger MemTR, (2) **JSON-state gating:** we allow triggering only when generating content-bearing positions, namely (i) the tool-name *value* following the `"name":` key, or (ii) an argument *value* inside the `"arguments":{...}` object (the token block after a colon until the next comma or closing brace at the same nesting level). We skip triggering on fixed-format tokens (e.g., delimiters, quotes, and schema keys such as `"name"` and `"arguments"`). We implement gating with a lightweight incremental parser over the generated prefix. Unless otherwise stated, we fix $w = 4$ and keep $(\beta, \lambda, \alpha, K)$ unchanged across all benchmarks in the zero-shot setting.

C Local Linearization View of MemTR and Logit-Margin Effect

This appendix gives a supplementary logit-level explanation for why FFN-output mixing can increase the margin of tool-consistent tokens under uncertainty-triggered intervention.

MemTR modifies the FFN-output at a triggered layer by mixing the original FFN-output y with a retrieved tool-memory vector r:

$$\tilde{y} = (1-\alpha)y + \alpha r. \tag{15}$$

Let h denote the hidden state before the LM head at the current decoding step, and assume the LM head is linear: logits $\ell = Wh$, where $W \in \mathbb{R}^{|V|\times d}$.

When MemTR is applied at an intermediate layer, the final hidden state becomes a deterministic function of $\tilde{y}$ instead of y. Locally (for small interventions), the resulting logit change can be approximated as:

$$\Delta\ell \approx J \cdot (\tilde{y} - y) = \alpha J \cdot (r - y), \tag{16}$$

where J is the Jacobian mapping from the FFN-output at the triggered layer to the pre-head representation.

Implication (margin improvement under alignment). Consider a set of tool-consistent tokens $\mathcal{S}$ (e.g., valid tool names / required keys / schema-consistent value tokens) and confusing alternatives $\mathcal{C}$. If the intervention direction $(r - y)$ is aligned such that for all $s \in \mathcal{S}$ and $c \in \mathcal{C}$,

$$\langle g_s - g_c, (r - y)\rangle \geq \delta, \tag{17}$$

where g_v denotes the corresponding row of $(J^\top W^\top)$ for token v, then MemTR increases the logit margin between s and c by at least $\alpha\delta$ in the local approximation:

$$(\tilde{\ell}_s - \tilde{\ell}_c) - (\ell_s - \ell_c) \geq \alpha\delta. \tag{18}$$

Connection to our diagnostics. Motivated by the above, we measure (i) token-level predictive entropy and (ii) logit margins on triggered tool-call positions before/after MemTR activation (Section 5). We define the logit margin at a triggered step as $\text{margin} = \ell(y^\star) - \max_{v \neq y^\star} \ell(v)$, where $y^\star$ is the gold token when available, or the highest-logit token among tool-consistent candidates.

D Experimental Implementation Details

D.0.1 Calibrated uncertainty-triggered MemTR decoding

Calibration set and strict isolation. We construct a small labeled calibration set by manually annotating tool-call outputs and it does not contain user-identifying information.. Specifically, we annotate 800 instances covering 300 tools across 10 domains. A tool calling is labeled Correct only if (i) the output is parseable (format-valid), (ii) the tool name matches the ground truth, and (iii) all required argument keys and values are correct, otherwise it is labeled Incorrect. Importantly, this calibration set is strictly disjoint from all evaluation benchmarks: we ensure no overlap in instances and tool schemas, and we do not use any labels from BFCL, ACEBench or API-Bank for hyperparameter selection.

How the trigger layer is chosen. We use a fixed candidate layers range

$$\mathcal{L} = \{\lceil 0.8L \rceil, \ldots, \lceil 1.0L \rceil\}, \tag{19}$$

where L is the number of transformer layers. During decoding, at each step s on the tool-call block, we compute layer-wise uncertainty $u_s^{(l)}$ for all $l \in \mathcal{L}$ and select the trigger layer when $u_s^{(l)} > \gamma$, and then MemTR applies the FFN-output intervention. This layer-selection rule is fixed *a priori* and is not tuned per benchmark.

How γ and α are chosen. We select α on the same calibration set using a small grid (e.g., $\alpha \in \{0.01, 0.03, 0.05, 0.1\}$) and choose the best value. The same choosing method is also applied to γ, whose range is from 0.6 to 0.95. Unless otherwise stated, γ and α are calibrated once per backbone model.

We emphasize that parameter choosing is model-dependent and an intrinsic property of the model, independent of the calibration dataset.

Cross-benchmark fixed hyperparameters. Unless stated otherwise, we calibrate $\mathcal{L}$ (by fixing the range rule above), γ, and α once per backbone model and then keep them fixed across all benchmarks, splits, and tool-calling settings, without any per-benchmark tuning.

D.0.2 Zero-shot uncertainty triggering via EMA.

To avoid any labeled calibration for the uncertainty threshold, we adopt an online, per-example adaptive threshold based on exponential moving averages (EMA). Within the `<tool_call>` block, at each decoding step s we first compute the layer-wise uncertainty $u_s^{(l)} = \text{Entropy}(p_s^{(l)})$ for $l \in \mathcal{L}$ and set $u_s = \max_{l\in\mathcal{L}} u_s^{(l)}$ with the corresponding layer $l^\star = \arg\max_{l\in\mathcal{L}} u_s^{(l)}$. We then maintain EMAs of the mean and (uncentered) variance of

the uncertainty sequence:

$$m_s = \beta m_{s-1} + (1-\beta)u_s, \tag{20}$$

$$v_s = \beta v_{s-1} + (1-\beta)(u_s - m_s)^2, \tag{21}$$

initialized as $m_0 = 0$ and $v_0 = 0$ at the beginning of each example. The adaptive trigger threshold is defined as

$$\gamma_s = m_s + \lambda\sqrt{v_s}, \tag{22}$$

where $\beta \in (0,1)$ controls the smoothing and $\lambda > 0$ controls the sensitivity. We trigger MemTR at step s if $u_s > \gamma_s$, and apply the FFN-output mixing intervention at layer $l^\star$.

No test-time leakage. EMA thresholding is computed *online within each example* and reset between examples, it uses only model-internal uncertainty signals and does not access gold labels, evaluation feedback, or aggregate statistics across the test set. All hyperparameters (β, λ, α) are fixed *a priori* and kept unchanged for all benchmarks in the zero-shot setting.

D.0.3 Two deployment settings.

MemTR does not update model weights. We consider two settings: (i) Calibrated MemTR, which selects γ and α once per backbone model using a small labeled calibration set, and (ii) Zero-shot MemTR, which uses an online adaptive threshold γ_s estimated from an exponential moving average of entropy and requires no labeled data. We report both settings in our main results to separate the gains from (light) calibration versus the decoding-time intervention itself.

Strict isolation and leakage prevention. To prevent any test-time leakage, we enforce disjointness between the calibration set and evaluation benchmarks at two levels: (i) Instance-level disjointness: no overlapping user queries or dialogue turns. (ii) Tool-schema disjointness: we ensure that tool names and argument schemas (key sets) in the calibration set do not appear in BFCL/ACEBench/API-Bank. Concretely, we hash each tool schema by (tool name, sorted argument key list, and normalized key types when available) and verify no hash collisions across splits. We will release the exact schema-hash scripts and the list of tool schemas used for calibration to enable independent verification.

D.0.4 Implementation details of the comparative methods

COT. The template of the CoT prompt is provided in Table 4.

ToolDec. The implement and code refer to paper ToolDec (Zhang et al., 2023).

Model/Method	Single-Turn		Multi-Turn	Hallucination		Overall
	None-Live	Live		Relevance	Irrelevance	
Qwen3-4B-FC	87.83%	76.39%	22.13%	87.50%	78.13%	70.05%
+MemTR	**89.48%**	**77.05%**	**22.63%**	**93.75%**	**78.48%**	**70.88%**
Qwen3-8B-FC	87.57%	80.46%	41.75%	93.75%	77.05%	74.51%
+MemTR	**90.00%**	**80.83%**	**44.63%**	93.75%	**77.68%**	**75.93%**
xLAM-2-1B	69.04%	55.14%	36.00%	87.50%	64.46%	57.76%
+MemTR	**74.09%**	**56.03%**	**42.38%**	**93.75%**	64.46%	**60.51%**
xLAM-2-3B	82.96%	62.99%	58.38%	87.50%	**59.82%**	66.62%
+MemTR	**86.43%**	**63.66%**	**61.38%**	**100.00%**	59.64%	**68.27%**
Llama-xLAM-2-8B	84.52%	67.95%	70.00%	87.50%	63.30%	71.51%
+MemTR	**87.57%**	**68.25%**	**71.38%**	**93.75%**	**63.39%**	**72.68%**

Table 3: Result on BFCLv3

E Analysis for Uncertainty Analysis.

Figure 12 shows the inference result including query, result and tool library. We conduct a preliminary analysis with 28-layer Qwen-2.5-1.5B-FC. Figure 10 shows the uncertainty scores of different early layers when decoding the answer, we observe that from the earlier layers to the later layers, the entropy of all tokens gradually decreases. However, the entropy of some tokens drops rapidly while that of others declines slowly. Moreover, the entropy of certain tokens does not decrease even in the final layer, for example "dont care" in Figure 10. These tokens require our focused attention, because they do not decrease even in the final layer.

Specifically, we also calculated the exact entropy values of the last but one layer. We can observe that the entropy values of special tokens and fixed-format values (e.g., "name" and "argument") are very low, whereas those of function names and input parameter values remain high.

These phenomenon suggests that LLM is still uncertain about its predictions in the last few layers and may inject more factual knowledge into the predictions. For special token and fix-format token, we observe that the uncertainty becomes very low from the earlier layers. This finding implies that the model is deterministic for easy-to-predict tokens at the intermediate layer and keeps the distribution of outputs almost constant at higher layers, however, it is more uncertain for difficult-to-predict key tokens and may constantly change its predictions until the final layer.

COT prompt template
You are a helpful assistant that can use tools. Before making any function call, please think step by step and explain: 1. What the user is asking for. 2. Whether you already know the answer or need external data. 3. If a function is needed, which one to use, why, and what arguments are required. Only after this reasoning, output the function call (if needed) or provide the final answer.

Table 4: COT prompt template.

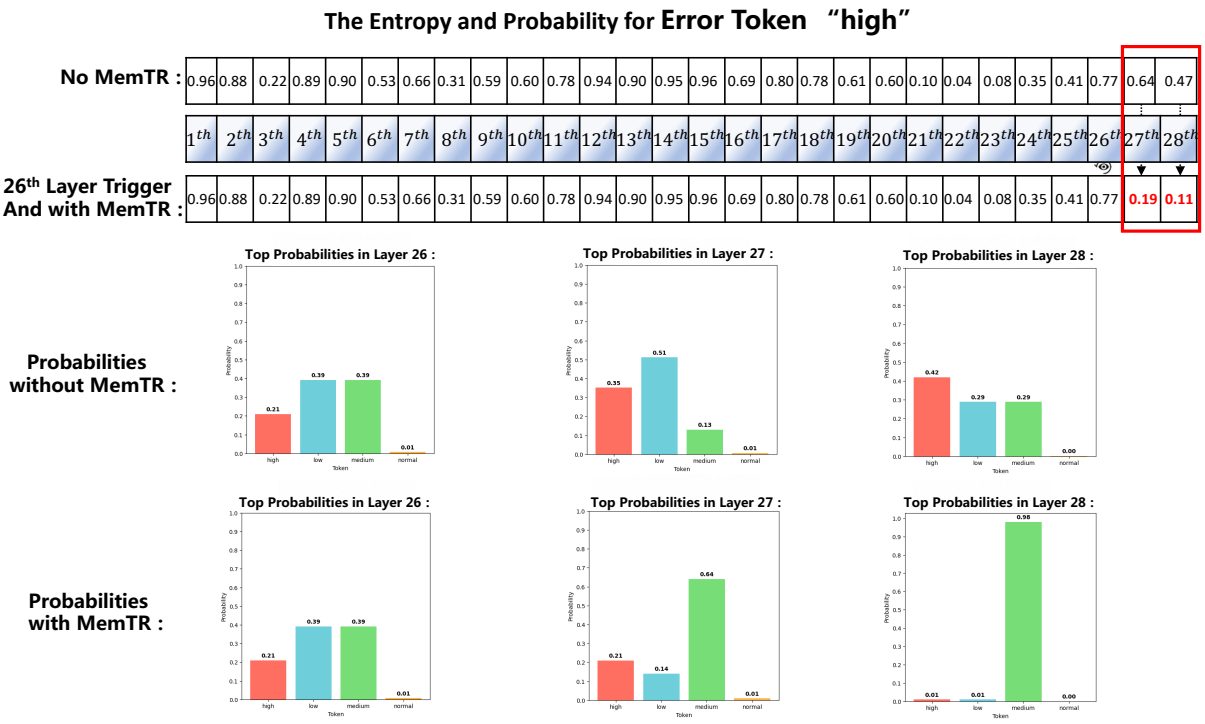

Figure 9: The entropy and probability values of the incorrect token "high" across different layers. Entropy trajectory before and after MemTR activation. We show token-level entropy across late layers for a representative tool-call token. MemTR activates at 26th layer and reduces entropy and widens the margins among the top 4 highest probability tokens.

Method	Token number	time(s)	tokens per second
Vanilla	40	2.34	17.50
MemTR	40	2.38	16.74
Vanilla	200	11.50	17.20
MemTR	200	11.62	16.68

Table 5: The time consumption of token generation on Qwen2.5-1.5B-FC

F Analysis for Computational Cost

We evaluated the time cost of our method on a single NVIDIA H20 GPU, as shown in Table 5. Overall, the total runtime increased by only about 1.7%. This is because we only performed operations on FFN modules of the specified layers, and the dimensions are all corresponding, resulting in basically no impact on memory usage and runtime.

G DETAILS FOR DATASETS

G.1 BFCL V2

BFCL V2 includes two main categories: non-live and live, which primarily consist of Python-style tool calling data. These are further subdivided into four types: Simple, Multiple, Parallel and Parallel Multiple.

- **Simple Function:** Single function evaluation contains the simplest but most commonly seen format, where the user supplies a single

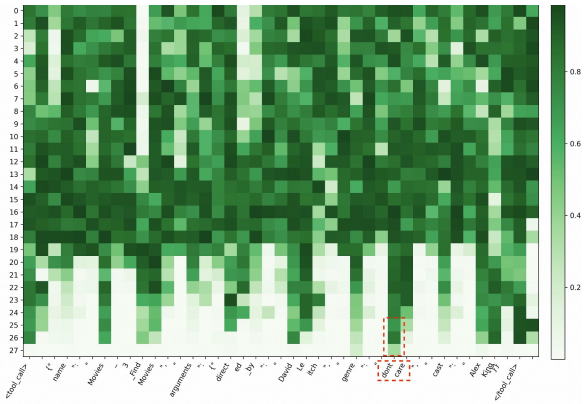

Figure 10: The uncertainty to predict the next token of varying layers. The vertical axis represents the number of layers, and the horizontal axis represents the decoded tokens. It can be observed that as the number of layers increases, the uncertainty in predicting the next token tends to decrease. The red-outlined regions showing higher uncertainty in the late layers for errors occur.

JSON function document, with one and only one function call will be invoked.

- **Multiple Function:** Multiple function category contains a user question that only invokes one function call out of 2 to 4 JSON function documentations. The model needs to be capable of selecting the best function to invoke according to user-provided context.

- **Parallel Function:** Parallel function is defined as invoking multiple function calls in parallel with one user query. The model needs to digest how many function calls need to be made and the question to model can be a single sentence or multiple sentence.

- **Parallel Multiple Function:** Parallel Multiple function is the combination of parallel function and multiple function. In other words, the model is provided with multiple function documentation, and each of the corresponding function calls will be invoked zero or more times.

G.2 ACEBench

ACEBench is a comprehensive tool-use benchmark that offers more detailed granularity. It is categorized into three main types: Normal, Special, and Agent. Atom cases involve a set of APIs that contain specific parameter types, such as booleans, enumerations, numbers, lists, and objects. The Single-turn category includes both single and parallel cases.

G.3 API-BANK

API-Bank is a dialogue-style tool calling dataset, consisting of two settings: Call and Retrieve + Call. In this dataset, the model is tasked with invoking predefined local Python tools based on the user's requirements in the dialogue. Accuracy is assessed by comparing whether the tool's returned values match the ground truth. In this work, we focus on the first step tool calling samples, disregarding further tool or retrieval feedback.

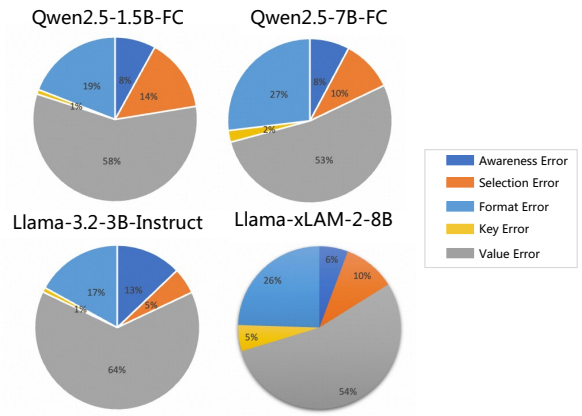

Figure 11: Proportion of error types for different LLMs on the BFCL dataset.

```
d": "live_simple_216-117-8",
esult": "<tool_call>\n{\"name\": \"Movies_3_FindMovies\", \"arguments\": {\"directed_by\": \"David Leitch\", \"genre\": \"dontcare\", \"cast\": \"Alex King\"}}\n</tool_call>"
nput_token_count": 383,
utput_token_count": 41,
atency": 0.8641681671142578,
nessage": [
  {
    "role": "user",
    "content": "Find me action movies that have David Leitch as the director and include Alex King in the cast?"
  }

unction": [
  {
    "name": "Movies_3_FindMovies",
    "description": "Retrieves a list of movies based on the director, genre, and cast specified by the user. Note that the provided function is in Python 3 syntax.",
    "parameters": {
      "type": "dict",
      "required": [

      ],
      "properties": {
        "directed_by": {
          "type": "string",
          "description": "Director of the movie. Use 'dontcare' if the director is not a specific search criterion.",
          "default": "dontcare"
        },
        "genre": {
          "type": "string",
          "description": "Genre of the movie. Use 'dontcare' to include all genres or genre is not specified by user",
          "enum": [
            "Offbeat",
            "Fantasy",
            "World",
            "Mystery",
            "Thriller",
            "Comedy",
            "Comedy-drama",
            "Horror",
            "Animation",
            "Sci-fi",
            "Cult",
            "Drama",
            "Anime",
            "Family",
            "Action",
            "dontcare"
          ],
          "default": "dontcare"
        },
        "cast": {
          "type": "string",
          "description": "Names of leading actors or actresses in the movie. Use 'dontcare' if the cast is not a specific search criterion.",
          "default": "dontcare"
        }
      }
    }
  }
```


Figure 12: Inference example.

Algorithm 2 Zero-shot MemTR with EMA-based adaptive threshold

Require: Backbone LLM with L layers, candidate layers $\mathcal{L} = \{\lceil 0.8L \rceil, \ldots, L\}$, tool memory $Z_T^{(l)}$ (cached), injection strength α, temperature τ, EMA parameters (β, λ), warm-up steps w $\zeta(\cdot)$ for vocabulary head on each layer, Top$K$ for entropy approximation.

Ensure: Generated output Y.

$Y \leftarrow \emptyset$; `tool_mode`$\leftarrow$false
$m \leftarrow 0$; $v \leftarrow 0$; `tstep`$\leftarrow 0$ EMA stats reset per example
for $s = 1, 2, \ldots$ **do**
  Run one decoding forward step (with KV-cache) to obtain $\{x_s^{(l)}\}_{l \in \mathcal{L}}$ and the final-layer state
  if generated prefix enters `<tool_call>` block **then**
    `tool_mode`$\leftarrow$true; $m \leftarrow 0$; $v \leftarrow 0$; `tstep`$\leftarrow 0$
  end if
  if `tool_mode` **then**
    `tstep`$\leftarrow$ `tstep` $+1$
    (1) compute layer-wise uncertainty and select trigger layer
    for $l \in \mathcal{L}$ **do**
      $p_s^{(l)} \leftarrow \mathrm{TopK}(\mathrm{softmax}(\zeta(x_s^{(l)})))$
      $u_s^{(l)} \leftarrow \mathrm{Entropy}(p_s^{(l)})$
    end for
    $u_s \leftarrow \max_{l \in \mathcal{L}} u_s^{(l)}$
    $l^\star \leftarrow \arg\max_{l \in \mathcal{L}} u_s^{(l)}$
    (2) update EMA and compute adaptive threshold
    $m \leftarrow \beta m + (1 - \beta) u_s$
    $v \leftarrow \beta v + (1 - \beta)(u_s - m)^2$
    $\gamma_s \leftarrow m + \lambda \sqrt{v}$
    (3)gating: only allow triggering on tool-name value or argument value
    `gate` $\leftarrow$ IsValuePosition(`generated_prefix`)
    (4) warm-up: do not trigger in first w tool-call steps
    if (`tstep` $> w$) **and** `gate` **and** $(u_s > \gamma_s)$ **then**
      $r_s \leftarrow \Delta(x_s^{(l^\star)}, Z_T^{(l^\star)})$
      $y_s \leftarrow \mathrm{FFN}^{(l^\star)}(x_s^{(l^\star)})$
      Replace FFN-output at layer $l^\star$ with $\tilde{y}_s \leftarrow (1 - \alpha) y_s + \alpha r_s$
    end if
  end if
  Decode next token from the final layer once, append to Y
  if EOS generated **or** closing tag `</tool_call>` generated **then**

    return Y
  end if
end for