
ThinkBrake: Mitigating Overthinking in Tool Reasoning

Minjae Oh^{1*} Sangjun Song^{2*} Seungkyu Lee^{1*} Sungmin Jo¹ Yohan Jo^{1†}

¹ Seoul National University ²Korea University

Abstract

Small reasoning models (SRMs) often overthink during tool use: they reach a correct tool-argument configuration, then continue reasoning and overwrite it with an incorrect final call. We diagnose overthinking via oracle rollouts that inject `</think>` at sentence boundaries. On the Berkeley Function Calling Leaderboard (BFCL), this oracle termination lifts average accuracy from 85.8% to 94.2% while reducing tokens by 80–94%, revealing substantial recoverable headroom and potentially redundant reasoning. While prior work on concise reasoning has largely targeted mathematics, tool reasoning remains underexplored. We adapt various early-termination baselines to tool use and introduce THINKBRAKE, a training-free decoding heuristic. THINKBRAKE monitors the log-probability margin between `</think>` and the current top token at sentence boundaries and triggers termination when this margin becomes small. Across BFCL’s single-turn, non-live and live splits, THINKBRAKE preserves or improves accuracy while reducing tokens up to 25%, outperforming various baselines.

1 Introduction

Recent advancements in large reasoning models (LRMs) [OpenAI, 2024, Guo et al., 2025] have demonstrated remarkable progress across domains such as mathematics, code generation, and tool usage. By allocating inference-time computation through chain-of-thought reasoning [Wei et al., 2022], LRMs exhibit capabilities well beyond their raw parameter count. Notably, even small reasoning models (SRMs) [Guo et al., 2025, Yang et al., 2025] have shown competitive performance with closed-source large models through distillation from LRMs. However, this also introduces a new failure mode: overthinking. Unlike their larger counterparts, distilled SRMs often struggle to decide when to stop reasoning [Zhang et al., 2025]. Our experiments reveal that SRMs frequently reach the correct intermediate reasoning step, only to continue deliberating and overwrite it with an incorrect final call (see Figure 1).

Tool reasoning is a particularly clear setting for observing this behavior. Effective tool use requires selecting the correct function and filling its arguments accurately through a chain of thought. Errors from overthinking are often directly visible: an SRM may initially generate the correct arguments but then revise them unnecessarily, resulting in an incorrect final tool call. This makes overthinking in tool use not only counterproductive but also easy to diagnose. However, most prior work on concise reasoning has focused on mathematical or common-sense domains, including methods such as reinforcement learning [Song and Zheng, 2025, Fatemi et al., 2025] or test-time heuristics [Li et al., 2025, Wang et al., 2025, Laaouach, 2025]. As such, little has been explored in tool reasoning, and naive adaptations of existing methods are often insufficient.

*Equal contribution.

†Corresponding author.

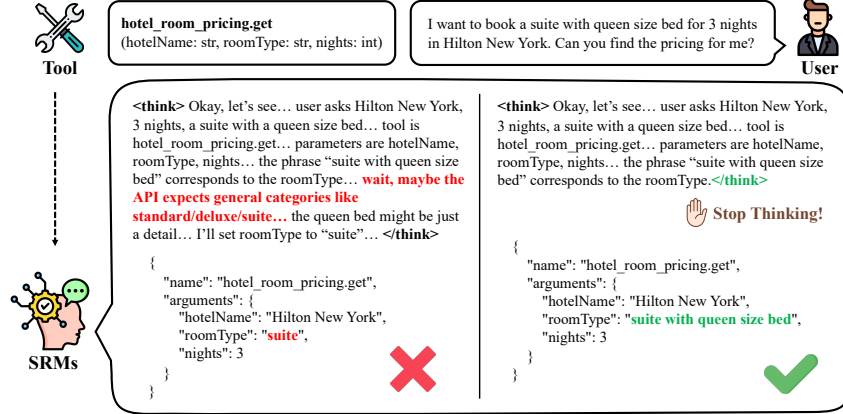


Figure 1: Overthinking in tool use. The SRM first reaches the correct tool–argument configuration (roomType=“suite with queen size bed”), but continued reasoning over-interprets the request (roomType=“suite”), yielding a wrong call. Halting at that moment by injecting `</think>` preserves the correct call.

We revisit several early-termination baselines and adapt them to the context of tool use and then propose THINKBRAKE, a lightweight, training-free decoding heuristic that terminates when the log-probability margin between `</think>` and the current top token at sentence boundaries gets low. Empirically, while prior heuristics can reduce tokens, they often degrade accuracy—especially on parallel settings—whereas THINKBRAKE reduces tokens while preserving or improving accuracy. Our contributions are threefold:

- We identify and verify overthinking in SRMs for tool use through causal oracle rollouts.
- We adapt various early-termination baselines to tool use and introduce THINKBRAKE.
- We show on BFCL that THINKBRAKE attains a better accuracy–efficiency trade-off than baselines, improving or maintaining accuracy with fewer generated tokens.

2 Tool Reasoning Analysis

We analyze the reasoning dynamics of SRM in tool use tasks, using Qwen3-4B-Thinking-2507 [Yang et al., 2025] as our SRM and evaluating on the single-turn non-live split of the Berkeley Function Calling Leaderboard (BFCL) [Patil et al., 2025].

Overthinking in tool calls. We first manually annotate incorrect outputs to identify *overthinking* failures, defined as cases where the SRM generates a valid tool and argument configuration but subsequently overrides it by switching tools or corrupting the arguments. Figure 2 shows that a substantial fraction of failures belong to this category, suggesting that the SRM often thinks of the correct tool call but fails to stop thinking there.

Early stop analysis. To test whether these errors are truly attributable to overthinking, we examine whether the SRM would have answered correctly if terminated at an appropriate earlier point. We conduct controlled rollouts with forced termination: for each reasoning trajectory, we insert a `</think>` token at the end of every sentence, forcing the SRM to stop thinking and produce an answer. We then evaluate whether the resulting tool call is correct. If a trajectory contains any such point where forced termination converts an otherwise incorrect output into a correct

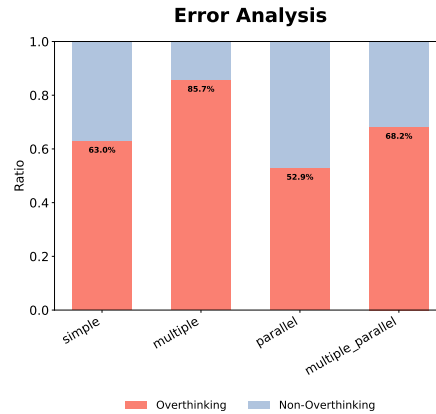


Figure 2: Proportion of overthinking vs. other errors across 119 incorrect cases over four categories in the BFCL single-turn non-live split.

Table 1: Accuracy (%) and ΔTok (token reduction, %) under oracle stopping. “Base” is Qwen-4B, and “Oracle” denotes accuracy with oracle `</think>` stopping.

Method	simple		multiple		parallel		multi-parallel		Avg.	
	Acc	ΔTok	Acc	ΔTok	Acc	ΔTok	Acc	ΔTok	Acc	ΔTok
Base	78.7	–	96.5	–	91.5	–	89.0	–	85.8	–
Oracle	91.5	-82.6%	98.5	-93.7%	96.0	-89.6%	95.5	-90.1%	94.2	-87.1%
<i>Avg. Base Token Count</i>	<i>1,154.5</i>		<i>985.0</i>		<i>1,199.9</i>		<i>1,621.2</i>		<i>1,214.1</i>	

one, we count the entire trajectory as recoverable. This yields an ‘oracle `</think>`’ accuracy—the fraction of examples where the SRM could have succeeded if it had known the exact moment to stop (see Appendix A for example). As reported in Table 1, oracle stopping achieves 94.2% accuracy on average, nearly 8.4% higher than the SRM’s original accuracy, leaving only 5.8% of cases irrecoverable. This confirms that most failures are caused by overthinking rather than an inability to solve the problem. Furthermore, the large token reduction in the oracle setting reveals that overthinking is prevalent not only in incorrect cases but also in cases where the model was already correct.

Key Takeaways

- While reasoning leads to performance gains for SRMs on the BFCL benchmark, overthinking is a key source of errors. Thus, terminating this overthinking not only reduces token usage but also promises better performance.
- Overthinking is prevalent not just in incorrect answers, but also in correct cases, meaning token usage can be further reduced for the correct cases as well.

3 THINKBRAKE

3.1 Intuition and Definition

Our hypothesis is that once the productive phase of reasoning is complete, the probability margin between continuing the chain of thought and emitting the `</think>` token decreases. However, since the model is biased towards overthinking, the probability of `</think>` does not rise high enough to actually be decoded. Therefore, we introduce THINKBRAKE. Let x be the input and $y_{<t}$ be the current reasoning chain. At each end-of-sentence boundary (e.g., a newline), let $y_t^* = \arg \max_y p_\theta(y \mid x; y_{<t})$ be the top-predicted token. We terminate when the log-probability margin between the top token and the `</think>` token is small:

$$\log \frac{p_\theta(y_t^* \mid x; y_{<t})}{p_\theta(y_{</think>} \mid x; y_{<t})} \leq \tau_{\text{threshold}}$$

We set $\tau_{\text{threshold}} = 0.25$ from searching against the Non-Live dataset. We also evaluate a probability-space variant (THINKBRAKE (prob)) that triggers when $p_\theta(y_t^*) - p_\theta(</think>) \leq \tau_{\text{threshold-prob}}$.

3.2 Baselines

We compare THINKBRAKE against previous early-termination heuristics:

- **NoWait** [Wang et al., 2025]: removes delay/filler tokens (e.g., “wait”, “hmm”; list in Appendix B.1) during decoding to stop overthinking and allow for earlier termination.
- **ThinkLess** [Li et al., 2025]: forces immediate termination by emitting `</think>` right after the initial `<think>`, minimizing the thinking process to two special tokens: `<think></think>`.
- **Confidence-of-`<tool_call>`** [Laaouach, 2025]: inspired by Halt-CoT, which monitors the entropy of answer tokens after each step and terminates once the model is confident with the answer. Because rolling out the full tool call after each step requires too many additional tokens, we monitor the confidence of the special `<tool_call>` token immediately after `</think>` and terminate once it is sufficiently confident (see Appendix B.2).

Table 2: Accuracy (%) and Δ Tok (token reduction against the base model, %) on BFCL splits. “Base Token Count” denotes the average thinking tokens, “<tool_call>” denotes monitoring the confidence of <tool_call> for early exit, THINKBRAKE(prob) denotes the probability-space variant of THINKBRAKE. The best performance is bolded, and the second best performance is underlined.

Mode	Method	simple		multiple		parallel		multi-parallel		Avg.	
		Acc	Δ Tok	Acc	Δ Tok	Acc	Δ Tok	Acc	Δ Tok	Acc	Δ Tok
Non-Live	Base	85.6	—	95.5	—	93.5	—	90.5	—	89.6	—
	NoWait	84.6	-56.6%	95.5	-61.5%	88.5	-44.4%	<u>88.5</u>	-53.5%	87.9	-54.8%
	ThinkLess	88.7	-100%	98.0	-100%	79.5	-100%	83.0	-100%	87.7	-100%
	<tool_call>	85.1	-15.7%	<u>96.5</u>	-25.5%	89.5	-39.6%	51.5	-52.5%	82.0	-28.0%
	THINKBRAKE (prob)	80.0	-25.5%	<u>96.0</u>	-51.3%	78.0	-26.9%	42.5	-33.4%	75.9	-31.2%
	THINKBRAKE	85.3	-25.6%	95.5	-17.5%	95.5	-26.8%	90.5	-28.2%	89.8	-24.9%
	Avg. Base Token Count	1,154.5		985.0		1,199.9		1,621.2		1,214.1	
Live	Base	86.4	—	82.1	—	87.5	—	<u>79.2</u>	—	82.9	—
	NoWait	87.2	-58.0%	81.1	-71.1%	<u>81.3</u>	-50.7%	75.0	-66.5%	82.2	-68.3%
	ThinkLess	77.1	-100%	76.4	-100%	37.5	-100%	45.8	-100%	75.5	-100%
	<tool_call>	<u>86.4</u>	-12.8%	<u>81.9</u>	-17.3%	75.0	-39.6%	62.5	-32.1%	82.3	-17.0%
	THINKBRAKE (prob)	81.4	-32.3%	76.9	-58.8%	62.5	-28.5%	29.2	-43.8%	76.8	-54.7%
	THINKBRAKE	85.7	-22.0%	81.4	-13.3%	87.5	-32.3%	87.5	-18.9%	<u>82.4</u>	-15.3%
	Avg. Base Token Count	1,120.4		1,792.8		1,374.0		2,389.6		1,670.0	

3.3 Main Results

We evaluate the various heuristics using Qwen3-4B-Thinking-2507 on the BFCL non-live and the more challenging live splits (Table 2).

Several patterns emerge from the main experiment. First, all heuristics reduce token generation as expected. Notably, ThinkLess eliminates all reasoning, resulting in a 100% token reduction. However, all prior heuristics lead to performance degradation, especially in the parallel and multi-parallel categories, with accuracy drops ranging from 4% to 50%. This suggests that useful reasoning is also being pruned.

Second, THINKBRAKE achieves a consistently better trade-off: it never degrades average accuracy and often improves it (notably in parallel and multi-parallel settings), while cutting tokens by up to $\sim 25\%$ on non-live and up to $\sim 15\%$ on live splits.

Third, the probability-space variant of THINKBRAKE underperforms, similar to previous heuristics. This is likely because log margins provide a more sensitive signal for THINKBRAKE, as they implicitly reflect the absolute probability of both tokens. When the two probabilities are low, achieving a small log margin is more difficult, requiring them to be more similar compared to when the probabilities are high. In this way, THINKBRAKE terminates model thinking conservatively when the model is uncertain about the </think> token and terminates mostly when it is more confident (see Appendix C). Additionally, the performance gap between the <tool_call> baseline and THINKBRAKE suggests that the </think> token provides better information for early termination than <tool_call>. This is because <tool_call> often comes with a high probability after </think>.

3.4 Additional Models

We conduct additional experiments with Qwen3-1.7B and Qwen3-8B models to further evaluate our method across different model scales (Table 3).

Similar to the 4B results, we find that THINKBRAKE achieves the best token-reduction and performance tradeoff across both smaller and larger models, consistently maintaining or improving accuracy while substantially reducing token usage.

Additionally, we observe that the 1.7B and 8B models exhibit fundamentally different reasoning dynamics compared to the 4B-thinking variant. These models generate substantially shorter thinking trajectories overall—the average base token counts are 293.9 (1.7B) and 337.0 (8B) on the non-live split, compared to 1,214.1 for the 4B-thinking model. Because these models rely less on extended reasoning, all heuristics achieve higher token reductions, with THINKBRAKE reducing tokens by $\sim 62\%$ and $\sim 52\%$ for 1.7B and 8B respectively on non-live, compared to $\sim 25\%$ for the 4B-thinking model. Furthermore, these models do not produce tokens such as ‘wait’ or ‘but’ as frequently, limiting the effectiveness of NoWait. This reveals a common problem of using heuristics that rely heavily

Table 3: Additional model evaluation. Accuracy (%) and Δ Tok (token reduction against the base model, %) on BFCL splits for Qwen3-1.7B and Qwen3-8B models. The best performance is bolded, and the second best performance is underlined.

Mode	Model	Method	simple		multiple		parallel		multi-parallel		Avg.	
			Acc	Δ Tok	Acc	Δ Tok	Acc	Δ Tok	Acc	Δ Tok	Acc	Δ Tok
Non-Live	1.7B	Base	<u>83.3</u>	—	<u>93.0</u>	—	86.0	—	<u>80.0</u>	—	<u>84.9</u>	—
		NoWait	79.3	7.3%	89.5	15.3%	80.5	3.8%	78.5	18.9%	81.1	10.0%
		ThinkLess	85.1	-100%	94.0	-100%	<u>84.5</u>	-100%	81.0	-100%	85.8	-100%
		THINKBRAKE (prob)	78.5	-66.7%	92.5	-57.5%	83.0	-69.8%	77.5	-58.9%	81.5	-64.3%
		THINKBRAKE	78.7	-66.0%	92.5	-56.6%	83.5	-66.5%	<u>80.0</u>	-54.1%	82.2	-62.4%
	Avg. Base Token Count		253.1		221.9		410.2		361.8		293.9	
	8B	Base	86.4	—	97.0	—	93.0	—	89.0	—	89.8	—
		NoWait	84.7	-19.5%	94.0	11.7%	87.5	-5.7%	84.0	-2.2%	86.7	1.5%
		ThinkLess	87.6	-100%	93.0	-100%	<u>92.5</u>	-100%	90.0	-100%	89.8	-100%
		THINKBRAKE (prob)	81.6	-60.1%	<u>95.5</u>	-48.1%	<u>92.5</u>	-62.8%	87.5	-55.0%	86.9	-57.6%
		THINKBRAKE	82.0	-54.2%	<u>95.5</u>	-44.2%	93.0	-58.1%	<u>89.0</u>	-48.3%	<u>87.5</u>	-52.1%
	Avg. Base Token Count		356.0		241.2		500.3		447.5		337.0	
Live	1.7B	Base	77.5	—	73.7	—	<u>62.5</u>	—	62.5	—	74.1	—
		NoWait	75.2	5.5%	67.6	-3.9%	<u>62.5</u>	31.7%	54.2	-11.9%	68.8	-2.1%
		ThinkLess	73.3	-100%	65.6	-100%	<u>62.5</u>	-100%	<u>58.3</u>	-100%	67.0	-100%
		THINKBRAKE (prob)	<u>76.0</u>	-65.6%	70.1	-64.9%	68.8	-67.9%	50.0	-68.3%	70.9	-65.2%
		THINKBRAKE	75.6	-64.9%	<u>70.4</u>	-63.4%	68.8	-64.8%	50.0	-67.2%	<u>71.0</u>	-63.8%
	Avg. Base Token Count		277.4		312.1		287.3		481.5		308.2	
	8B	Base	86.8	—	79.7	—	75.0	—	75.0	—	80.9	—
		NoWait	79.8	-13.6%	72.6	-13.9%	<u>62.5</u>	44.8%	<u>70.8</u>	-10.1%	73.8	-13.0%
		ThinkLess	76.0	-100%	77.4	-100%	<u>62.5</u>	-100%	66.7	-100%	76.9	-100%
		THINKBRAKE (prob)	<u>81.0</u>	-55.8%	<u>78.2</u>	-59.3%	<u>62.5</u>	-66.0%	66.7	-59.2%	<u>78.4</u>	-58.7%
		THINKBRAKE	<u>81.0</u>	-52.1%	78.1	-55.8%	<u>62.5</u>	-62.8%	<u>70.8</u>	-54.9%	<u>78.4</u>	-55.2%
	Avg. Base Token Count		327.4		369.4		420.2		462.0		363.6	

on inductive bias about specific reasoning patterns. The fact that such models rely less on long thinking trajectories also results in ThinkLess performing relatively better on these models than on 4B-thinking. Yet despite these variations in reasoning style, THINKBRAKE consistently achieves the best performance across all model sizes and settings.

4 Conclusion, Limitation and Future Work

We study overthinking in SRMs for tool use. Causal oracle rollouts show that SRMs frequently find a correct tool–argument configuration yet fail to terminate, leading to degraded final calls. Oracle termination reveals a promising headroom for accuracy with dramatically fewer tokens—indicating potential performance gain from better decoding heuristics. We introduce THINKBRAKE, a simple probability margin-based decoding heuristic for deciding termination. On BFCL, THINKBRAKE reduces tokens while preserving or improving accuracy, outperforming representative early-termination baselines.

Current limitations include evaluation on a single SRM family and one benchmark, and reliance on sentence boundary detection. Thus, future work will test robustness across models/datasets. Furthermore, we have not yet exhausted the upper performance bound and token reduction bound suggested by the oracle setting, leaving space for further improved pruning methods. Overall, our results suggest that while SRMs overthink, a well-defined decoding strategy can increase performance and reduce the token budget.

References

- Mehdi Fatemi, Banafsheh Rafiee, Mingjie Tang, and Kartik Talamadupula. Concise reasoning via reinforcement learning. *arXiv preprint arXiv:2504.05185*, 2025.
- Yichao Fu, Xuwei Wang, Yuandong Tian, and Jiawei Zhao. Deep think with confidence. *arXiv preprint arXiv:2508.15260*, 2025.
- Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

- Yassir Laaouach. HALT-cot: Model-agnostic early stopping for chain-of-thought reasoning via answer entropy. In *4th Muslims in ML Workshop co-located with ICML 2025*, 2025.
- Gengyang Li, Yifeng Gao, Yuming Li, and Yunfang Wu. Thinkless: A training-free inference-efficient method for reducing reasoning redundancy. *arXiv preprint arXiv:2505.15684*, 2025.
- OpenAI. Learning to reason with llms. Associated Press (AP), February 2024.
- 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 *International Conference on Machine Learning (ICML)*, 2025.
- Mingyang Song and Mao Zheng. Walk before you run! concise llm reasoning via reinforcement learning. *arXiv preprint arXiv:2505.21178*, 2025.
- Chenlong Wang, Yuanning Feng, Dongping Chen, Zhaoyang Chu, Ranjay Krishna, and Tianyi Zhou. Wait, we don't need to "wait"! removing thinking tokens improves reasoning efficiency. *arXiv preprint arXiv:2506.08343*, 2025.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems (NeurIPS)*, 2022.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.
- Xuechen Zhang, Zijian Huang, Chenshun Ni, Ziyang Xiong, Jiasi Chen, and Samet Oymak. Making small language models efficient reasoners: Intervention, supervision, reinforcement. *arXiv preprint arXiv:2505.07961*, 2025.

A Example of overthinking

Tables 4 and 5 present a BFCL example and the corresponding SRM response. Green highlights mark points at which terminating would yield a correct answer, whereas red highlights indicate an incorrect one. Notably, the model’s reasoning turns red after a certain point, illustrating overthinking.

Table 4: Full question and prompt.

Question
Find the area and perimeter of a circle with a radius of 5 and also find the circumference of a circle with diameter of 10.
Prompt
<p>system:</p> <p># Tools</p> <p>You may call one or more functions to assist with the user query.</p> <p>You are provided with function signatures within <tools> XML tags:</p> <pre><tools> { "name": "circle.calculate_circumference", "description": "Calculate the circumference of a circle based on the diameter.", "parameters": { "type": "dict", "properties": { "diameter": { "type": "integer", "description": "The diameter of the circle." } }, "required": ["diameter"] } }, { "name": "circle.calculate_area", "description": "Calculate the area of a circle based on the radius.", "parameters": { "type": "dict", "properties": { "radius": { "type": "integer", "description": "The radius of the circle." } }, "required": ["radius"] } }, { "name": "rectangle.calculate_perimeter", "description": "Calculate the perimeter of a rectangle based on the length and breadth.", "parameters": { "type": "dict", "properties": { "length": { "type": "integer", "description": "The length of the rectangle." }, "breadth": { "type": "integer", "description": "The breadth of the rectangle." } }, "required": ["length", "breadth"] } } </tools></pre> <p>For each function call, return a json object with function name and arguments within <tool_call> XML tags:</p> <pre><tool_call> { "name": <function-name>, "arguments": <args-json-object> } </tool_call></pre> <p>user: The above geometry question.</p>

Table 5: Qwen3-4B thinking trajectory.

<think>
Okay, let's tackle this problem. The user wants the area and perimeter of a circle with radius 5, and the circumference of a circle with diameter 10.
First, I need to check which tools are available.
There's circle.calculate_area which takes radius, and rectangle.calculate_perimeter but wait, the user is asking for the perimeter of a circle? Wait, no.
Wait, the problem says "area and perimeter of a circle".
Wait, circles don't have perimeters; they have circumferences.
Maybe the user made a mistake here.
Let me read again.
The user says: "Find the area and perimeter of a circle with a radius of 5 and also find the circumference of a circle with diameter of 10."
Hmm, perimeter for a circle is called circumference.
So maybe the user is confused.
But according to the tools provided, there's circle.calculate_circumference (for diameter), circle.calculate_area (for radius), and rectangle.calculate_perimeter (for length and breadth).
Wait, the user is asking for perimeter of a circle.
But the tools don't have a circle perimeter function.
Wait, the rectangle function is for perimeter of a rectangle, which is different.
So maybe the user intended to say circumference for the circle's perimeter.
Let me parse this.
The user says: "area and perimeter of a circle".
But in math terms, the perimeter of a circle is the circumference.
So perhaps they meant circumference.
But the problem says "perimeter", but the tool for that is circle.calculate_circumference, which takes diameter.
Wait, the tools given are:
- circle.calculate_circumference: takes diameter, returns circumference.
- circle.calculate_area: takes radius, returns area.
- rectangle.calculate_perimeter: takes length and breadth.
So the user wants:
1.
Area of a circle with radius 5: that's circle.calculate_area with radius=5.
2.
Perimeter of a circle with radius 5: but there's no such tool.
Wait, but the user might have confused perimeter with circumference.
Since the circle's perimeter is circumference, maybe they want the circumference here.
Wait, no, the problem says "area and perimeter of a circle with radius 5".
Wait, but the user also says "find the circumference of a circle with diameter 10".
Wait, maybe the user made a typo.
Let me re-read the query.
"Find the area and perimeter of a circle with a radius of 5 and also find the circumference of a circle with diameter of 10."
So, the user is asking for two things:
- For a circle with radius 5: area and perimeter.
- For a circle with diameter 10: circumference.
But in standard terms, the perimeter of a circle is the circumference.
So perhaps the user intended to say circumference for the perimeter part.
But the tools have a calculate_circumference function that takes diameter.
Wait, the circle.calculate_circumference function is based on diameter, as per its description:
"Calculate the circumference of a circle based on the diameter."
So for the circle with radius 5, to get the circumference (which the user might be calling perimeter), we can compute it using diameter.
Wait, radius 5 means diameter is 10.
So the circumference would be $\pi \times \text{diameter} = \pi \times 10$.
But the user is asking for the perimeter (circumference) of the circle with radius 5.

So maybe they want that.
However, the tools don't have a function that takes radius to calculate circumference.
The `circle.calculate_circumference` takes diameter.
So for the radius 5 circle, the diameter is 10, so we could call `calculate_circumference` with `diameter=10`.
But the user says "perimeter of a circle with radius 5", which would be the circumference, so the diameter is 10.
So perhaps the user is conflating perimeter with circumference, and wants the circumference for that circle.
Wait, but the user says "area and perimeter".
Let's break it down:

1.
Area of circle with radius 5: use `circle.calculate_area(radius=5)`.
2.
Perimeter of circle with radius 5: but the tools don't have a perimeter function for circles. The only circle-related functions are `calculate_circumference (diameter)` and `calculate_area (radius)`.
The rectangle function is for rectangles.
Because perimeter is a term for 2D shapes, but for circles it's circumference.
Assuming that the user wants:

- Area of radius 5 circle: call `calculate_area` with `radius=5`.
- Circumference of radius 5 circle (which they called perimeter): but since the tool requires diameter, we can compute diameter as 10 (since radius is 5), so call `calculate_circumference` with `diameter=10`.
- Also, circumference with diameter 10: call `calculate_circumference` with `diameter=10`.

Wait, the user says: "find the area and perimeter of a circle with a radius of 5 and also find the circumference of a circle with diameter of 10."
So that's three things:

1.
Area (radius 5)
2.
Perimeter (radius 5) → which is circumference (diameter 10)
3.
Circumference (diameter 10)

So the first two (area and perimeter) for the radius 5 circle: perimeter here would be circumference, so we need to calculate it via diameter.
But the user might have intended that.
So the steps are:

- For area: use `circle.calculate_area(radius=5)`
- For perimeter (circumference) of radius 5 circle: since radius is 5, diameter is 10, so use `circle.calculate_circumference(diameter=10)`
- For circumference with diameter 10: use `circle.calculate_circumference(diameter=10)`

Wait, but the third part is the same as the second part.
So the user wants two calculations for the same value? Let me check again.
The user says: "Find the area and perimeter of a circle with a radius of 5 and also find the circumference of a circle with diameter of 10."
So:

- Area of radius 5 circle: yes.
- Perimeter of radius 5 circle: which is circumference, so diameter 10.
- Circumference of diameter 10 circle: same as above.

So the user is asking for:

1.
Area (radius 5) → `circle.calculate_area(radius=5)`
2.
Perimeter (radius 5) → which is circumference (diameter 10) → `circle.calculate_circumference(diameter=10)`
3.
Circumference (diameter 10) → `circle.calculate_circumference(diameter=10)`

So two tool calls for the circumference part? But the user says "also find the circumference of a circle with diameter of 10", which is the same as the perimeter of the radius 5 circle.

Hmm.

Maybe the user intended to have three separate things, but in reality, it's two distinct calculations.

But according to the problem statement, I need to call the functions as per the user's request.

So the user wants:

- area of radius 5: call `circle.calculate_area` with `radius=5`.

- perimeter of radius 5 circle: but there's no perimeter function for circles.

The available circle functions have `calculate_circumference` (which is circumference) and `calculate_area`.

So perhaps the user made a mistake here, but given the tools, the perimeter of the circle (which is circumference) can be calculated via diameter.

Since radius is 5, diameter is 10, so we call `calculate_circumference` with `diameter=10`.

- circumference of diameter 10 circle: call `calculate_circumference` with `diameter=10`.

So the tool calls needed are:

- 1.

`circle.calculate_area` with `radius=5`.

- 2.

`circle.calculate_circumference` with `diameter=10`.

- 3.

`circle.calculate_circumference` with `diameter=10`.

Wait, but the third part is the same as the second.

So the user is asking for the same value twice.

But according to the problem, I need to make the tool calls as per the user's request.

Wait, the user says: "area and perimeter of a circle with a radius of 5" → two things (area and perimeter).

Then "also find the circumference of a circle with diameter of 10" → third thing.

So the second and third are two separate items.

Wait, but the perimeter of radius 5 circle is circumference with diameter 10, and the circumference with diameter 10 is the same as that.

So the user might have intended to say "perimeter" (circumference) for the radius 5 circle, and then "circumference" for diameter 10.

But since they are the same, it's one value calculated twice.

But in terms of tool calls, we need to generate two calls for the circumference: one for the perimeter (radius 5's circumference) and one for the diameter 10's circumference.

But given that the diameter 10's circumference is the same as the radius 5's circumference, it's the same calculation, but the user is asking for it twice.

Wait, perhaps the user made a mistake in the problem statement, but I have to go with what's given.

So the plan:

- Call `circle.calculate_area(radius=5)`

- Call `circle.calculate_circumference(diameter=10)` for the perimeter (since perimeter of circle is circumference, and diameter is 10 for radius 5)

- Call `circle.calculate_circumference(diameter=10)` for the circumference with diameter 10.

But the last two tool calls are identical.

However, the user says "perimeter of a circle with radius 5" (which is circumference via `diameter=10`) and "circumference of a circle with diameter 10", so they are the same.

So the tool calls would be two for the circumference part.

Wait, but the user might have intended that the perimeter is a different calculation.

Wait, no.

The perimeter of a circle is its circumference.

So I think the user is confused in terminology here.

Given the tools available, I need to make the following tool calls:

- 1.

`circle.calculate_area` with `radius 5`.

- 2.

`circle.calculate_circumference` with `diameter 10` (for the perimeter part).

3.
circle.calculate_circumference with diameter 10 (for the circumference part).
But maybe the user intended that the "perimeter" is a mistake and they just want the circumference for the radius 5 circle, which would be calculated via diameter.
But according to the problem statement, I have to follow the user's request as given.
So the answer would require three tool calls? Wait, no.
Let me recheck.
The user says: "Find the area and perimeter of a circle with a radius of 5 and also find the circumference of a circle with diameter of 10."
So two parts:
A.
Area and perimeter of a circle (radius 5)
B.
Circumference of a circle (diameter 10)
Part A: area uses radius, perimeter (which for circle is circumference) uses diameter (10).
So for part A, we need two tool calls: calculate_area (radius) and calculate_circumference (diameter=10).
Part B: circumference (diameter 10) → calculate_circumference (diameter=10).
Wait, but part B is the same as the circumference part of A.
So the user is asking for:
- Area (radius 5)
- Perimeter (radius 5) → circumference (diameter 10)
- Circumference (diameter 10)
So three things, but two of them are the same calculation.
But the user says "area and perimeter" (two things) and "also find the circumference" (third thing).
So three tool calls.
But in reality, the perimeter of the radius 5 circle is the same as circumference of diameter 10 circle.
So the user is asking for two different things that result in the same value.
But the problem says to call the functions as per the user's request.
So the tool calls would be:
1.
circle.calculate_area(radius=5)
2.
circle.calculate_circumference(diameter=10) [for perimeter of radius 5 circle]
3.
circle.calculate_circumference(diameter=10) [for circumference with diameter 10]
Wait, but the user says "perimeter of a circle with radius of 5" and "circumference of a circle with diameter of 10".
So two separate requests that map to the same calculation.
But according to the tools, we need to make two calls for the circumference.
But maybe the user intended that the perimeter part is a mistake, and they meant circumference.
However, given the tools available, I have to work with what's there.
So the correct tool calls are:
- For area: circle.calculate_area with radius=5.
- For perimeter (circle's circumference): since perimeter of a circle is circumference, and the user gives radius=5, so diameter=10.
So call circle.calculate_circumference(diameter=10).
- For circumference (diameter=10): call circle.calculate_circumference(diameter=10).
Therefore, three tool calls? Wait, no.
Wait, the user says "Find the area and perimeter of a circle with a radius of 5 and also find the circumference of a circle with diameter of 10."
So it's:
- Area of radius 5 circle.
- Perimeter of radius 5 circle.
- Circumference of diameter 10 circle.
So three things.

But in reality, the perimeter of the circle (radius 5) is the same as circumference of diameter 10. So two tool calls for the circumference (but the user asks for it twice).
Wait, but the problem says "the user query", so the assistant needs to generate the correct tool calls.
Let me think in terms of the provided functions.
The tools are:

1.
`circle.calculate_circumference(diameter)`
2.
`circle.calculate_area(radius)`
3.
`rectangle.calculate_perimeter(length, breadth)`

So for the user's request:

- Area of radius 5: call `calculate_area` with `radius=5`.
- Perimeter of a circle with radius 5: but the rectangle function is for rectangles.

There's no circle perimeter tool.
Wait, the `circle.calculate_circumference` is for circumference based on diameter.
So the perimeter here (which is circumference) would be calculated using diameter.
So for radius 5, diameter is 10, so call `calculate_circumference` with `diameter=10`.

- Circumference of diameter 10: call `calculate_circumference` with `diameter=10`.

So the tool calls are:

1.
`circle.calculate_area(radius=5)`
2.
`circle.calculate_circumference(diameter=10)`
3.
`circle.calculate_circumference(diameter=10)`

But maybe the user intended that "perimeter" is a mistake and they just want circumference for radius 5, but according to the problem statement, we have to go with what's written.
Wait, but the user says "perimeter of a circle", which is a misnomer.
The correct term is circumference.
But given that, the assistant has to map the user's request to the available functions.
So the answer requires two tool calls: one for area (radius 5), and two for circumference (diameter 10) because the user asks for perimeter (which is circumference) of radius 5 (diameter 10) and circumference of diameter 10.
Wait, no.
The user says "Find the area and perimeter of a circle with a radius of 5" → two things (area and perimeter).
Then "also find the circumference of a circle with diameter of 10" → third thing.
So three tool calls.
But the first two (area and perimeter) require two tool calls, and the third requires another.
But the perimeter of the circle (radius 5) is calculated via diameter 10 (`circle.calculate_circumference`), and the circumference (diameter 10) is the same function call.
So the tool calls are:

- `circle.calculate_area` with radius 5
- `circle.calculate_circumference` with diameter 10 (for perimeter)
- `circle.calculate_circumference` with diameter 10 (for circumference)

But the assistant's job is to generate the tool calls.
So three tool calls? Or is the perimeter part a mistake and the user just meant circumference?
Wait, the `rectangle.calculate_perimeter` is for rectangles, so that's not relevant here.
The user is talking about circles, so we ignore the rectangle tool.
So, the correct tool calls are:

1.
`circle.calculate_area(radius=5)`
2.
`circle.calculate_circumference(diameter=10)`
- 3.

```
circle.calculate_circumference(diameter=10)
```

But in reality, the user is asking for two separate values: perimeter (which is circumference) for radius 5 (diameter 10) and circumference for diameter 10.

So they are the same calculation, but the user is asking for it twice.

But according to the problem statement, the assistant should generate the tool calls as per the user's request.

So even if the user is asking for the same value twice, we have to generate the tool calls.

Wait, but maybe the user made a typo and the perimeter part is a mistake.

For example, maybe they meant "circumference" instead of "perimeter".

But given the tools, we have to proceed.

Let me check the problem statement again: "Find the area and perimeter of a circle with a radius of 5 and also find the circumference of a circle with diameter of 10."

So, to rephrase:

- area (circle, radius 5)

- perimeter (circle, radius 5) → which is circumference (diameter 10)

- circumference (circle, diameter 10)

So three requests, but the second and third are the same calculation.

But the problem says "you may call one or more functions".

So the assistant should generate the necessary tool calls.

The first function call is for area: `circle.calculate_area(radius=5)`.

The second and third are both `circle.calculate_circumference(diameter=10)`.

So three tool calls? Or is the user's "perimeter" a mistake and they just want the circumference for the radius 5 circle, which is the same as the diameter 10 circumference.

But the user says "perimeter" and "circumference" as two separate terms, but in reality, they are the same for a circle.

Hmm.

Given the time I've spent, I'll proceed to generate the two tool calls for the circumference (since the user asks for perimeter which is circumference and the circumference with diameter 10), but wait, the user says "find the area and perimeter of a circle with a radius of 5" → two things, and "also find the circumference of a circle with diameter of 10" → third thing.

So the assistant should generate three tool calls.

But wait, no.

Let's see:

- For the area of radius 5: one tool call.

- For the perimeter of radius 5 circle: since perimeter of a circle is circumference, which is calculated via diameter.

So diameter is .

B Details for Baselines

B.1 NoWait

We use the following list in Table 6 for NoWait.

Table 6: Keyword list for NoWait.

Keywords
wait, hmm, hmmm, but, however, check, verify, alternatively

B.2 Confidence-of-`<tool_call>`

We calculate the confidence of the `<tool_call>` token, following Fu et al. [2025] as follows:

$$C = -\frac{1}{k} \sum_{j=1}^k \log p^{(j)}.$$

We use top-k=20 token log probabilities, and a threshold of 20.

C Effect of log on THINKBRAKE

As shown in Table 2, raw probability gaps are insufficient for THINKBRAKE. A meaningful log-probability gap only arises when both competing probabilities are relatively high (see Figure 3), indicating that the model is genuinely confident at that step. If we used a raw probability gap, both $\Delta P1$ and $\Delta P2$ would trigger early termination; in contrast, using a log-probability gap triggers only for $\Delta P1$ —where both the top token and `</think>` have high probabilities—signaling confidence in stopping the reasoning.

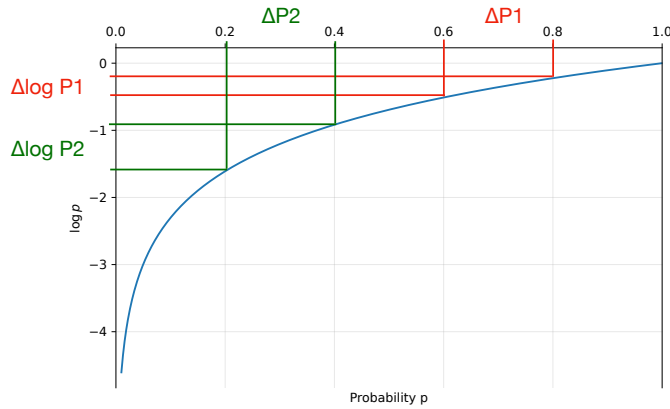


Figure 3: Log-probability plot with two illustrative cases, $\Delta P1$ and $\Delta P2$. Here, $\Delta P1$ denotes a scenario where the most likely token has high probability (0.8), and $\Delta P2$ denotes a scenario where the most likely token has lower probability (0.4), while the raw gap is the same (i.e., $\Delta P1 = \Delta P2$).