LLM Agents Already Know When to Call Tools - Even Without Reasoning

Chung-En Sun **Linbo Liu** **Ge Yan*** **Zimo Wang*** **Tsui-Wei Weng**
University of California, San Diego
{cesun, linbo, geyan, zimowang, lweng}@ucsd.edu

Abstract

Tool-augmented LLM agents tend to call tools indiscriminately, even when the model can answer directly. Each unnecessary call wastes API fees and latency, yet no existing benchmark systematically studies *when* a tool call is actually needed. We propose WHEN2TOOL, a benchmark of 18 environments (15 single-hop, 3 multi-hop) spanning three categories of tool necessity — computational scale, knowledge boundaries, and execution reliability — each with controlled difficulty levels that create a clear decision boundary between tool-necessary and tool-unnecessary tasks. We evaluate two families of training-free baselines: Prompt-only (varying the prompt to discourage unnecessary calls) and Reason-then-Act (requiring the model to reason about tool necessity before acting). Both provide limited control: Prompt-only suppresses necessary calls alongside unnecessary ones, and Reason-then-Act still incurs a disproportionate accuracy cost on hard tasks. To understand *why* these baselines fail, we probe the models' hidden states and find that tool necessity is linearly decodable from the pre-generation representation with AUROC 0.89–0.96 across six models, substantially exceeding the model's own verbalized reasoning. This reveals that *models already know when tools are needed, but fail to act on this knowledge during generation*. Building on this finding, we propose PROBE&PREFILL, which uses a lightweight linear probe to read the hidden-state signal and prefills the model's response with a steering sentence. Across all models tested, PROBE&PREFILL reduces tool calls by 38% with only 1.7% accuracy loss, while the best baseline at comparable accuracy only reduces 4.2% of tool calls, or achieves a similar tool call reduction but incurs a 5× higher accuracy loss. On the real-world Search-o1 agentic benchmark, PROBE&PREFILL reduces API calls by up to 56% without accuracy degradation. Our code is available at: https://github.com/Trustworthy-ML-Lab/when2tool

1 Introduction

Large language models have demonstrated remarkable capabilities across a wide range of tasks, such as deep research [Shao et al., 2025, OpenAi, 2025], software engineering [Jimenez et al., 2023, Liu et al., 2025, Merrill et al., 2026], mathematical reasoning [Sun et al., 2025a], search and retrieval [Jin et al., 2025], and user interaction [Yao et al., 2024, Barres et al., 2025]. Recently, the agentic paradigm has further extended these capabilities by equipping LLMs with external tools for complex planning, multi-step problem solving, and real-world interactions [Schick et al., 2023, Qin et al., 2023, Patil et al., 2024]. However, current tool-augmented agents tend to call tools indiscriminately, even when the model already possesses the ability to answer directly. A central design question in these systems is therefore: *when should the model call a tool versus solve the task directly?* In many cases, tool calls are unnecessary: an agent does not need to launch a web search or RAG pipeline to answer "what

*Equal contribution.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

year did humans land on the Moon?. Each unnecessary call wastes API fees, costs that compound rapidly when an agent makes dozens of decisions per session at deployment scale.

Recent work has begun to address tool-call efficiency, but existing approaches either target a different problem or bypass the question of *why* models overcall. Wu et al. [2025] reduces redundant calls by jointly refining agent instructions and tool descriptions, but focus on improving calls that are already needed, not on deciding whether a call is necessary. Xu et al. [2025] study when to skip tools entirely, but evaluate with oracle tools that simply return the correct answer upon invocation, and rely on SFT to modify behavior without understanding why the model overcalls. This leaves a fundamental question unanswered: *do models overcall because they lack the information to decide, or because they fail to act on information they already have*?

Motivated by these limitations, we propose WHEN2TOOL (Section 2), a benchmark designed to study the tool-call decision in a setting that closely mirrors real-world agent deployments. WHEN2TOOL comprises 18 environments (15 single-hop and 3 multi-hop), each providing tools that the model must invoke with correctly formatted arguments and whose responses require parsing, matching the interaction pattern of real APIs. We identify three categories of situations where an agent must decide whether to use a tool, covering what we believe are the major real-world scenarios: (1) *"Can I compute this?"*: the model understands the operation, but the operands may exceed what it can compute reliably (e.g. $12 + 7$ is trivial; trillion-scale multiplication is not); (2) *"Do I know this?"*: the answer may or may not exist in the model's parameters (e.g. the capital of France is common knowledge; an obscure historical date may not be); and (3) *"Can I execute this reliably?"*: the model knows the rules, but mentally tracing many sequential steps is error-prone (e.g. predicting `print(2+3)` is easy; tracing deep recursion might not). Each environment has three difficulty levels: *easy* (most models can reliably solve without tools), *medium* (the decision boundary where models sometimes succeed and sometimes fail), and *hard* (most models cannot succeed without tools). Using WHEN2TOOL, we systematically evaluate two training-free baselines: Prompt-only, which varies the system prompt to discourage unnecessary calls, and Reason-then-Act, which asks the model to explicitly reason about tool necessity before acting. We find that both provide limited control over tool-call decisions, with hard tasks paying a high accuracy cost for each saved call (Section 3).

Given the limited ability of prompting and reasoning, we ask a deeper question: *does the model internally encode information about whether a tool is needed?* To investigate, we probe the model's hidden representations (Section 4). We extract the hidden state at the last input token and train a simple linear classifier to predict whether a tool call is necessary. Surprisingly, the probe achieves AUROC above 0.9 across models. This reveals that **the model already encodes a clean signal about tool necessity in its hidden state**, but the generation process fails to translate it into calibrated decisions. Notably, even for models that completely fail under Reason-then-Act, the probe still extracts a strong signal, demonstrating that representation-level knowledge exists independently of the model's ability to express it through text.

Building on this finding, we propose **PROBE&PREFILL** (Section 5). We train a lightweight linear probe on the hidden states with binary tool-necessity labels. At inference time, the probe reads the hidden state and prefills the model's response with a short steering sentence (e.g., "I can solve this directly without using a tool" or "I need to use a tool for this question"). The model then continues generating from this prefill. By adjusting the probe's decision threshold, PROBE&PREFILL provides smooth, fine-grained control over the accuracy–efficiency tradeoff. Across all tested models, PROBE&PREFILL outperforms every Prompt-only and Reason-then-Act baseline, and transfers well between tasks. It reduces unnecessary tool calls while preserving accuracy on hard tasks, requiring only a simple linear prediction and no additional reasoning from the model.

Our contributions are three-fold, progressing systematically from benchmark design and failure analysis, to a proposed mitigation method:

- **Benchmark:** We design **WHEN2TOOL**, the first benchmark for studying tool-call decisions. It comprises 18 environments (15 single-hop, 3 multi-hop) across three categories of tool necessity with controlled difficulty levels, totaling 1,080 training and 2,700 test tasks.

- **Failure Analysis:** We evaluate Prompt-only and Reason-then-Act baselines, revealing that both provide limited and coarse control over tool-call decisions, with hard tasks paying a disproportionate accuracy cost for each saved call.

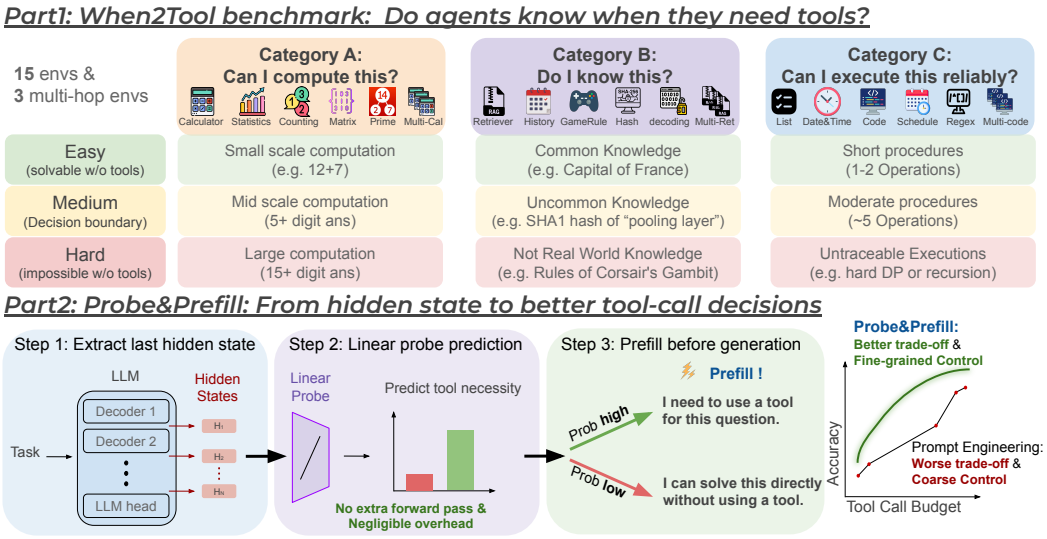

Figure 1: Overview. **Part 1**: We design WHEN2TOOL for studying whether LLM agents know when they need tools, spanning 15 single and 3 multi-hop environments across three categories, each with three difficulty levels. **Part 2**: PROBE&PREFILL reads the model's hidden state via a linear probe and prefills a steering sentence to guide the tool-call decision, achieving better tradeoffs.

- **Discovery & Mitigation:** We probe pre-generation hidden states and find that tool necessity is linearly decodable. We therefore exploit this signal and propose **PROBE&PREFILL**, a lightweight method (<1ms overhead) that prefills the model's response based on the probe prediction. **PROBE&PREFILL** reduces tool calls by 38% with only 1.7% accuracy loss, while the best baselines either reduce tool calls by only 4.2% at comparable accuracy (i.e. 9× less efficient) or suffer 5× more accuracy loss for a similar reduction. Furthermore, our method generalizes to real-world agentic search, reducing API calls by up to 56% on the Search-o1 agentic benchmark [Li et al., 2025a].

2 WHEN2TOOL: A Benchmark for Tool-Call Decisions

To systematically study how LLM agents decide whether to call a tool, we build WHEN2TOOL, a controlled benchmark of 18 tool-use environments (15 single-hop and 3 multi-hop) spanning three categories of self-assessment. Existing benchmarks evaluate *whether* models can use tools correctly [Zhuang et al., 2023, Li et al., 2023b], assuming every task requires a tool. WHEN2TOOL instead tests *whether the model knows when a tool is needed*: tasks range from those the model generally can solve directly to those that are impossible without a tool. The benchmark includes 15 single-hop environments and 3 multi-hop environments (requiring a chain of 3 dependent tool calls where each step's output is the next step's input). Table 1 summarizes the key differences.

2.1 Three categories of tool necessity

In real-world agent deployments, the decision of whether to call a tool generally falls into three categories, each requiring the model to assess a different aspect of its own capability. We design 5 single-hop environments and 1 multi-hop environment for each category.

Category A: "Can I compute this?" (Computational scale.) These environments test whether the model can assess the limits of its own mental arithmetic. The model understands the operation in every case; the question is whether the numbers involved exceed what it can compute reliably. At easy difficulty, the scale is small enough that the model can compute directly (e.g., 235×48); at hard difficulty, the operands grow to a scale that guarantees failure without a tool (e.g., trillion-scale arithmetic, 5×5 determinants, $C(80, 40)$). A well-calibrated agent should recognize the boundary and call the tool only when scale demands it.

Category B: "Do I know this?" (Knowledge boundary.) These environments test whether the model can assess what information exists in its own parameters. The model must judge whether it possesses the factual knowledge needed to answer, a fundamentally different self-assessment from computational feasibility. At easy difficulty, tasks query widely known facts (e.g., the capital of France); at hard difficulty, we use fictional entities, invented events, and custom algorithms that cannot exist in any training data, guaranteeing that the model must consult the tool to get the answer. A well-calibrated agent should recognize the boundary between what it knows and what it does not.

Category C: "Can I execute this reliably?" (Execution tracking.) These environments test whether the model can assess its own reliability when tracing sequential procedures. Unlike Category A or B, the model knows the rules of execution, and has all the information needed to produce the answer. The question is whether it can execute the steps faithfully without accumulating errors. At easy difficulty, the procedure is short enough to trace mentally (e.g., predicting the output of `print(2+3)`, checking two meetings for overlap); at hard difficulty, the procedure involves enough steps that mental execution becomes error-prone (e.g., tracing a 20-iteration dynamic programming algorithm, finding free slots across 10+ meetings). A well-calibrated agent should recognize when the execution trace exceeds its reliable tracking capacity.

2.2 Difficulty as the decision boundary

Each environment has three difficulty levels that control where the tool-call decision boundary falls:

- **Easy**: The model can mostly solve without tools. These tasks test whether the model *overcalls*, invoking tools when it does not need them.
- **Medium**: The decision boundary where most models sometimes succeed and sometimes fail without tools. This is where calibrated decision-making matters most.
- **Hard**: The model almost cannot succeed without tools. These tasks test whether the model can recognize the limits of its own capability and call the tool when it genuinely needs one.

We validate these difficulty assignments empirically by running all tasks in a no-tool setting where the model is forced to answer directly (Table 6, Appendix A).

In total, WHEN2TOOL contains 1,080 training tasks and 2,700 test tasks: 900 train / 2,250 test for the 15 single-hop environments, plus 180 train / 450 test for the 3 multi-hop environments (each with 3 difficulties $\times$ 20 or 50 tasks per split). Full environment details, tool descriptions, and example tasks at each difficulty level are provided in Appendix A.

Table 1: Comparison with existing tool-use benchmarks. WHEN2TOOL is the first to evaluate the tool-call decision with controlled difficulty, multi-hop tasks, and zero API cost.

Benchmark	Tool-call decision	Difficulty levels	Multi-hop tasks	Realistic tool I/O	Zero API cost
Toolformer [Schick et al., 2023]	✗	✗	✗	✗	✓
ToolLLM [Qin et al., 2023]	✗	✗	✓	✓	✗
Gorilla [Patil et al., 2024]	✗	✗	✗	✓	✗
API-Bank [Li et al., 2023b]	✗	✗	✓	✓	✗
ToolQA [Zhuang et al., 2023]	✗	✗	✓	✓	✗
BFCL [Patil et al., 2025]	✗	✗	✓	✓	✗
Xu et al. [2025]	✓	✗	✗	✗	✓
WHEN2TOOL (ours)	✓	✓	✓	✓	✓

3 Failure Analysis: The Limits of Prompting and Explicit Reasoning

With WHEN2TOOL in place, we systematically test whether models can calibrate their tool usage through the two most natural training-free approaches: varying the prompt (Prompt-only) and asking the model to explicitly reason about tool necessity before acting (Reason-then-Act). Surprisingly, we found that these approaches based on prompt engineering and reasoning *fail* to selectively reduce unnecessary tool calls as we show in the experiments.

Table 2: Accuracy cost per saved call ($\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$) when switching from Default ($\star$) to Sparse (S). All changes are relative to Default (Prompt-only). More negative $\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$ means each saved call costs more accuracy. Hard tasks pay a disproportionate price.

		Qwen3-4B-Inst.			Qwen3-14B			Llama-3.3-70B		
Mode	Difficulty	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$
Prompt-only	Easy	−14.5	−0.84	−17.3	−8.8	−0.59	−14.9	+1.6	−0.51	+3.2
	Medium	−20.7	−0.86	−24.1	−12.9	−0.53	−24.3	+2.0	−0.41	+4.8
	Hard	−20.3	−0.48	**−42.4**	−27.3	−0.47	**−58.4**	−0.2	−0.34	**−0.5**
Reason-then-Act	Easy	−14.5	−0.86	−16.9	−4.4	−0.67	−6.6	−4.8	−1.98	−2.4
	Medium	−22.4	−0.90	−24.8	−10.4	−0.62	−16.8	−18.9	−1.87	−10.1
	Hard	−13.0	−0.35	**−36.6**	−9.7	−0.28	**−34.7**	−63.3	−1.99	**−31.7**

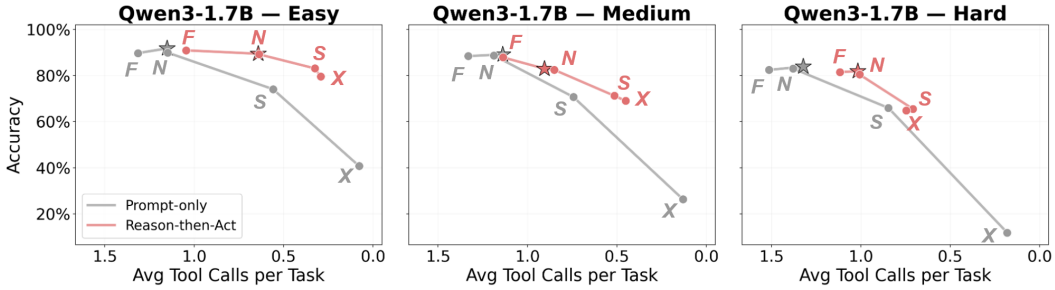

Figure 2: Accuracy vs. Avg tool calls per difficulty for Qwen3-1.7B. $\star$=Default, F=Force, N=Necessary, S=Sparse, X=No Tool. Reason-then-Act (red) partially shifts the tradeoff, reducing unnecessary easy-task calls, but still produces negative efficiency on hard tasks. Lines fold non-monotonically, reflecting coarse prompt control.

3.1 Experimental setup

We evaluate six models spanning two families: Qwen3-1.7B/4B/14B/32B and Llama-3.1-8B/3.3-70B. All experiments are run 3 times with different random seeds and we report mean results.

Prompt-only baselines. We test five prompt modes spanning the full range of tool-use instructions: **Force (F)**, "tool use is mandatory"; **Default ($\star$)**, no explicit requirements; **Necessary (N)**, "only if necessary"; **Sparse (S)**, "expensive, use sparingly"; and **No Tool (X)**, "do not use any tools."

Reason-then-act baselines. In addition to prompt-only control, we evaluate a stronger baseline inspired by the think-before-act paradigm from ReAct [Yao et al., 2022] and Reflexion [Shinn et al., 2023]. Before making a tool-call decision, the model is instructed to first reason about whether it can solve the task directly or needs a tool, then act on its own assessment. We apply this reasoning step to each of the same five prompt modes (Force, Default, Necessary, Sparse, No Tool).

3.2 Key findings

Finding 1: Models default to tool overuse. Under the Default ($\star$) setting in Prompt-only baselines, models make 2,100–4,400 total tool calls across the 2,250-task single-hop test set, more than one calls per task. Even on easy tasks, Qwen3-1.7B makes 864 tool calls out of 750 easy tasks, and Llama-3.3-70B makes 1,482. The model's default behavior is "tools are available, therefore use them," even when the task is simple enough to solve directly.

Finding 2: Prompt engineering reduces tool calls indiscriminately, and hard tasks pay a disproportionate price. Prompts that discourage tool usage reduce calls across *all* difficulty levels, including hard tasks where tools are genuinely needed. The reduction is far too indiscriminate: hard tasks lose substantially more accuracy per saved call than easy tasks. We quantify this with the *accuracy cost per saved call*, $\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$: how much accuracy is lost for each tool call eliminated. More negative means each saved call is more costly. Table 2 shows this cost when moving from Default ($\star$)

to Sparse (S). On Qwen3-4B-Instruct, the cost is -17.3 on easy but reaches -42.4 on hard, meaning hard tasks lose $2.5\times$ more accuracy per saved call. This pattern holds across model sizes.

Finding 3: Reason-then-Act only partially helps, but with additional cost. Reasoning before decisions partially mitigates this problem: the accuracy cost per saved call improves on easy tasks (e.g., Qwen3-14B easy improves from -14.9 to -6.6 in Table 2). Figure 2 shows Reason-then-Act (red lines) sits closer to the upper-right region on easy tasks, indicating that reasoning does reduce some unnecessary calls. However, reasoning still carries a high accuracy cost per saved call on hard tasks (e.g., -34.7 for Qwen3-14B), since it also suppresses tool calls where they are genuinely needed. Critically, this partial improvement comes at a cost: reasoning requires the model to generate additional tokens, increasing the generation overhead. Moreover, reasoning is model-dependent: on Llama-3.1-8B, accuracy drops from 79.5% to 31.2%; on Llama-3.3-70B, from 83.1% to 47.9%, as the model narrates its intent to call tools but never produces a valid invocation, resulting in near-zero tool calls (see Table 8 in Appendix B for full details).

Finding 4: Prompt engineering cannot precisely control the accuracy–tool-call tradeoff. In practice, a user may want to set a tool-call budget and maximize accuracy under that budget. Figure 2 shows that neither Prompt-Only nor Reason-then-Act can achieve this: each prompt mode provides a single, fixed operating point, with no way to smoothly adjust the tradeoff. Furthermore, many of these operating points are nearly indistinguishable: instructing the model to use tools "only if necessary" (N) produces almost identical behavior to the neutral Default ($\star$), while in Reason-then-Act mode, Sparse (S) and No Tool (X) collapse to nearly the same point. The two baselines offer only a few effective operating points, making it impossible to precisely target a desired budget.

These findings establish that prompt-level control over tool-call decisions is limited, coarse, and unreliable across models. The models appear to lack the ability to translate task understanding into calibrated tool-call decisions through the generation process. This raises the question: is the model *unable* to assess tool necessity, or does it know internally but fail to act on it? We investigate this in the next section.

4 Probing Analysis: Decoding Implicit Tool Necessity

We now investigate whether models encode tool-necessity information internally, even when they fail to act on it during generation. Interestingly, we find that *tool necessity is already encoded in hidden states* as shown in the experiments.

Setup. For each task, we first collect a binary label: we force the model to answer without tool access, and label tasks where it succeeds as tool-unnecessary ($y{=}0$) and tasks where it fails as tool-necessary ($y{=}1$). We then extract hidden states by running a single forward pass and taking the hidden state at the last token position across all layers. Finally, we concatenate all-layer features and train an L2-regularized logistic regression to predict tool necessity. The probe trains on 900 training examples and is evaluated on 2,250 held-out test tasks. Training takes seconds on CPU.

Tool necessity is linearly decodable. The probe achieves AUROC 0.89–0.96 across all six tested models (Table 3), confirming that tool necessity is consistently encoded in pre-generation hidden states regardless of model family or size. Even the smallest model carries a strong signal, while larger models reach 0.95+. The per-difficulty breakdown shows strong performance across all levels,

Table 3: AUROC and accuracy for predicting tool necessity from pre-generation hidden states. All probes achieve high AUROC, confirming that tool necessity is consistently encoded in models.

			AUROC by difficulty		
Model	AUROC	Acc	Easy	Med	Hard
Qwen3-1.7B	0.894	0.847	0.864	0.831	0.904
Qwen3-4B-Inst.	0.948	0.877	0.933	0.906	0.948
Llama-3.1-8B-Inst.	0.927	0.849	0.892	0.867	0.884
Qwen3-14B	0.957	0.892	0.955	0.907	0.941
Qwen3-32B	0.952	0.885	0.951	0.903	0.939
Llama-3.3-70B-Inst.	0.936	0.872	0.906	0.849	0.956

with medium tasks being the most challenging, consistent with medium being the decision boundary where the model is uncertain.

The signal exists even when generation fails. The most striking evidence comes from the Llama models. As discussed in Section 3, Reason-then-Act completely breaks tool calling on Llama-3.1-8B (79.5% $\rightarrow$ 31.2%) and Llama-3.3-70B (83.1% $\rightarrow$ 47.9%). Yet the linear probe still achieves AUROC above 0.9; the information about tool necessity is clearly present in the representation, even though the model is entirely unable to express it during generation.

Based on these findings, our next step is to use this signal to directly steer the model's tool-call behavior. In the next section, we show how injecting a short steering sentence into the model's output can translate this hidden knowledge into better tool-call decisions.

5 PROBE&PREFILL: Turning Hidden Knowledge into Better Decisions

In this section, we propose PROBE&PREFILL, a lightweight inference-time method that uses the probe's prediction to prefill the model's response with a short steering sentence, guiding the model to either solve directly or call a tool. This translates the internal signal identified in Section 4 into better tool-call decisions, requiring no model fine-tuning and no reasoning overhead.

5.1 Method

PROBE&PREFILL operates in three steps at inference time:

Step 1: Extract last hidden states. Given a task prompt, we run a single forward pass over the input tokens and extract the hidden states at the last token position. This is the standard prompt-encoding step that every LLM performs to get KV cache before autoregressive decoding begins, so hidden state extraction adds no additional forward passes.

Step 2: Linear probe prediction. We apply the trained linear probe to all-layer hidden states, producing a probability p. A threshold τ converts this into a binary decision: if $p < \tau$, the task is predicted to be solvable without tools; otherwise, a tool call is predicted to be necessary. The threshold τ provides a single knob for controlling the accuracy–efficiency tradeoff: lower τ skips more tool calls (saves cost, risks missing necessary ones); higher τ preserves more tool calls (higher accuracy, fewer savings).

Step 3: Prefill before generation. Based on the probe's prediction, we prepend a short steering sentence to the beginning of the model's response:

- If $p < \tau$ (tool unnecessary): "`I can solve this directly without using a tool.`"
- If $p \geq \tau$ (tool necessary): "`I need to use a tool for this question.`"

The model then continues generating from this prefill, producing either a direct answer or a tool call. This *soft* prefill allows the model to override the suggestion if its own assessment disagrees. We also evaluate a *hard* prefill mode that forces the output format (`\boxed{` for direct answers, tool-call JSON for tool use), leaving no room for the model to deviate (Table 12, Appendix E.1).

5.2 Main results

Figures 3 and 4 show the accuracy vs. tool-call tradeoff across six models.

PROBE&PREFILL outperforms the Prompt-only baseline. Compared to the Prompt-only baseline (gray), PROBE&PREFILL (green) is strictly better on Qwen models (Figure 3): it achieves higher accuracy at every tool-call budget, or equivalently, fewer tool calls at every accuracy level. Moreover, by sweeping the threshold τ, PROBE&PREFILL traces a smooth tradeoff curve, providing fine-grained control over the operating point. In contrast, prompt engineering offers only a handful of discrete, fixed operating points with no way to interpolate between them. For Llama models (Figure 4), soft prefill is partially ignored due to the bad instruction following; hard prefill restores full tradeoff control by forcing the output format.

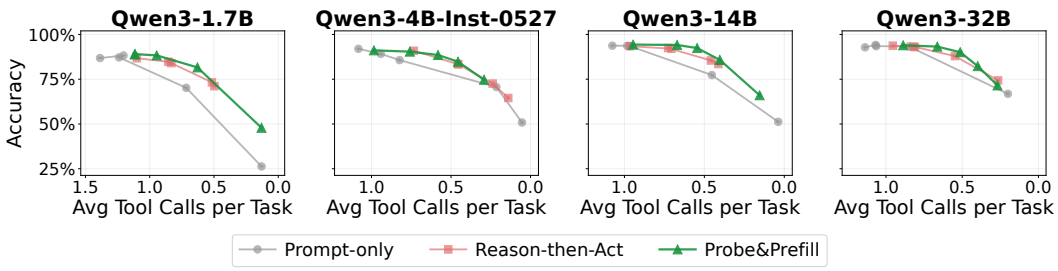

Figure 3: Accuracy vs. total tool calls for Qwen models. PROBE&PREFILL sweeps threshold τ from 0.1 to 0.9, achieving a strictly better tradeoff than both baselines.

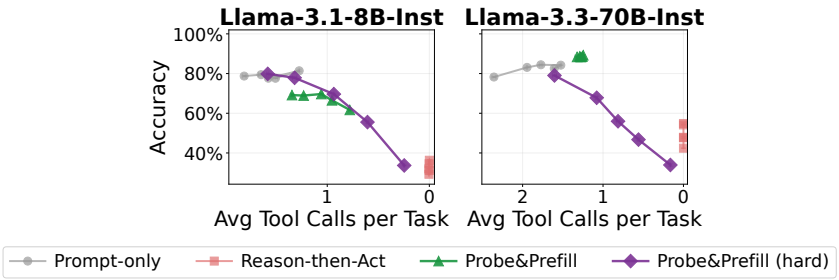

Figure 4: Accuracy vs. total tool calls for Llama models. Soft prefill (green) is partially ignored; hard prefill (purple) forces the output format and restores full tradeoff control.

PROBE&PREFILL outperforms Reason-then-Act. PROBE&PREFILL also outperforms the Reason-then-Act baseline (red) in most cases, despite requiring no additional reasoning tokens. Reasoning asks the model to verbally assess tool necessity before acting, yet the probe's prediction from the pre-generation hidden state is more accurate than the model's own verbalized assessment. This suggests that reasoning about tool necessity is largely superficial: it does not improve the model's underlying decision beyond what is already encoded in its hidden states before any reasoning takes place. A striking example comes from the Llama models, where reasoning completely collapses tool calling (the red line in Figure 4 drops to near-zero tool calls with large accuracy loss), yet PROBE&PREFILL still achieves strong performance by reading the hidden state directly.

Adaptive tool-call reduction. A key advantage of PROBE&PREFILL over the baselines is *adaptive* reduction: the probe selectively skips easy calls while preserving hard ones. Table 4 compares the accuracy cost per saved call across all prompt strategies and PROBE&PREFILL (τ=0.5), averaged over six models. Every baseline shows strongly negative costs, meaning each saved call comes at a significant accuracy penalty, especially on hard tasks. PROBE&PREFILL achieves the lowest cost across all difficulty levels (-1.6 easy, -3.4 hard), reducing tool calls with minimal accuracy loss.

Table 4: Accuracy cost per saved call ($\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$), averaged across six models (threshold τ is set to 0.5 for PROBE&PREFILL). All changes are relative to Default ($\star$, Prompt-only). More negative $\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$ means each saved call costs more accuracy. PROBE&PREFILL achieves lowest cost per saved call.

	Easy			Medium			Hard			Overall		
Method	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$
Necessary (N)	−0.3	−0.09	−3.5	−0.7	−0.04	−17.9	−1.9	−0.04	−44.4	−1.0	−0.06	−16.8
Necessary + Reason-then-Act	−8.1	−0.95	−8.5	−16.3	−0.82	−19.7	−23.2	−0.70	−32.9	−15.8	−0.82	−19.2
Sparse (S)	−6.3	−0.55	−11.3	−7.9	−0.47	−16.9	−11.1	−0.35	−31.5	−8.4	−0.46	−18.4
Sparse + Reason-then-Act	−9.9	−1.13	−8.8	−19.9	−1.04	−19.1	−29.3	−0.84	−34.7	−19.7	−1.00	−19.6
No Tool (X)	−18.1	−0.72	−25.2	−26.7	−0.72	−36.9	−41.4	−0.69	−60.4	−28.7	−0.71	−40.5
No Tool + Reason-then-Act	−12.4	−1.18	−10.5	−23.5	−1.12	−20.9	−36.6	−0.94	−39.1	−24.2	−1.08	−22.4
PROBE&PREFILL (Ours)	−1.1	−0.66	**−1.6**	−3.4	−0.54	**−6.2**	−0.8	−0.24	**−3.4**	−1.7	−0.48	**−3.6**

5.3 Additional experiments

Since PROBE&PREFILL introduces a learned probe, it is important to verify that results are not artifacts of overfitting to the training environments or sensitive to hyperparameter choices. We conduct the following additional experiments (full details in the Appendix):

- **Complete per-model results** (Appendix B): Full numerical tables for all models, prompt modes, reasoning settings, and probe thresholds on single-hop tasks.
- **Multi-hop tasks** (Appendix C): On the 3 multi-hop environments, PROBE&PREFILL reduces tool calls by up to 75% on Qwen models while maintaining or improving accuracy.
- **Out-of-distribution transfer** (Appendix D): The probe trained on a subset of environments generalizes to held-out environments within the same category.
- **Ablation studies** (Appendix E): Soft vs. hard prefill, temperature scaling, layer selection, data efficiency, and regularization strength.
- **Inference overhead** (Appendix F): The probe adds $<$1ms per task on top of the standard prefill forward pass, with no additional model calls.
- **Real-world agentic search** (Appendix G): On Search-o1 agentic benchmark, PROBE&PREFILL reduces search API calls by up to 56% while matching or exceeding baseline accuracy.
- **Supervised training baselines** (Appendix H): We compare with full SFT on 900 examples and with full SFT and rank-16 LoRA on 6,000 examples; these training recipes do not consistently reduce tool calls.
- **Confidence-based controllers** (Appendix I): Verbalized confidence, first-token likelihood, and self-ask drive the same soft-prefill mechanism.
- **Difficulty supervision control** (Appendix J): We compare necessity labels with an easy-versus-hard training signal.

6 Related Work

Agentic Tool-use benchmarks. Several benchmarks evaluate LLM tool use. ToolQA [Zhuang et al., 2023] tests question answering over external data sources; API-Bank [Li et al., 2023b] evaluates API selection across hundreds of real APIs; Toolformer [Schick et al., 2023] and ToolLLM [Qin et al., 2023] train models to invoke tools correctly; Gorilla [Patil et al., 2024] targets correct API call generation; and BFCL [Patil et al., 2025] provides a comprehensive function-calling leaderboard spanning single-turn, multi-turn, and agentic settings. All these benchmarks evaluate *whether* models can use tools correctly, assuming every task requires a tool. In contrast, WHEN2TOOL evaluates the tool-call *decision*: given the correct tool, the model must decide whether to use it or solve directly (Table 1).

Efficient tool calling. Recent work on reducing unnecessary tool calls takes several approaches. Xu et al. [2025] fine-tune the model to invoke tools only when its confidence is low, reducing calls by $\sim$50% on arithmetic and QA tasks. Wu et al. [2025] jointly optimize agent instructions and tool descriptions via verbalized feedback, reducing calls by up to 70%. Yang et al. [2026] survey efficiency across memory, tool learning, and planning in LLM agents. These works directly build costly interventions, such as SFT pipelines or iterative prompt optimization, without first investigating *why* models overcall. We instead take a mechanistic approach in a more realistic agentic setting, showing that the model's hidden state already encodes the tool-call decision and can be leveraged with only a lightweight linear probe that takes seconds to train and adds negligible inference cost.

Probing and controlling LLM behavior. Linear probing has revealed that LLM hidden states encode syntactic structure [Hewitt and Manning, 2019], factual knowledge [Burns et al., 2022], truthfulness [Li et al., 2023a], and self-knowledge [Kadavath et al., 2022]. Building on these findings, a growing body of work uses internal representations to steer model behavior: activation addition [Turner et al., 2024] and representation engineering [Zou et al., 2023] modify hidden activations during generation, concept bottleneck LLMs [Sun et al., 2024b,a] route predictions through interpretable concept layers, and recent work edits model weights guided by steering directions for reasoning control [Sun et al., 2025b, 2026, Yan et al., 2025] and skill unlearning [Li

et al., 2025b]. Our work extends probing to a new domain, agentic tool-use decisions, showing that models "mostly know" when they need tools but fail to act on it. Unlike prior steering methods that modify activations or weights, we steer via output prefilling based on a probe prediction, requiring no modification to the forward pass and remaining compatible with any serving infrastructure.

7 Conclusion

In this work, we showed that LLM agents already encode reliable tool-necessity signals in their hidden states, even when they fail to act on them during generation. A simple linear probe extracts this signal, and prefilling the model's response based on the probe prediction yields a strictly better accuracy–efficiency tradeoff than both Prompt-only and Reason-then-Act baselines. Our benchmark, probing analysis, and method together demonstrate that lightweight, training-free interventions can meaningfully improve agent tool-use efficiency.

Acknowledgement

C. Sun, G. Yan, and T.-W. Weng are partially supported by National Science Foundation under Grant No. 2313105, 2430539, and Intel Rising Star Faculty Award. The authors would like to thank the National Research Platform (NRP) at UCSD for computational support and anonymous reviewers for valuable feedback.

References

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Collin Burns, Haotian Ye, Dan Klein, and Jacob Steinhardt. Discovering latent knowledge in language models without supervision. *arXiv preprint arXiv:2212.03827*, 2022.

John Hewitt and Christopher D Manning. A structural probe for finding syntax in word representations. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4129–4138, 2019.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, et al. Language models (mostly) know what they know. *arXiv preprint arXiv:2207.05221*, 2022.

Kenneth Li, Oam Patel, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. Inference-time intervention: Eliciting truthful answers from a language model. *Advances in Neural Information Processing Systems*, 36:41451–41530, 2023a.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 3102–3116, 2023b.

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. Search-o1: Agentic search-enhanced large reasoning models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 5420–5438, 2025a.

Yongce Li, Chung-En Sun, and Tsui-Wei Weng. Effective skill unlearning through intervention and abstention. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6358–6371, 2025b.

Linbo Liu, Xinle Liu, Qiang Zhou, Lin Chen, Yihan Liu, Hoan Nguyen, Behrooz Omidvar-Tehrani, Xi Shen, Jun Huan, Omer Tripp, et al. Migrationbench: Repository-level code migration benchmark from java 8. *arXiv preprint arXiv:2505.09569*, 2025.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

OpenAi. Deep research system card, 2025. URL `https://openai.com/index/deep-research-system-card/`. Accessed: 2026-04-26.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37: 126544–126565, 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 *Forty-second International Conference on Machine Learning*, 2025.

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

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

Rulin Shao, Akari Asai, Shannon Zejiang Shen, Hamish Ivison, Varsha Kishore, Jingming Zhuo, Xinran Zhao, Molly Park, Samuel G Finlayson, David Sontag, et al. Dr tulu: Reinforcement learning with evolving rubrics for deep research. *arXiv preprint arXiv:2511.19399*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Chung-En Sun, Tuomas Oikarinen, Berk Ustun, and Tsui-Wei Weng. Concept bottleneck large language models. *arXiv preprint arXiv:2412.07992*, 2024a.

Chung-En Sun, Tuomas Oikarinen, and Tsui-Wei Weng. Crafting large language models for enhanced interpretability. *arXiv preprint arXiv:2407.04307*, 2024b.

Chung-En Sun, Ge Yan, Akshay Kulkarni, and Tsui-Wei Weng. Refine: A framework for trustworthy large reasoning models with reliability, faithfulness, and interpretability. *arXiv preprint arXiv:2510.09062*, 2025a.

Chung-En Sun, Ge Yan, and Tsui-Wei Weng. Thinkedit: Interpretable weight editing to mitigate overly short thinking in reasoning models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 17012–17036, 2025b.

Chung-En Sun, Ge Yan, Zimo Wang, and Tsui-Wei Weng. Steer2edit: From activation steering to component-level editing. *arXiv preprint arXiv:2602.09870*, 2026.

Alexander Matt Turner, Lisa Thiergart, Gavin Leech, David Udell, Ulisse Mini, and Monte MacDiarmid. Activation addition: Steering language models without optimization. 2024.

Bin Wu, Edgar Meij, and Emine Yilmaz. A joint optimization framework for enhancing efficiency of tool utilization in llm agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 22361–22373, 2025.

Hongshen Xu, Zihan Wang, Zichen Zhu, Lei Pan, Xingyu Chen, Shuai Fan, Lu Chen, and Kai Yu. Alignment for efficient tool calling of large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 17787–17803, 2025.

Ge Yan, Chung-En Sun, et al. Reflctrl: Controlling llm reflection via representation engineering. *arXiv preprint arXiv:2512.13979*, 2025.

Xiaofang Yang, Lijun Li, Heng Zhou, Tong Zhu, Xiaoye Qu, Yuchen Fan, Qianshan Wei, Rui Ye, Li Kang, Yiran Qin, et al. Toward efficient agents: Memory, tool learning, and planning. *arXiv preprint arXiv:2601.14192*, 2026.

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

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. Toolqa: A dataset for llm question answering with external tools. *Advances in Neural Information Processing Systems*, 36: 50117–50143, 2023.

Andy Zou, Long Phan, Sarah Chen, James Campbell, Phillip Guo, Richard Ren, Alexander Pan, Xuwang Yin, Mantas Mazeika, Ann-Kathrin Dombrowski, et al. Representation engineering: A top-down approach to ai transparency. *arXiv preprint arXiv:2310.01405*, 2023.

Table of Contents

A Benchmark environment details **14**
- A.1 Design principles . . . 14
- A.2 Evaluation framework . . . 14
- A.3 Category A: Computational scale . . . 15
- A.4 Category B: Knowledge boundary . . . 17
- A.5 Category C: Execution tracking . . . 19
- A.6 Multi-hop environments . . . 22
- A.7 Difficulty validation . . . 22

B Detailed single-hop results **24**

C Multi-hop evaluation **25**

D Out-of-distribution generalization **27**

E Ablation studies **28**
- E.1 Soft vs. hard prefill . . . 28
- E.2 Temperature scaling . . . 29
- E.3 Layer selection . . . 29
- E.4 Data efficiency . . . 30
- E.5 Regularization strength . . . 30

F Inference overhead **32**

G Generalization to agentic search (Search-o1 benchmarks) **33**

H Additional baseline: Compare PROBE&PREFILL with Supervised Fine-Tuning (SFT) **35**
- H.1 Original training setting: 900 examples . . . 35
- H.2 Larger training setting: 6,000 examples . . . 35

I Additional baselines: Compare PROBE&PREFILL with Confidence-Based Tool-Use Controllers **37**

J Necessity Supervision versus Difficulty Supervision **38**

A Benchmark environment details

Table 5 provides an overview of all 15 environments. Below we describe each environment in detail, including its real-world motivation, available tools, answer format, and how difficulty levels are constructed.

Table 5: Overview of WHEN2TOOL environments. Single-hop envs have 20 train / 50 test tasks per difficulty. Multi-hop envs follow the same split with 3-hop chained tasks ($x \to y \to z$).

Category	Type	Environment	Example (easy → hard)
A: Scale	1-hop	CalculatorEnv	"$20 + 20$" → "$(10^{12} \times 10^{11}) - 10^{8}$"
		StatisticsEnv	"Mean of [5,7,13]" → "Correlation of 25-element lists"
		CountingEnv	"$C(6, 4)$" → "$C(80, 40)$"
		MatrixEnv	"det of 2×2" → "det of 5×5"
		PrimeEnv	"Is 29 prime?" → "Is 104729 prime?"
	3-hop	ChainedCalculatorEnv	x=40−10, y=x+5, z=y−19 → trillion-scale chains
B: Knowledge	1-hop	RetrieverEnv	"Capital of France" → "Synthetic entity lookup"
		HistoricalYearEnv	"Moon landing year" → "Fictional event year"
		GameRuleEnv	"Chess pieces per player" → "Fictional game stats"
		HashEnv	"MD5 of 'hello' " → "Custom hash algorithm"
		DecodingEnv	"Morse: SOS" → "Custom cipher decode"
	3-hop	ChainedRetrieverEnv	Look up x, use x to find y, look up z about y
C: Execution	1-hop	ListManipulationEnv	"Remove from [1,2,3]" → "2D list operations"
		DateTimeEnv	"Days in April" → "Day-of-week for arbitrary date"
		CodeExecutorEnv	"print(2+3)" → "Trace 20-iteration DP"
		ScheduleEnv	"2 meetings overlap?" → "10+ meetings, find free slots"
		RegexMatchEnv	"\d+ on 'abc123' " → "Complex regex on 60-char string"
	3-hop	ChainedCodeExecutorEnv	Run code$_1$ → x, code$_2$(x) → y, code$_3$(y) → z

A.1 Design principles

WHEN2TOOL is designed to be lightweight, zero-cost, and easily extensible. All environments run locally with no API keys, external services, or network access required. New environments or difficulty levels can be added by writing a task generator with a fixed random seed.

1. **Zero-cost, fully offline**: All tool responses are simulated locally and deterministically. No paid APIs, no network calls, no rate limits. The entire benchmark can be run on a single machine.
2. **Exact-form answers**: Answers are numbers, strings, or lists that can be verified programmatically with no ambiguity or need for LLM-based judging.
3. **Short inputs**: Task descriptions are 1–15 lines, keeping prompt overhead low and ensuring the tool-call decision is the primary challenge.
4. **Unambiguous category membership**: Each environment belongs to exactly one category, enabling clean ablation across tool-necessity types.
5. **Deterministic and reproducible**: All tasks are generated with fixed random seeds. Re-running any generator produces byte-identical output.
6. **Easily extensible**: Adding a new environment requires only a self-contained task generator script. The benchmark grew from 15 single-hop to 18 environments (including 3 multi-hop) with no changes to the evaluation infrastructure.

A.2 Evaluation framework

WHEN2TOOL reports two quantities per model per setting:

Accuracy. Each task has an exact-form expected answer (number, string, or list). The model's final response is extracted from the `\boxed{}` output format and compared against the ground truth using a deterministic evaluator that handles numeric tolerance, case-insensitive string matching, and equivalent representations (e.g., different date formats). No LLM-based judging is used.

Total tool calls. We count the total number of tool calls made across the test set. Combined with the three difficulty levels (easy, medium, hard), this allows users to compare how different models or methods trade off accuracy against tool usage at each difficulty, revealing whether a method reduces calls indiscriminately or adaptively targets unnecessary ones.

Reproducibility. All tasks are generated with fixed random seeds. All experiments are run 3 times with different random seeds; we report mean and standard deviation. The benchmark, evaluation code, and all task generators will be released upon acceptance.

A.3 Category A: Computational scale

These environments test whether the model can assess the limits of its own mental arithmetic. The model understands the operation in every case; the question is whether the numbers or data involved exceed what it can compute reliably.

A.3.1 CalculatorEnv

Motivation. Arithmetic is the most basic tool-use scenario for any LLM agent. Agents that assist with financial calculations, scientific computations, or everyday math must decide whether an expression is simple enough to compute directly or requires calling a calculator. This environment isolates that decision by varying only the magnitude of the operands.

Tools.

- `evaluate_expression(expr)`: Evaluates a mathematical expression string and returns the exact numerical result.
- `get_last_result()`: Returns the result of the most recent evaluation, useful for multi-step calculations.
- `clear_last_result()`: Clears the stored result.

Answer format. Exact number.

Difficulty levels.

- **Easy**: Small numbers (2–40) with 5 expression templates: `a+b`, `a-b`, `a*b`, `(a+b)-c`, `(a*b)+c`.
  Example: "Compute exactly: $20 + 20$" $\rightarrow$ **40**
- **Medium**: 3-digit numbers (80–900) with multiplication, division, and modulo.
  Example: "Compute exactly: $(810 \times 87) - 85 + 178$" $\rightarrow$ **70563**
- **Hard**: Numbers in 10^9–10^{12}, far exceeding mental computation.
  Example: "Compute exactly: $(39006255142 \times 342002902703) - 702386298$" $\rightarrow$ **13340252482137117062528**

A.3.2 StatisticsEnv

Motivation. Data analysis agents frequently need to compute summary statistics, such as means, standard deviations, and correlations, to generate reports, evaluate A/B tests, or summarize datasets. While simple averages of a few numbers are easy to compute mentally, statistics involving larger datasets or more complex measures (e.g., Pearson correlation) quickly become infeasible without a tool.

Tools.

- `compute_stat(data, stat_type)`: Computes a specified statistic (`mean`, `median`, `std`, `percentile`, `correlation`, etc.) on the provided data and returns the exact result.

- `describe(data)`: Returns a full descriptive summary (count, mean, std, min, quartiles, max) for the dataset.

Answer format. Exact number to specified precision.

Difficulty levels.

- **Easy**: Mean or median of 3–5 small numbers.
  Example: "What is the median of [3, 7, 1, 9, 5]?" → **5**
- **Medium**: Standard deviation or percentiles of 8–15 numbers.
  Example: "What is the standard deviation of [12, 15, 18, 22, 25, 30, 14, 19, 27, 11]? Round to 2 decimal places." → **6.33**
- **Hard**: Pearson correlation on 20–30 numbers.
  Example: "What is the Pearson correlation between X=[...20 numbers...] and Y=[...20 numbers...]? Round to 4 decimal places." → **0.9994**

A.3.3 CountingEnv

Motivation. Combinatorial calculations arise in planning, scheduling, and resource allocation tasks. An agent planning team assignments or seating arrangements may need to compute combinations or permutations. Small values are manageable mentally, but combinatorial results grow extremely fast, making tools essential for larger inputs.

Tools.

- `combination(n, k)`: Computes $\binom{n}{k}$ and returns the exact integer.
- `permutation(n, k)`: Computes $P(n,k) = \frac{n!}{(n-k)!}$ and returns the exact integer.
- `factorial(n)`: Computes $n!$ and returns the exact integer.

Answer format. Exact integer.

Difficulty levels.

- **Easy**: Small factorials and combinations ($C(8,3)$, $6!$).
  Example: "How many ways can you choose 2 items from 5?" → **10**
- **Medium**: Larger combinations ($C(20,7)$, $14!$).
  Example: "Compute $P(15,4)$." → **32760**
- **Hard**: Very large values ($C(80,30)$, $25!$).
  Example: "What is $C(50,25)$?" → **126410606437752**

A.3.4 MatrixEnv

Motivation. Matrix operations are fundamental to ML engineering (weight matrices, attention computations), computer graphics (transformations), and scientific computing (solving linear systems). Computing a 2×2 determinant is straightforward, but determinants of larger matrices involve recursive expansion with many terms, making mental computation error-prone.

Tools.

- `matrix_determinant(matrix)`: Computes the determinant of a square matrix and returns the exact value.
- `matrix_multiply(A, B)`: Computes the product of two matrices and returns the result matrix.
- `matrix_trace(matrix)`: Computes the trace (sum of diagonal elements) and returns the exact value.

Answer format. Exact number or matrix.

Difficulty levels.

- **Easy**: 2×2 determinant or trace.
  Example: "What is the trace of $\begin{bmatrix} 3 & 1 \\ 7 & 4 \end{bmatrix}$?" $\rightarrow$ **7**
- **Medium**: 3×3 determinant.
  Example: "What is the determinant of $\begin{bmatrix} 2 & 3 & 1 \\ 4 & 1 & 3 \\ 1 & 2 & 4 \end{bmatrix}$?" $\rightarrow -$**17**
- **Hard**: 4×4 or 5×5 determinant, where cofactor expansion requires tracking dozens of terms.

A.3.5 PrimeEnv

Motivation. Primality testing and factorization arise in cryptography agents, math assistants, and puzzle solvers. Recognizing small primes is easy, but testing primality of large numbers or factoring them requires systematic trial division or more advanced algorithms that exceed mental capacity.

Tools.

- `is_prime(n)`: Tests whether n is prime and returns a boolean.
- `nth_prime(n)`: Returns the n-th prime number.
- `factorize(n)`: Returns the complete prime factorization of n as a string (e.g., "$2 \times 3 \times 5$").

Answer format. Boolean, factor string, or integer.

Difficulty levels.

- **Easy**: Small primes and factorizations.
  Example: "Is 17 a prime number?" $\rightarrow$ **True**
- **Medium**: 3-digit numbers.
  Example: "What is the 50th prime number?" $\rightarrow$ **229**
- **Hard**: 5–6 digit numbers.
  Example: "What is the prime factorization of 8191?" $\rightarrow$ **8191** (it is prime)

A.4 Category B: Knowledge boundary

These environments test whether the model can assess what information exists in its own parameters. The model must judge whether it possesses the factual knowledge needed to answer, a fundamentally different self-assessment from computational feasibility.

A.4.1 RetrieverEnv

Motivation. Research agents and question-answering systems routinely need to look up facts from external corpora. The key self-assessment is whether the model already knows the answer from pretraining or needs to search. This environment is unique in requiring a two-step retrieval process: first searching for relevant documents, then reading the full content.

Tools.

- `search_corpus(query, top_k)`: Searches a document corpus by keyword matching against document titles. Returns metadata and a short snippet (first 100 characters) for the top-k results, but *not* the full document text.
- `read_doc(doc_id)`: Retrieves the full text of a specific document by its ID, including title, content, and word count.

This is the only environment requiring two tool calls: the model must first call `search_corpus` to identify the relevant document, then call `read_doc` to obtain the full text containing the answer.

Answer format. Exact string (name, number, or phrase).

Difficulty levels.

- **Easy**: Well-known facts across 10 categories (capitals, currencies, elements, authors, etc.; 75 facts in pool).
  Example: "What is the capital of France?" → **Paris**
- **Medium**: Less common facts the model might partially know (obscure capitals, element symbols like Sn for Tin, Sb for Antimony; 72 facts).
  Example: "What is the chemical symbol for Tin?" → **Sn**
- **Hard**: *Synthetic* entities and relations generated with random names and values, embedded in a corpus with distractor documents. No model can have seen these during pretraining.
  Example: "What is the coolant class for Taskforce Nimbus-73?" → **Class-C8**

A.4.2 HistoricalYearEnv

Motivation. Education assistants, research agents, and trivia solvers frequently need to recall when historical events occurred. The model must assess whether it confidently knows the date or should look it up. We control difficulty by moving from universally known events to obscure ones to entirely fictional events.

Tools.

- `lookup_year(event)`: Takes an event description and returns the year it occurred, along with a brief contextual summary.

Answer format. Exact integer (year).

Difficulty levels.

- **Easy**: Famous events every model knows.
  Example: "What year did humans first land on the Moon?" → **1969**
- **Medium**: Less well-known events the model might get wrong.
  Example: "What year was the Treaty of Tordesillas signed?" → **1494**
- **Hard**: *Fictional* events that exist only in our database.
  Example: "What year was the Accord of Velmorath signed?" → **1723**

A.4.3 GameRuleEnv

Motivation. Game assistants and trivia agents need to recall numeric facts about games, such as board sizes, piece counts, and deck sizes. These facts range from universally known (chess has 64 squares) to obscure (number of tiles in a Mahjong set) to completely fictional, testing the model's ability to recognize the limits of its own knowledge.

Tools.

- `lookup_rule(game, attribute)`: Takes a game name and an attribute query, and returns the numeric answer along with a rule description.

Answer format. Exact integer.

Difficulty levels.

- **Easy**: Well-known game facts.
  Example: "How many squares are on a standard chessboard?" → **64**
- **Medium**: Less common game facts.
  Example: "How many tiles are in a standard Mahjong set?" → **144**
- **Hard**: *Fictional* games with invented rules.
  Example: "How many cards are in a Zephyr deck?" → **72**

A.4.4 HashEnv

Motivation. Security agents, file integrity checkers, and API authentication systems routinely compute cryptographic hashes. While a model might have memorized the MD5 hash of common strings like "hello" from its training data, it cannot compute hashes for novel inputs or custom

algorithms. This environment tests whether the model can distinguish memorized outputs from those requiring computation.

Tools.

- `compute_hash(algorithm, input_string)`: Computes the hash of the input string using the specified algorithm and returns the hex-encoded digest.

Answer format. Exact hex string.

Difficulty levels.

- **Easy**: MD5 or SHA1 of well-known short strings (the model may have memorized these). *Example*: "What is the MD5 hash of 'hello'?" → **5d41402abc4b2a76b9719d911017c592**
- **Medium**: SHA256/SHA1/MD5 of short phrases less likely memorized. *Example*: "What is the SHA1 hash of 'machine learning'?"
- **Hard**: 5 *custom* hash algorithms (`fnv1a_custom`, `djb2_custom`, `sdbm_custom`, `murmur_custom`, `jenkins_custom`) with non-standard primes and offsets. No model can know the outputs of these algorithms. *Example*: "What is the MURMUR_CUSTOM hash of 'xK9mQ2'?"

A.4.5 DecodingEnv

Motivation. Communication agents and puzzle solvers encounter various encoding and cipher schemes. Morse code for "SOS" is universally known, and ROT13 is a simple well-known transformation, but custom substitution ciphers with arbitrary mappings cannot be decoded without access to the cipher definition. This environment tests the boundary between known and unknown encoding schemes.

Tools.

- `decode(scheme, ciphertext)`: Decodes the ciphertext using the specified scheme and returns the plaintext.
- `encode(scheme, plaintext)`: Encodes the plaintext using the specified scheme and returns the ciphertext.

Answer format. Exact string.

Difficulty levels.

- **Easy**: Morse code for short well-known words and ROT13. *Example*: "Encode 'SOS' in Morse code." → **... --- ...**
- **Medium**: Caesar cipher with various shifts and Morse for longer words. *Example*: "Decode 'NWTPYE' using Caesar cipher with shift 11." → **CIPHER**
- **Hard**: 4 *custom* substitution ciphers (`scramble1`, `scramble2`, `alpha7`, `reverse`) with arbitrary letter mappings that the model has never seen. *Example*: "Decode 'KFPQA' using the scramble1 cipher." → **HELLO**

A.5 Category C: Execution tracking

These environments test whether the model can assess its own reliability when tracing sequential procedures. The model knows the algorithm and has all the information; the question is whether it can execute the steps without error.

A.5.1 ListManipulationEnv

Motivation. Data processing agents, database operations, and ETL pipelines frequently perform list transformations, including inserting, removing, sorting, and reversing elements. Tracking a single operation on a short list is trivial, but applying operations to larger lists or 2D arrays (where row and column axes must be tracked simultaneously) quickly exceeds reliable mental execution.

Tools.

- `append(list, value)`: Appends a value to the end of the list and returns the updated list.
- `remove(list, index)`: Removes the element at the given index and returns the updated list.
- `insert(list, index, value)`: Inserts a value at the given index and returns the updated list.
- `sort(list, axis)`: Sorts the list (or sorts a 2D list along the specified axis) and returns the result.
- `reverse(list)`: Reverses the list and returns the result.

Answer format. Exact list in Python format.

Difficulty levels.

- **Easy**: 1D list of 3–5 small integers (1–40), single operation.
  Example: "Initial [7, 19, 29]. Apply insert(index=2, value=36). Return final list." → **[7, 19, 36, 29]**
- **Medium**: 1D list of 6–10 medium integers (40–260), single operation.
  Example: "Initial [86, 197, 199, 232, 66, 53, 234]. Apply sort(). Return final list." → **[53, 66, 86, 197, 199, 232, 234]**
- **Hard**: 2D list (matrix) of large integers (300–5000), operations along row or column axis.
  Example: "Initial [[2063, 3740, ...], [4252, 3661, ...]]. Apply sort(axis=0). Return final 2D list."

A.5.2 DateTimeEnv

Motivation. Scheduling agents, calendar assistants, and deadline trackers must perform date arithmetic, such as counting days between dates, adding durations, and determining days of the week. Simple within-month counting is easy, but calculations that cross month boundaries, handle leap years, or require day-of-week computation for arbitrary dates involve enough edge cases that mental execution becomes unreliable.

Tools.

- `date_add(date, days)`: Adds a number of days to a date and returns the resulting date in YYYY-MM-DD format.
- `date_diff(date1, date2)`: Computes the number of days between two dates.
- `day_of_week(date)`: Returns the day of the week for a given date.

Answer format. Exact number, date string (YYYY-MM-DD), or day name.

Difficulty levels.

- **Easy**: Simple counting within one month.
  Example: "How many days between January 3 and January 18?" → **15**
- **Medium**: Crossing month/year boundaries, leap years.
  Example: "How many days between February 25 and March 10, 2024?" → **14** (2024 is a leap year)
- **Hard**: Day-of-week for arbitrary dates, multi-year calculations.
  Example: "What day of the week is August 15, 2027?" → **Sunday**

A.5.3 CodeExecutorEnv

Motivation. Coding assistants and code review agents are frequently asked to predict the output of code snippets. Simple expressions are trivial, but code involving loops, recursion, or dynamic programming requires tracing many iterations where errors compound. This environment tests whether the model can recognize when code is too complex to trace mentally.

Tools.

- `run_python(code)`: Executes a Python code snippet in a sandboxed environment and returns the captured stdout output.

Answer format. Exact stdout string.

Difficulty levels.

- **Easy**: Simple one-line expressions.
  Example: "What is the output of: `print(len('hello'))`" → **5**
- **Medium**: Short loops, list comprehensions, string operations.
  Example: "What is the output of: `print(sum(x**2 for x in range(1,6)))`" → **55**
- **Hard**: Recursion, dynamic programming, Collatz sequences: code with 10–30+ iterations where mental tracing reliably fails.
  Example: "What is the output of: `n=27; steps=0; while n!=1: n=n//2 if n%2==0 else 3*n+1; steps+=1; print(steps)`" → **111**

A.5.4 ScheduleEnv

Motivation. Meeting schedulers and resource booking agents must find free time slots among existing appointments. With 2–3 meetings, checking for conflicts is easy. With 10+ overlapping meetings and specific duration constraints, mentally tracking all intervals to find available slots becomes error-prone.

Tools.

- `find_free_slot(meetings, duration, start, end)`: Finds available time slots of the specified duration within the given time range, considering all existing meetings.
- `check_conflict(meetings, new_meeting)`: Checks whether a proposed meeting conflicts with any existing meetings and returns a boolean.
- `list_meetings(meetings)`: Returns a formatted summary of all meetings sorted by start time.

Answer format. Exact time slot or boolean.

Difficulty levels.

- **Easy**: 2–3 meetings, find a free slot.
  Example: "Meetings: 9:00–10:00, 14:00–15:00. Is there a free 1-hour slot between 10:00 and 14:00?" → **Yes**
- **Medium**: 6–10 meetings, find all free slots.
  Example: "Given 8 meetings, list all free 30-min slots between 9:00 and 17:00."
- **Hard**: 15+ meetings with constraints, requiring the model to trace all intervals.
  Example: "Given 15 meetings, find the first available 1-hour slot."

A.5.5 RegexMatchEnv

Motivation. Log parsing agents, data extraction pipelines, and text processing tools rely on regular expressions. While simple patterns like `\d+` on short strings are easy to mentally evaluate, complex patterns with overlapping matches, lookahead assertions, or backreferences on long strings require the model to simulate a regex engine, a sequential process that quickly exceeds reliable mental execution.

Tools.

- `regex_match(pattern, text, operation)`: Applies the specified regex operation (`findall`, `match`, `search`, `sub`) to the text and returns the result.

Answer format. Exact Python list or matched string.

Difficulty levels.

- **Easy**: Simple character classes and quantifiers on short text.
  Example: "What does `re.findall(r'\d+', 'abc123def456')` return?" → **['123', '456']**

- **Medium**: Groups, lookahead/lookbehind, alternation on moderate text.
  Example: "What does `re.findall(r'(\w+)@(\w+)\.(\w+)', 'user@example.com admin@test.org')` return?" → **[('user', 'example', 'com'), ('admin', 'test', 'org')]**
- **Hard**: Complex patterns with overlapping matches on 60+ character strings, requiring the model to simulate regex engine backtracking.
  Example: "What does `re.findall(r'(?=([a-z]{3}))', 'abcdbcdabcdebcde...')` return?"

A.6 Multi-hop environments

In addition to the 15 single-hop environments, WHEN2TOOL includes 3 multi-hop environments that require chains of 3 dependent tool calls following the pattern $x \to y \to z$: each hop's output is needed as input for the next. These test whether models can assess tool necessity across a sequence of dependent operations.

All three multi-hop environments follow the same structure: the model executes hop 1 to obtain x, uses x as input to hop 2 to obtain y, and uses y as input to hop 3 to obtain the final answer z. Each hop reuses the same tools as its single-hop counterpart. Difficulty scales identically (easy = solvable mentally, hard = requires tools), but now the model must make a tool-call decision at each hop.

A.6.1 ChainedCalculatorEnv (Category A)

Three chained arithmetic computations where each expression depends on the previous result. Uses `evaluate_expression`.

Easy: "First compute $x = 40 - 10$. Then compute $y = x + 5$. Finally compute $z = y - 19$. Return z." → x=30, y=35, z=16.
Hard: "First compute $x = 808522010435 - 8197325888$. Then compute $y = x + 17046220916$. Finally compute $z = y \bmod 2343374$. Return z." → x=800324684547, y=817370905463, z=2054263.

A.6.2 ChainedRetrieverEnv (Category B)

Three chained knowledge lookups where the answer to each question determines what to ask next. The corpus contains target documents for all three hops plus distractors (10 docs). Uses `search_corpus` and `read_doc`.

Easy: "What is the longest river in the world? (x=Nile). Through how many countries does x flow? (y=11). Into which sea does the river that passes through y countries empty? (z=Mediterranean Sea)."
Hard: "What river formed the Central Lowlands? (x=the Ironflow River). What gives x its name? (y=iron-rich sediments). What color is the water of the river that carries y? (z=reddish-brown)." All entities are fictional and exist only in the provided corpus.

A.6.3 ChainedCodeExecutorEnv (Category C)

Three chained code executions where each script's output is the input to the next. Uses `run_code`.

Easy: "Run `print(17+6)` (x=23). Run `print(x*3)` (y=69). Run `print(y-7)` (z=62)."
Hard: "Run coin-change DP for amount=27 with coins [1,5,10] (x=5). Sum even Fibonacci numbers up to $2x$ (y=44). Compute LIS of an array with first element replaced by $y \bmod 50$ (z=7)." Each hop involves loops, recursion, or DP.

A.7 Difficulty validation

To validate that our difficulty levels create a meaningful decision boundary, we evaluate all six models in a no-tool setting where tools are unavailable and the model must answer directly. Table 6 reports the results across all 18 environments (single-hop and multi-hop combined). Averaged across models, easy tasks are solvable 69.4% of the time, medium tasks 54.4%, and hard tasks only 15.5%, confirming that the difficulty levels behave as intended.

Table 6: No-tool accuracy (%) by difficulty level across all 18 environments (1,080 train / 2,700 test). Each model is evaluated without tool access to validate that easy tasks are largely solvable directly, medium tasks are partially solvable, and hard tasks require tools.

	Train (1,080 tasks)				Test (2,700 tasks)			
Model	Easy	Med	Hard	Avg	Easy	Med	Hard	Avg
Qwen3-1.7B	34.7	26.1	4.7	21.9	37.0	25.2	6.3	22.9
Qwen3-4B-Inst.	73.6	64.4	21.9	53.3	74.4	61.2	20.4	52.0
Qwen3-14B	81.1	61.1	17.5	53.2	77.9	58.1	15.2	50.4
Qwen3-32B	83.3	69.4	20.3	57.7	82.8	69.1	21.3	57.7
Llama-3.1-8B	69.7	48.6	9.2	42.5	69.8	48.9	9.4	42.7
Llama-3.3-70B	75.8	65.0	22.2	54.4	74.7	63.7	20.6	53.0
Average	69.7	55.8	16.0	47.2	69.4	54.4	15.5	46.4

B Detailed single-hop results

This section provides the complete per-model results for the 15 single-hop environments that are summarized in the main text figures and tables.

Accuracy cost per saved call. Table 7 breaks down the accuracy cost per saved call ($\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$) from Table 4 by model. For each model, we compare Sparse (S), Sparse + Reason-then-Act, and PROBE&PREFILL (τ=0.5), all relative to Default ($\star$, Prompt-only).

Table 7: Per-model accuracy cost per saved call (τ=0.5). All deltas relative to Default ($\star$, Prompt-only). More negative = each saved call is more costly. **Bold** = best per model in each column.

		Easy			Medium			Hard			Avg		
Model	Method	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$	ΔAcc	ΔTC	$\frac{\Delta\text{Acc}}{-\Delta\text{TC}}$
Qwen3-1.7B	Sparse (S)	−17.7	−0.59	−29.9	−18.4	−0.39	−46.7	−18.0	−0.48	−37.7	−18.0	−0.49	−37.1
	Sparse + Reason-then-Act	−8.7	−0.83	−10.5	−17.9	−0.62	−28.7	−18.4	−0.61	−30.0	−15.0	−0.69	−21.8
	PROBE&PREFILL	**−0.9**	−0.37	**−2.4**	**+0.2**	−0.23	**+1.0**	**+0.9**	−0.17	**+5.3**	**+0.1**	−0.26	**+0.3**
Qwen3-4B-Inst.	Sparse (S)	−14.5	−0.84	−17.3	−20.7	−0.86	−24.1	−20.3	−0.48	−42.4	−18.5	−0.73	−25.5
	Sparse + Reason-then-Act	−14.5	−0.86	−16.9	−22.4	−0.90	−24.8	−13.0	−0.35	−36.6	−16.6	−0.70	−23.6
	PROBE&PREFILL	**−2.5**	−0.49	**−5.1**	**−5.5**	−0.51	**−10.7**	**+6.0**	−0.08	**+73.9**	**−0.7**	−0.36	**−1.9**
Qwen3-14B	Sparse (S)	−8.8	−0.59	−14.9	−12.9	−0.53	−24.3	−27.3	−0.47	−58.4	−16.3	−0.53	−30.8
	Sparse + Reason-then-Act	−4.4	−0.67	−6.6	−10.4	−0.62	−16.8	−9.7	−0.28	−34.7	−8.2	−0.52	−15.6
	PROBE&PREFILL	**−0.4**	−0.63	**−0.6**	**−3.6**	−0.55	**−6.5**	**+0.2**	−0.13	**+1.4**	**−1.3**	−0.44	**−2.9**
Qwen3-32B	Sparse (S)	**−0.8**	−0.38	**−2.2**	**−0.3**	−0.20	**−1.6**	**−1.3**	−0.08	**−16.9**	**−0.8**	−0.22	**−3.6**
	Sparse + Reason-then-Act	−4.6	−0.80	−5.8	−8.8	−0.56	−15.8	−5.3	−0.21	−25.1	−6.2	−0.52	−11.9
	PROBE&PREFILL	−2.1	−0.83	−2.5	−6.4	−0.66	−9.7	−3.6	−0.18	−20.1	−4.0	−0.56	−7.2
Llama-3.1-8B	Sparse (S)	**+2.7**	−0.42	**+6.4**	**+2.8**	−0.43	**+6.4**	**+0.4**	−0.27	**+1.6**	**+2.0**	−0.37	**+5.3**
	Sparse + Reason-then-Act	−22.6	−1.66	−13.6	−41.0	−1.69	−24.3	−65.8	−1.60	−41.2	−43.1	−1.65	−26.1
	PROBE&PREFILL	−5.3	−0.84	−6.4	−10.9	−0.68	−16.1	−13.1	−0.25	−51.3	−9.8	−0.59	−16.6
Llama-3.3-70B	Sparse (S)	+1.6	−0.51	+3.2	+2.0	−0.41	+4.8	−0.2	−0.34	−0.5	+1.1	−0.42	+2.7
	Sparse + Reason-then-Act	−4.8	−1.98	−2.4	−18.9	−1.87	−10.1	−63.3	−1.99	−31.7	−29.0	−1.95	−14.9
	PROBE&PREFILL	**+4.8**	−0.78	**+6.2**	**+6.0**	−0.61	**+9.8**	**+4.8**	−0.62	**+7.6**	**+5.2**	−0.67	**+7.8**

Full single-hop results. Table 8 reports accuracy (%) and total tool calls (TC) for all 6 models across all prompt modes, reasoning settings, and probe thresholds on the 2,250-task single-hop test set.

Table 8: Full single-hop results (2,250 test tasks, mean±std over 3 runs). F/D/N/S/X = Force/Default/Necessary/Sparse/No-tool. τ = probe threshold.

		Qwen3-1.7B		Qwen3-4B		Qwen3-14B		Qwen3-32B		Llama-8B		Llama-70B	
		Acc	TC	Acc	TC	Acc	TC	Acc	TC	Acc	TC	Acc	TC
Prompt-only	F	86.8±.2	3120±32	92.0±.0	2435±5	93.7±.1	2421±7	92.8±.1	2559±9	78.7±.4	4081±40	78.3±.3	5302±55
	D	88.2±.1	2709±22	89.2±.2	2118±9	93.7±.0	2211±2	94.1±.3	2404±20	79.5±.6	3708±55	83.1±.2	4377±7
	N	87.2±.3	2787±13	85.7±.1	1852±3	93.5±.2	2175±5	93.5±.2	2394±23	77.7±.6	3548±51	84.4±.1	3988±32
	S	70.2±.2	1611±20	70.7±.3	484±10	77.3±.2	1020±3	93.3±.3	1912±10	81.4±.4	2868±3	84.2±.1	3431±15
	X	26.3±.1	293±13	50.8±.5	121±14	51.2±.2	85±5	66.8±.5	451±15	77.6±.2	3391±53	82.5±.3	3615±39
Reason-then-Act	F	86.7±.5	2475±19	90.9±.1	1651±6	93.4±.1	2172±9	93.7±.1	2150±12	29.3±.2	9±8	42.4±.2	0±0
	D	84.7±.4	1923±27	83.4±.2	1024±0	92.0±.2	1589±3	92.9±.4	1823±6	31.2±.9	2±0	47.9±.5	0±0
	N	84.0±.7	1871±30	83.4±.4	1027±7	92.4±.2	1634±13	93.2±.1	1845±7	32.0±.1	11±3	47.6±.5	0±0
	S	73.2±.6	1161±22	72.5±.7	535±15	85.5±.3	1034±12	87.9±.2	1231±20	36.3±.4	0±0	54.1±.2	0±0
	X	71.1±.3	1117±19	64.5±.4	315±12	83.5±.4	928±9	74.3±.3	596±19	34.5±.3	6±2	54.8±.4	0±0
PROBE&PREFILL	τ=.1	88.8±.2	2512±16	91.1±.3	2216±10	94.3±.1	2128±9	94.0±.1	1996±9	69.2±.6	3027±20	88.4±.4	2976±58
	τ=.3	89.0±.3	2507±36	90.4±.4	1707±5	94.1±.2	1509±4	93.2±.1	1493±10	68.9±.2	2770±33	88.6±.2	2902±31
	τ=.5	88.3±.2	2128±18	88.5±.2	1309±1	92.4±.2	1227±5	90.1±.5	1155±22	69.7±.4	2381±28	88.3±.1	2871±49
	τ=.7	81.6±.3	1415±19	84.8±.2	1026±9	85.8±.1	907±6	82.3±.3	896±16	66.5±.2	2146±30	88.6±.5	2828±9
	τ=.9	47.9±.5	293±15	74.7±.3	657±4	66.0±.6	347±8	71.5±.4	604±14	61.7±.4	1753±23	89.2±.2	2804±10

C Multi-hop evaluation

We evaluate PROBE&PREFILL on the 3 multi-hop environments (ChainedCalculatorEnv, ChainedRetrieverEnv, ChainedCodeExecutorEnv), each requiring a chain of 3 dependent tool calls. The probe is trained on the 180-task multi-hop training set and evaluated on the 450-task test set.

Summary. Table 9 compares the best baseline (highest accuracy among all Prompt-only and Reason-then-Act settings that achieve at least 20% tool-call reduction) against the best probe threshold. On Qwen3-4B, the probe achieves higher accuracy (85.3% vs. 83.9%) with 75% fewer tool calls compared to the best baseline's 63%. On Qwen3-32B, the probe reduces calls by 55% while the best baseline only achieves 20%. On Llama models, the probe increases tool calls but also substantially increases accuracy (Llama-3.1-8B: 40.2→60.2%, Llama-3.3-70B: 62.4→80.3%), indicating that the probe correctly identifies these multi-hop tasks as genuinely requiring tools and steers the model toward necessary calls that the Default setting was missing.

Table 9: Multi-hop summary (450 test tasks). ΔTC relative to Default. **Bold** = larger reduction.

		Default		Best Baseline			PROBE&PREFILL		
Model	N	Acc	TC	Acc	TC	ΔTC	Acc	TC	ΔTC
Qwen3-1.7B	450	21.2	1180	**60.6**	175	−85%	59.2	85	**−93%**
Qwen3-4B	450	82.1	1719	83.9	636	−63%	**85.3**	437	**−75%**
Qwen3-14B	450	87.5	1503	85.7	789	**−47%**	**86.2**	996	−34%
Qwen3-32B	450	88.9	1634	88.6	1306	−20%	**89.0**	727	**−55%**
Llama-3.1-8B	450	40.2	1005	41.3	595	**−41%**	**60.2**	1361	+35%
Llama-3.3-70B	450	62.4	985	67.6	347	**−65%**	**80.3**	1789	+82%

Probe quality. Table 10 reports probe AUROC on the multi-hop test set. The probe achieves AUROC 0.84–0.97 across models, with Qwen3-4B reaching 0.966, confirming that tool necessity remains linearly decodable even for chained tasks. Llama-3.3-70B has the lowest AUROC (0.804), likely because this model's Default setting already under-calls tools on multi-hop tasks (TC=985 for 450 three-hop tasks), making the binary label noisier. Despite the lower AUROC, the probe still substantially improves accuracy on this model (62.4% → 80.3%).

Table 10: Multi-hop probe AUROC and classification accuracy.

Model	AUROC	Accuracy
Qwen3-1.7B	0.839	0.796
Qwen3-4B	0.966	0.947
Qwen3-14B	0.906	0.822
Qwen3-32B	0.944	0.873
Llama-3.1-8B	0.895	0.829
Llama-3.3-70B	0.804	0.729

Full results. Table 11 reports all settings. Several patterns emerge beyond the summary table. First, Reason-then-Act is particularly effective on Qwen3-1.7B multi-hop tasks (Sparse+R-t-A reaches 60.6% vs. Sparse Prompt-only's 41.3%), suggesting that explicit reasoning helps small models plan multi-step tool chains. Second, the Llama models exhibit the same reasoning collapse as on single-hop tasks: Reason-then-Act reduces tool calls to near zero on both Llama-3.1-8B (TC≤5) and Llama-3.3-70B (TC=0), with accuracy dropping substantially. Third, PROBE&PREFILL shows a clear threshold–accuracy tradeoff: on Qwen3-4B, sweeping τ from 0.1 to 0.9 reduces TC from 1287 to 175 while accuracy decreases gradually from 82.1% to 79.6%, providing smooth control that baselines cannot achieve.

Table 11: Full multi-hop results (450 test tasks, mean±std over 3 runs). F/D/N/S/X = Force/Default/Necessary/Sparse/No-tool. τ = probe threshold.

		Qwen3-1.7B		Qwen3-4B		Qwen3-14B		Qwen3-32B		Llama-8B		Llama-70B	
		Acc	TC	Acc	TC	Acc	TC	Acc	TC	Acc	TC	Acc	TC
Prompt-only	F	28.8±.4	1438±23	76.1±.7	2002±10	88.0±.4	1787±2	89.4±.5	1797±20	42.4±1.1	1288±20	77.8±1.6	1243±12
	D	21.2±.6	1180±37	82.1±.2	1719±14	87.5±.8	1503±4	88.9±.8	1634±22	40.2±1.7	1005±23	62.4±.5	985±32
	N	23.3±1.0	1230±21	84.3±.5	1390±17	87.0±.5	1495±10	85.4±3.1	1586±14	40.1±.5	1017±20	60.9±1.3	793±8
	S	41.3±1.2	397±16	83.6±.6	83±4	74.4±1.2	496±9	87.6±.8	832±28	41.3±1.0	595±42	67.6±.6	347±40
	X	26.7±.9	108±4	75.9±.5	13±2	67.4±.9	384±8	70.1±.5	58±10	34.3±1.2	639±36	36.7±2.8	385±22
Reason-then-Act	F	51.9±.5	841±24	83.5±.7	940±4	87.3±1.1	1326±9	88.6±.7	1306±21	23.2±.9	1±0	56.0±1.2	0±0
	D	55.7±2.4	345±14	83.6±.6	652±7	85.7±.4	789±24	86.7±1.1	1010±21	29.3±.9	2±1	57.0±1.7	0±0
	N	56.4±2.0	348±23	83.9±.5	636±6	84.4±.8	863±26	87.1±.5	1036±21	29.6±.8	5±2	58.3±.6	0±0
	S	60.6±.6	175±13	81.9±.9	253±6	76.1±.2	404±16	82.0±.8	565±9	37.0±2.7	4±4	60.6±1.2	0±0
	X	59.9±.9	206±14	78.5±.4	106±3	73.5±.6	301±19	74.0±1.1	294±14	36.4±.5	4±4	60.3±.7	0±0
PROBE&PREFILL	τ=.1	22.9±.9	1197±11	82.1±.4	1287±11	87.3±.5	1483±18	88.7±.5	1481±8	60.1±1.5	1361±9	80.1±1.0	1856±29
	τ=.3	24.4±.5	1190±31	85.3±.6	437±3	86.1±.5	996±13	89.0±.6	727±6	59.0±.9	1291±24	80.3±1.2	1789±4
	τ=.5	32.5±1.2	1121±10	83.2±1.3	366±9	82.9±1.0	771±21	86.1±.3	553±5	57.2±1.5	1186±35	79.7±1.1	1494±24
	τ=.7	39.5±.5	627±16	83.4±.1	354±8	79.9±.4	701±31	83.3±.7	493±13	55.6±.9	1156±16	78.7±1.3	1228±21
	τ=.9	59.2±.3	85±2	79.6±.7	175±10	79.3±.5	686±12	74.6±.3	353±14	54.3±.5	1047±11	69.9±1.3	1074±58

D Out-of-distribution generalization

To test whether the probe generalizes beyond its training environments, we conduct within-category leave-two-out experiments: for each category, we train the probe on 3 of the 5 environments and evaluate on all 5. Figure 5 compares the OOD probe (blue) against the in-distribution probe (green). The OOD probe achieves comparable accuracy–efficiency tradeoffs across all models, confirming that the probe learns general signals rather than environment-specific shortcuts.

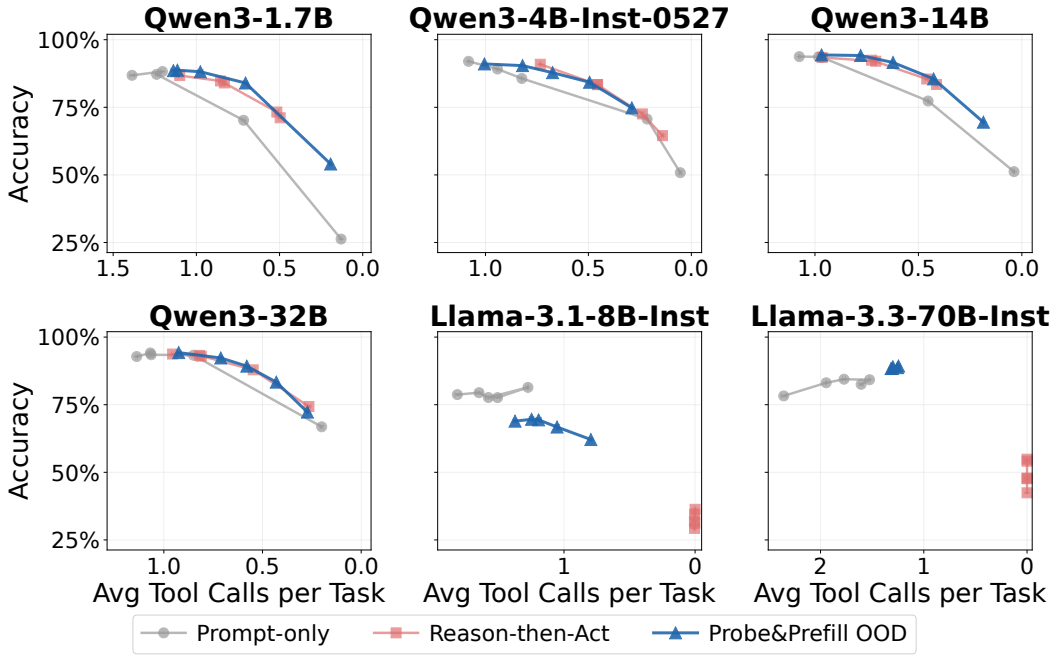

Figure 5: OOD generalization: in-distribution probe (green) vs. OOD probe trained on held-out environments (blue). Gray and red lines show Prompt-only and Reason-then-Act baselines. The OOD probe closely tracks the in-distribution probe across all models.

E Ablation studies

E.1 Soft vs. hard prefill

Soft prefill injects a natural language steering sentence that the model may override. Hard prefill forces the output format (`\boxed{` for direct answers, `{"name":` for tool calls), leaving no room for deviation. Figure 6 visualizes the tradeoff curves. On Qwen models, soft prefill generally achieves higher accuracy than hard at matched tool-call levels, because the model can override incorrect predictions. On Llama-3.1-8B, hard prefill achieves better accuracy at low thresholds (79.9% vs. 69.2% at τ=0.1) because the soft steering sentence is frequently ignored. On Llama-3.3-70B, soft prefill is remarkably stable (88.4–89.2% across all thresholds) because the model largely ignores the steering, while hard prefill provides a wide range (79.0% at τ=0.1 to 33.9% at τ=0.9), confirming that forcing the output format is the only way to control this model's tool-call behavior. Table 12 compares both modes across thresholds.

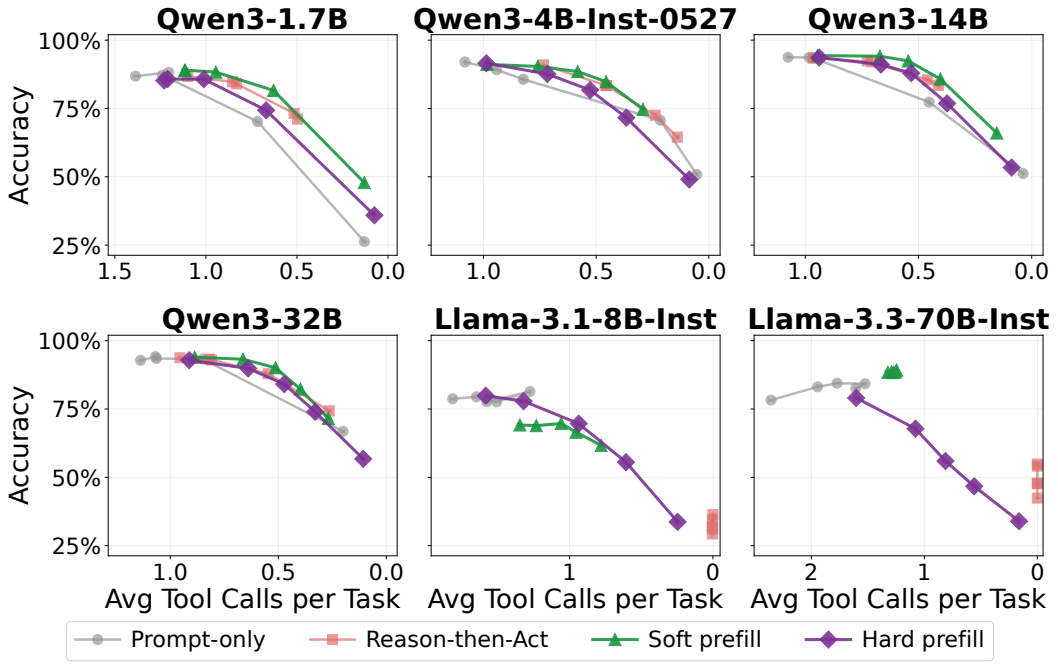

Figure 6: Soft prefill (green) vs. hard prefill (purple). Hard prefill forces the output format, while soft prefill allows the model to override.

Table 12: Soft vs. hard prefill (T=2.0). Each cell: Acc (%) / total tool calls.

Model	Mode	τ=0.1	τ=0.3	τ=0.5	τ=0.7	τ=0.9
Qwen3-1.7B	Soft	88.8/2512	89.0/2507	88.3/2128	81.6/1415	47.9/293
	Hard	85.3/2765	85.7/2723	85.7/2275	74.3/1506	35.9/169
Qwen3-4B	Soft	91.1/2216	90.4/1707	88.5/1309	84.8/1026	74.7/657
	Hard	91.6/2222	87.6/1612	81.7/1185	71.6/824	49.0/195
Qwen3-14B	Soft	94.3/2128	94.1/1509	92.4/1227	85.8/907	66.0/347
	Hard	93.6/2111	91.1/1498	87.9/1198	76.9/841	53.4/200
Qwen3-32B	Soft	94.0/1996	93.2/1493	90.1/1155	82.3/896	71.5/604
	Hard	92.9/2053	89.9/1439	84.1/1064	74.0/741	56.8/241
Llama-3.1-8B	Soft	69.2/3027	68.9/2770	69.7/2381	66.5/2146	61.7/1753
	Hard	79.9/3561	77.9/2969	69.6/2105	55.5/1363	33.7/554
Llama-3.3-70B	Soft	88.4/2976	88.6/2902	88.3/2871	88.6/2828	89.2/2804
	Hard	79.0/3609	67.8/2427	56.0/1830	46.7/1258	33.9/366

E.2 Temperature scaling

The probe outputs a logit $z = \mathbf{w}^\top \mathbf{x} + b$, which is converted to a probability via $p = \sigma(z/T)$ before thresholding. Higher T flattens the distribution, making the probe more conservative about predicting tool necessity. Table 13 compares $T \in \{1.0, 2.0, 3.0\}$.

Figure 7 visualizes the tradeoff curves. At T=1.0, the probe is sharp: low thresholds already reduce many tool calls, providing a wider operating range. At T=3.0, the probe is diffuse, offering finer control in the middle range. T=2.0 provides a good balance across models. The choice of temperature does not qualitatively change the finding that PROBE&PREFILL outperforms prompt baselines.

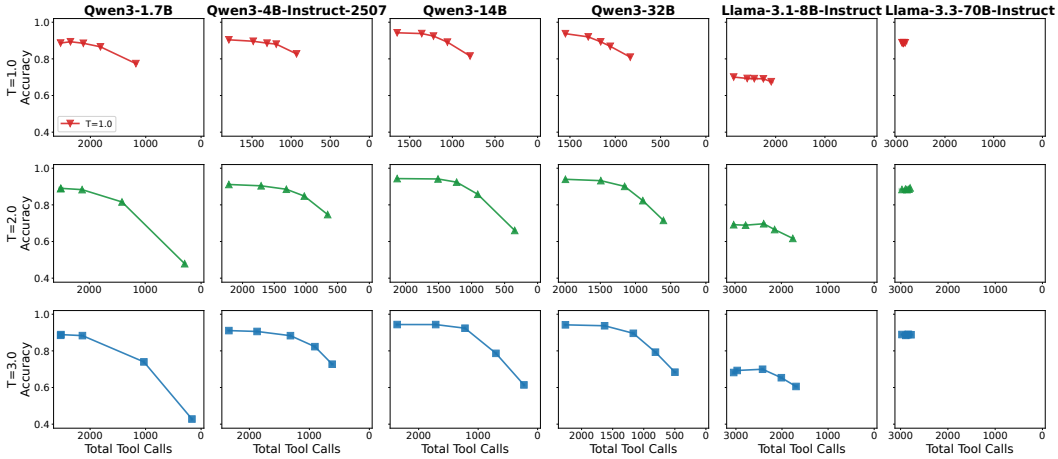

Figure 7: Temperature scaling: T=1.0 (red), T=2.0 (green, default), T=3.0 (blue). Higher temperature flattens the probe's confidence distribution.

Table 13: Temperature scaling (soft prefill). Each cell: Acc (%) / total tool calls.

Model	T	τ=0.1	τ=0.3	τ=0.5	τ=0.7	τ=0.9
Qwen3-1.7B	1.0	88.6/2539	89.3/2361	88.5/2128	86.6/1819	77.4/1178
	2.0	88.8/2512	89.0/2507	88.3/2128	81.6/1415	47.9/293
	3.0	88.5/2532	88.9/2526	88.3/2139	74.0/1031	42.8/165
Qwen3-4B	1.0	90.4/1797	89.6/1486	88.6/1309	88.0/1191	82.7/931
	2.0	91.1/2216	90.4/1707	88.5/1309	84.8/1026	74.7/657
	3.0	91.1/2342	90.6/1872	88.3/1315	82.3/907	72.7/620
Qwen3-14B	1.0	94.2/1648	93.8/1359	92.4/1221	89.1/1058	81.6/792
	2.0	94.3/2128	94.1/1509	92.4/1227	85.8/907	66.0/347
	3.0	94.4/2362	94.4/1714	92.4/1221	78.7/701	61.4/230
Qwen3-32B	1.0	93.7/1550	92.0/1298	89.3/1161	86.9/1055	80.9/835
	2.0	94.0/1996	93.2/1493	90.1/1155	82.3/896	71.5/604
	3.0	94.2/2260	93.7/1630	89.6/1168	79.3/812	68.4/498
Llama-3.1-8B	1.0	70.1/2852	69.3/2578	69.2/2434	69.1/2250	67.5/2090
	2.0	69.2/3027	68.9/2770	69.7/2381	66.5/2146	61.7/1753
	3.0	68.2/3048	69.3/2973	70.0/2422	65.3/2014	60.6/1696
Llama-3.3-70B	1.0	88.8/2882	88.6/2860	88.9/2841	88.5/2854	88.8/2815
	2.0	88.4/2976	88.6/2902	88.3/2871	88.6/2828	89.2/2804
	3.0	88.9/2971	88.4/2890	88.8/2853	88.8/2780	89.0/2835

E.3 Layer selection

We compare three probe configurations: all layers concatenated, middle layer only, and last layer only. Table 14 reports probe test AUROC and accuracy. All-layer concatenation consistently performs

best, confirming that tool-necessity information is distributed across the network. Single-layer probes remain competitive, with the mid-layer probe slightly outperforming the last-layer probe on most models, suggesting the signal emerges early and persists through the network.

Table 14: Layer selection: probe test AUROC and accuracy.

Model	Layers	AUROC	Accuracy
Qwen3-1.7B	All (29 layers)	0.894	0.847
	Mid (layer 14)	0.835	0.795
	Last (layer 28)	0.863	0.796
Qwen3-4B	All (37 layers)	0.948	0.877
	Mid (layer 18)	0.916	0.860
	Last (layer 36)	0.893	0.805
Qwen3-14B	All (41 layers)	0.957	0.892
	Mid (layer 20)	0.920	0.851
	Last (layer 40)	0.929	0.865
Qwen3-32B	All (65 layers)	0.952	0.885
	Mid (layer 32)	0.921	0.844
	Last (layer 64)	0.916	0.835
Llama-3.1-8B	All (33 layers)	0.927	0.849
	Mid (layer 16)	0.894	0.805
	Last (layer 32)	0.880	0.795
Llama-3.3-70B	All (81 layers)	0.936	0.872
	Mid (layer 40)	0.912	0.839
	Last (layer 80)	0.908	0.828

E.4 Data efficiency

We subsample the 900-example training set to $\{10\%, 25\%, 50\%, 75\%, 100\%\}$ and retrain the probe. Table 15 shows that even with only 90 labeled examples (10%), the probe achieves AUROC above 0.81 on all tested models. Performance improves steadily with more data, but the marginal gain diminishes beyond 50%, suggesting the signal is easy to extract with minimal supervision.

Table 15: Data efficiency: probe test AUROC at varying training fractions.

Model	10%	25%	50%	75%	100%
Qwen3-1.7B	0.813	0.872	0.880	0.888	0.894
Qwen3-4B	0.894	0.926	0.936	0.943	0.948
Qwen3-14B	0.882	0.929	0.937	0.949	0.957
Qwen3-32B	0.887	0.926	0.942	0.948	0.952
Llama-3.1-8B	0.823	0.885	0.914	0.920	0.927
Llama-3.3-70B	0.826	0.908	0.916	0.931	0.936

E.5 Regularization strength

We sweep the L2 regularization parameter $\lambda \in \{1, 10, 100, 1000, 10000, 100000\}$ (sklearn $C = 1/\lambda$). Table 16 shows that probe performance is stable across four orders of magnitude (λ=10 to 10000), with a modest decline at very low regularization (λ=1) and very high regularization (λ=100000). The default λ=10000 used throughout the paper is near-optimal across all models.

Table 16: Regularization strength: probe test AUROC.

Model	λ=1	10	100	1000	10000	100000
Qwen3-1.7B	0.863	0.873	0.882	0.892	0.894	0.879
Qwen3-4B	0.936	0.940	0.944	0.949	0.948	0.935
Qwen3-14B	0.947	0.951	0.952	0.955	0.957	0.947
Qwen3-32B	0.935	0.940	0.944	0.948	0.952	0.947
Llama-3.1-8B	0.907	0.916	0.922	0.928	0.927	0.905
Llama-3.3-70B	0.915	0.921	0.925	0.929	0.936	0.934

F Inference overhead

Table 17 reports the computational overhead of PROBE&PREFILL at inference time. The probe requires two operations beyond the standard generation pipeline: (1) extracting the hidden state at the last token position from the prefill forward pass, and (2) applying the linear probe (standardization + dot product + sigmoid). The forward pass itself is *not* additional cost, as it is already required to build the KV cache before autoregressive generation begins. The only overhead is reading the hidden states from this existing forward pass and running the probe.

Across all six models, the total additional latency is under 0.7 ms, compared to a typical prefill forward pass of 10–100 ms and per-token generation of 5–50 ms. This represents less than 1% overhead, making PROBE&PREFILL essentially free at inference time.

Table 17: Inference overhead of PROBE&PREFILL. The forward pass is shared with standard generation (no additional cost). The only overhead is hidden state extraction and the linear probe.

Model	Layers	Hidden dim	Probe dim	Overhead (ms)
Qwen3-1.7B	29	2,048	59,392	0.38
Qwen3-4B	37	2,560	94,720	0.38
Llama-3.1-8B	33	4,096	135,168	0.35
Qwen3-14B	41	5,120	209,920	0.40
Qwen3-32B	65	5,120	332,800	0.57
Llama-3.3-70B	81	8,192	663,552	0.69

G Generalization to agentic search (Search-o1 benchmarks)

To evaluate whether PROBE&PREFILL generalizes beyond WHEN2TOOL, we apply it to six open-domain QA benchmarks from the Search-o1 framework [Li et al., 2025a]: NQ, TriviaQA, HotpotQA, 2WikiMultihopQA, Bamboogle, and MuSiQue. These benchmarks cover single-hop factual QA (NQ, TriviaQA), two-hop reasoning (HotpotQA, 2WikiMultihopQA), and complex multi-hop reasoning (MuSiQue, Bamboogle). We use Qwen3-4B-Instruct for all the evaluations.

Setup. We follow the same experimental design as WHEN2TOOL: 5 Prompt-only modes and 5 Reason-then-Act modes as baselines, plus PROBE&PREFILL with threshold sweep. We train the probes on a 50/50 split of each dataset. The probe trains in seconds on CPU.

Summary. Table 18 compares the best baseline (highest accuracy among all Prompt-only and Reason-then-Act settings) against the best PROBE&PREFILL operating point for each dataset. On 4 of 6 datasets, PROBE&PREFILL achieves comparable or better accuracy while reducing search calls more than the best baseline. On TriviaQA, PROBE&PREFILL achieves 69.2% accuracy (vs. best baseline 68.8%) with 20% fewer searches compared to the baseline's 16%. On HotpotQA and Bamboogle, PROBE&PREFILL *exceeds* all baselines in accuracy while using 50–54% fewer searches. MuSiQue is the exception: this 3–4 hop dataset requires nearly all questions to be searched, and the best baseline (PO Force, −56%) achieves stronger reduction than PROBE&PREFILL (−48%).

Table 18: Search-o1 generalization summary. All on 50% held-out test split. ΔTC = search call reduction relative to Default. **Bold** = probe beats all baselines.

		Default		Best Baseline			PROBE&PREFILL		
Dataset	N	Acc	TC	Acc	TC	ΔTC	Acc	TC	ΔTC
NQ	250	44.8	263	**44.8**	252	−4%	42.0	248	**−6%**
TriviaQA	250	69.6	344	68.8	289	−16%	**69.2**	275	**−20%**
HotpotQA	250	26.0	802	27.2	713	−11%	**28.8**	404	**−50%**
2Wiki	250	36.4	670	38.4	561	−16%	**39.2**	297	**−56%**
Bamboogle	63	25.4	188	33.3	145	−23%	**34.9**	87	**−54%**
MuSiQue	250	19.6	829	**20.4**	362	**−56%**	**20.4**	431	−48%

Probe quality. Table 19 reports probe AUROC on the held-out test split. We compare two probes: (1) WHEN2TOOL *transfer*, the probe trained on WHEN2TOOL and applied directly without any retraining, and (2) *in-domain*, a probe trained on 250 in-domain examples from each dataset. The WHEN2TOOL transfer probe achieves AUROC 0.67–0.80, confirming that the tool-necessity signal learned on our controlled benchmark partially transfers to real-world QA tasks. The in-domain probe (trained in seconds) achieves AUROC 0.64–0.84, with the strongest signal on TriviaQA (0.84) and 2WikiMultihopQA (0.80).

Table 19: Probe AUROC on Search-o1 benchmarks (test split).

Dataset	N	WHEN2TOOL transfer	In-domain
NQ	250	0.731	0.746
TriviaQA	250	0.787	0.835
HotpotQA	250	0.742	0.798
2Wiki	250	0.803	0.803
Bamboogle	63	0.675	0.802
MuSiQue	250	0.695	0.640

Full results. Tables 20–20 report complete results across all Prompt-only, Reason-then-Act, and PROBE&PREFILL settings. Acc = accuracy (substring match), EM = exact match, TC = total search calls.

Table 20: Full results on Search-o1 QA benchmarks (test split only). Acc = accuracy (%), EM = exact match (%), TC = total searches. Baselines evaluated on 250 test items (63 for Bamboogle); PROBE&PREFILL evaluated on the probe's held-out test split.

		NQ			TriviaQA			HotpotQA			2Wiki			Bamboogle			MuSiQue		
		Acc	EM	TC	Acc	EM	TC	Acc	EM	TC	Acc	EM	TC	Acc	EM	TC	Acc	EM	TC
Prompt-only	F	44.8	30.4	252	68.8	58.8	289	26.0	22.8	367	36.0	30.8	312	30.2	28.6	78	20.4	18.4	362
	D	44.8	31.6	263	69.6	60.4	344	26.0	24.0	802	36.4	30.8	670	25.4	23.8	188	19.6	16.8	829
	N	42.8	30.4	377	67.6	58.0	465	26.4	24.0	726	41.2	35.2	706	23.8	22.2	180	20.0	17.6	695
	S	42.0	30.0	353	65.6	58.0	347	24.8	21.6	452	34.8	31.2	477	23.8	20.6	105	12.8	10.4	415
	X	35.6	24.8	170	53.2	46.8	137	19.2	17.2	166	28.8	26.8	159	23.8	22.2	58	8.8	6.8	157
Reason-then-Act	F	45.6	31.6	272	68.8	58.8	363	27.2	24.8	713	38.4	32.0	561	33.3	31.7	145	20.0	17.2	692
	D	42.0	28.8	369	69.2	60.0	444	23.6	21.2	842	38.0	31.6	785	27.0	25.4	195	19.2	17.2	899
	N	42.0	29.2	314	67.2	58.0	366	24.0	20.8	669	37.6	31.6	688	23.8	22.2	175	17.2	15.6	690
	S	40.8	29.6	363	65.6	56.4	366	23.2	21.2	631	33.6	30.0	592	25.4	23.8	144	16.0	12.8	568
	X	39.6	27.2	255	60.8	52.0	247	21.6	19.6	301	28.0	25.6	295	23.8	20.6	80	10.0	8.4	263
PROBE&PREFILL	τ=.1	41.6	29.2	263	69.2	60.4	275	28.8	25.6	404	38.4	33.2	337	33.3	31.7	97	18.8	17.2	471
	τ=.3	42.0	28.8	248	68.0	60.0	211	28.4	26.8	379	39.2	32.4	297	27.0	25.4	82	18.4	16.0	433
	τ=.5	41.2	27.6	233	60.4	53.6	181	25.6	23.2	372	36.4	30.4	266	34.9	34.9	87	20.4	18.0	431
	τ=.7	40.8	27.6	194	56.4	50.0	117	26.4	24.4	274	33.6	28.8	175	31.7	27.0	80	20.0	16.8	397
	τ=.9	33.6	22.4	92	46.4	41.2	33	20.0	18.8	175	32.0	25.6	109	33.3	30.2	37	13.2	11.6	285

H Additional baseline: Compare PROBE&PREFILL with Supervised Fine-Tuning (SFT)

To provide a stronger baseline, we compare PROBE&PREFILL against supervised fine-tuning (SFT) that directly modifies model weights to learn tool-call decisions. We note that this comparison is inherently asymmetric: SFT requires full fine-tuning on multiple GPUs for hours, while PROBE&PREFILL trains a linear probe in seconds on CPU with no weight modification.

H.1 Original training setting: 900 examples

Training data collection. We construct SFT training data from the 900 single-hop training tasks using the same binary labels as the probe. For each task, we first evaluate the model without tool access. If the model answers correctly (tool unnecessary, y=0), we use that direct-answer trajectory as the training target. If the model fails (tool necessary, y=1), we run the model with the Default prompt and tools available, collecting the full multi-round trajectory (tool call, tool response, and final answer). This gives the model examples of both when to answer directly and when to call tools.

Training setup. We perform full-parameter fine-tuning using the HuggingFace Trainer with gradient checkpointing across 4 GPUs. We train for 2 epochs with learning rate 10^{-5} and mask all non-assistant tokens (system prompt, user messages, tool responses) so the model only learns from its own response tokens at each turn. We evaluate on Qwen3-1.7B, Qwen3-4B-Instruct, and Llama-3.1-8B.

Results. Table 21 compares SFT against the Default baseline and PROBE&PREFILL (τ=0.5). SFT improves accuracy by 2–3% across all three models, which is expected since the training data provides correct trajectories directly. However, SFT does not consistently reduce tool calls: on Qwen3-4B, tool calls slightly increase, and on Llama-3.1-8B they also increase. Only on Qwen3-1.7B does SFT achieve meaningful reduction (-18%). In contrast, PROBE&PREFILL at τ=0.5 reduces tool calls by 21–38% on the all models.

Table 21: SFT (full fine-tuning) vs. PROBE&PREFILL (τ=0.5). SFT improves accuracy but does not reliably reduce tool calls. PROBE&PREFILL achieves larger TC reductions with no weight modification. Mean over 3 runs.

Model	Method	Acc (%)	Total TC
Qwen3-1.7B	Default baseline	88.2	2709
	SFT	**91.0**	2211
	PROBE&PREFILL (τ=0.5)	88.3	**2128**
Qwen3-4B-Inst.	Default baseline	89.2	2118
	SFT	**91.3**	2140
	PROBE&PREFILL (τ=0.5)	88.5	**1309**
Llama-3.1-8B	Default baseline	79.5	3708
	SFT	**81.7**	3777
	PROBE&PREFILL (τ=0.5)	69.7	**2381**

Discussion. SFT learns to produce better answers overall but does not learn the tool-call *decision boundary* effectively. In contrast, PROBE&PREFILL works consistently across all model sizes (1.7B to 70B), requires only seconds of CPU training, and provides a smooth accuracy–efficiency tradeoff via the threshold τ. These results highlight that PROBE&PREFILL offers an effective, nearly zero-cost solution compared to the substantially more expensive SFT alternative.

H.2 Larger training setting: 6,000 examples

To test whether the limited call reduction of the 900-example SFT baseline is due to training-set size, we extend supervised training to 6,000 examples on Qwen3-4B-Instruct and evaluate both full-parameter SFT and LoRA with rank $r = 16$. The supervision remains model-specific: direct-answer and tool-use targets are determined by the model's measured no-tool outcomes, rather than by a separate teacher model. We evaluate on the same 2,250 single-hop test tasks over three runs.

Table 22: Scaled-data SFT and LoRA on Qwen3-4B-Instruct. Accuracy is reported as mean $\pm$ standard deviation over three runs; calls/task is the mean number of tool calls per test task. The probe uses its original 900-example training set.

Method	Acc (%)	Calls/task
Default baseline	89.2 ± 0.2	0.941
SFT, 900 examples	91.3 ± 0.1	0.951
SFT, 6,000 examples	90.0 ± 0.2	0.998
LoRA, 6,000 examples	89.6 ± 0.3	0.937
PROBE&PREFILL, $\tau = 0.1$	91.1 ± 0.3	0.985
PROBE&PREFILL, $\tau = 0.5$	88.5 ± 0.2	**0.582**

Table 22 shows that scaling this training recipe does not yield substantial tool-call savings: full SFT increases calls to 0.998 per task, while LoRA remains close to Default at 0.937. In contrast, PROBE&PREFILL at $\tau = 0.5$ reduces calls by approximately 38% relative to Default, with accuracy decreasing from 89.2% to 88.5%. These results characterize the evaluated training recipes and do not rule out stronger objectives or alternative fine-tuning configurations for calibrated tool use.

I Additional baselines: Compare PROBE&PREFILL with Confidence-Based Tool-Use Controllers

We compare the hidden-state probe with three confidence-based controllers on Qwen3-4B-Instruct. All controllers drive the same soft-prefill mechanism and evaluation loop as PROBE&PREFILL, so the comparison changes the decision signal while preserving the downstream intervention. We evaluate on 2,250 single-hop test tasks and report means over three runs.

Decision signals. **Verbalized confidence** asks the model for a score from 0 to 100 indicating its confidence in answering without tools; we convert confidence c to a necessity score $1 - c/100$. **First-token likelihood** uses the model's probability of emitting a tool-call token at the first generated position. **Self-ask** asks whether the model can solve the task without tools and uses the probability assigned to "No," normalized over the "Yes" and "No" alternatives. All scores are oriented so that larger values favor tool use.

Thresholds and comparison protocol. For each confidence baseline, we set the decision cutoff to the empirical score quantiles $q \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ on the evaluated tasks. For PROBE&PREFILL, we use the actual probe-output thresholds $\tau \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$. Scores below the cutoff trigger the direct-answer prefill; otherwise they trigger the tool-use prefill. Thus, the columns denote quantile cutoffs for the baselines and probability thresholds for the probe, and do not necessarily represent matched call budgets. Increasing either cutoff favors skipping tools. Repeated verbalized-confidence operating points reflect tied scores.

Table 23: Confidence-based controllers versus PROBE&PREFILL on Qwen3-4B-Instruct (soft prefill, 2,250 tasks, three runs). Each cell reports accuracy (%) / mean calls per task. Column values are score quantiles q for the three baselines and probe-output thresholds τ for PROBE&PREFILL. Compare the reported call counts when assessing accuracy at similar budgets.

Controller	q or $\tau = 0.1$	0.3	0.5	0.7	0.9
Self-ask	88.6 / 0.949	87.1 / 0.757	85.5 / 0.620	81.4 / 0.497	77.5 / 0.361
Verbalized confidence	89.5 / 1.025	89.5 / 1.025	89.5 / 1.025	82.4 / 0.519	78.2 / 0.356
First-token likelihood	86.7 / 0.936	85.6 / 0.760	82.9 / 0.597	78.4 / 0.449	75.2 / 0.324
PROBE&PREFILL	91.1 / 0.985	90.4 / 0.759	88.5 / 0.582	84.8 / 0.456	74.7 / 0.292

Results. At approximately 0.76 calls per task, PROBE&PREFILL achieves 90.4% accuracy, compared with 87.1% for self-ask and 85.6% for first-token likelihood. At $\tau = 0.5$, the probe achieves 88.5% accuracy with 0.582 calls per task, exceeding the nearby self-ask and first-token operating points in accuracy while using fewer calls. At $\tau = 0.7$, it also achieves higher accuracy with fewer calls than the verbalized-confidence controller at $q = 0.7$. The most aggressive probe threshold yields the fewest calls but lower accuracy than the baselines' most aggressive listed points, illustrating the remaining accuracy–efficiency trade-off.

J Necessity Supervision versus Difficulty Supervision

To examine whether the benchmark's difficulty tiers alone provide an adequate supervision signal, we train a second hidden-state probe using easy tasks as negative labels and hard tasks as positive labels. Medium tasks are excluded from this probe's training, leaving 600 easy/hard examples from the 900-example training split. The necessity probe uses all 900 examples labeled by the model's measured no-tool outcomes. Both probes use all-layer hidden-state features, standardization, and L2-regularized logistic regression; the difficulty probe uses inverse regularization strength $C = 10^{-4}$. We evaluate both probes through the same soft-prefill pipeline on all 2,250 single-hop test tasks, including medium tasks, using Qwen3-4B-Instruct over three runs.

Table 24: Necessity versus difficulty supervision on Qwen3-4B-Instruct. Each cell reports mean accuracy (%) / mean calls per task over three runs. Both rows use actual probe-output thresholds τ. Difficulty supervision excludes medium training examples; necessity supervision uses measured no-tool outcomes across all tiers.

Probe	$\tau = 0.1$	$\tau = 0.3$	$\tau = 0.5$	$\tau = 0.7$	$\tau = 0.9$
Necessity	91.1 / 0.985	90.4 / 0.759	88.5 / 0.582	84.8 / 0.456	74.7 / 0.292
Difficulty-only	90.9 / 0.930	85.5 / 0.644	83.9 / 0.538	82.2 / 0.469	75.9 / 0.312

Results and scope. At $\tau = 0.7$, the necessity probe achieves 84.8% accuracy with 0.456 calls per task, compared with 82.2% accuracy and 0.469 calls for the difficulty-only probe. At other thresholds, the probes reach different operating points; neither row dominates every point in the other row. The results support the usefulness of model-specific necessity supervision beyond this easy-versus-hard control. They do not establish that the hidden-state probe is independent of surface difficulty cues, and the comparison also differs in training-set size and coverage of medium tasks.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes], [No], or [N/A].
- [N/A] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a proper justification is given (e.g., error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: All claims are supported by experimental results in Sections 3–5 and the Appendix.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

Justification: Dataset-specific limitations on MuSiQue is discussed in Appendix G.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: The paper does not include theoretical results.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Benchmark tasks are generated with fixed random seeds (Section 2, Appendix A). Probe training details and hyperparameters are specified in Sections 4–5 and Appendix E.

   Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Code and data will be released upon acceptance. The benchmark is fully self-contained with deterministic task generators.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.

- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Data splits (Section 2), probe training (Section 4), and hyperparameter ablations (Appendix E) are all specified.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We report mean$\pm$std over 3 runs for all results (Appendix B, C). Standard deviations are computed across independent runs with different random seeds.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: Inference overhead is reported in Appendix F. All models are served via vLLM on GPUs. Probe training takes seconds on CPU.

   Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conforms with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This is foundational research on understanding LLM internal representations for tool-call decisions. We do not foresee direct negative societal impacts.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: We release only a benchmark of synthetic tasks and lightweight linear probes, which pose no risk for misuse.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All models and datasets are cited. Models are used under their respective open-source licenses.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: WHEN2TOOL is documented in Section 2 and Appendix A with full environment descriptions, task examples, and design principles.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve crowdsourcing or human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This paper does not involve human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLMs are used only for writing, editing, or formatting purposes.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.