Ask, Answer, or Act: Learning Interaction Modes for Tool-Use Agents

Anonymous ACL submission

Abstract

Robust tool-use agents must navigate mixed-intent interactions, deciding whether to ask for missing information, answer directly, or invoke tools. We tackle this challenge by introducing a novel task formulation that structures the planner's action space into explicit interaction instructions over an explicit Ask/Answer/Act space, thereby making these behavioral boundaries an explicit target of optimization. In this framework, a planner predicts the interaction mode and semantic payload, while a frozen executor realizes the instruction into an observable behavior. Coupled with this structured decision process, we propose Executor-Grounded Failure-Routed Planner Learning (EFPL). EFPL optimizes the planner via executor-observed outcomes and strategically leverages privileged feedback for targeted correction when outcome rewards provide insufficient guidance, further refining these behavioral boundaries. Experiments on BFCL and When2Call across three backbone sizes show that EFPL consistently improves clarification, abstention, when-to-call decisions, and robustness to irrelevant tools, while fully preserving strong multi-turn tool-calling performance.

1 Introduction

A robust tool-use agent should not blindly treat every user request as an actionable command. Prior to API invocation, it must evaluate the context to determine if the request is fully specified, answerable from the available context, or beyond the scope of available tools (Huang et al., 2024; Ross et al., 2025; Zhang et al., 2025b). Errors in this judgment manifest as hallucination failures: fabricating tool arguments, invoking APIs unnecessarily, or forcing actions on unanswerable queries. Figure 1 illustrates this boundary problem. This motivates us to study tool use as a high-level interaction planning problem, explicitly modeling the decision space

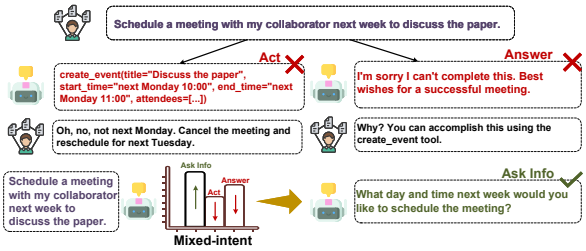

Figure 1: Illustration of mixed-intent boundary errors. For the same underspecified scheduling request, ACT prematurely fabricates tool arguments and ANSWER incorrectly refuses a feasible task, whereas the correct behavior is to ASK for the missing date and time.

to route requests toward user clarification, direct response, or grounded tool execution.

Existing tool-use training often optimizes models through end-to-end outputs, such as final responses, tool calls, or full action trajectories (Schick et al., 2023; Qian et al., 2025). This is suitable when the desired behavior has a clear execution form, but mixed-intent interactions require a training target that can represent different kinds of next-step behaviors. We use an instruction-level planning interface to make this intended next step explicit, specifying both the interaction mode and the information needed to carry it out. This turns the otherwise implicit choice among asking, answering, abstaining, and acting into a concrete object of optimization.

However, the planning instruction is only useful if it can be evaluated by the behavior it induces (Li et al., 2026c). Consequently, we evaluate planner outputs after they are grounded by a frozen executor, utilizing the subsequent response or tool interaction as the supervision signal. Through this approach, the planner is optimized for the practical behaviors its instructions elicit.

While outcome-based reinforcement learning effectively aligns planning with task success, this reward-driven selection inherently falters when the

correct interaction mode is absent from all sampled trajectories. If a policy repeatedly forces tool execution instead of exploring the necessary clarification or abstention behavior, scalar rewards merely penalize these errors without explicitly teaching the missing paradigm. Drawing on self-distillation for dense correction, we address this supervision void by re-evaluating failed rollouts under a privileged context. Encouraging the policy to move toward this privileged-context-conditioned teacher distribution provides dense, token-level supervision to correct the behavioral boundary while preserving the on-policy formulation (Hubotter et al., 2026; Zhao et al., 2026).

However, recent studies caution that unconditionally replacing reward-driven learning with self-distillation risks destabilizing the policy (Shen et al., 2026; Kim et al., 2026). We introduce Executor-Grounded Failure-Routed Planner Learning (EFPL), which employs self-distillation strictly as a conditional repair mechanism. As long as executor outcomes yield valid preferences among samples, updates remain efficiently reward-driven. EFPL intervenes with privileged-context distillation only when all sampled instructions completely fail, supplying a directional signal to help guide the planner toward the missing interaction mode. Our main contributions are summarized as follows:

- We formalize mixed-intent tool use as an executor-grounded instruction-level planning problem over an explicit ASK/ANSWER/ACT space. This formulation makes the boundary among clarification, direct response or abstention, and tool execution an explicit object of planner optimization.
- We propose Executor-Grounded Failure-Routed Planner Learning (EFPL), a planner optimization framework that routes learning signals according to executor-observed outcomes. EFPL provides corrective planner-level supervision and further strengthens the behavioral boundaries among clarification, direct response or abstention, and tool execution.
- Experiments on BFCL and When2Call across three backbone sizes show that EFPL consistently improves boundary-oriented behaviors, including clarification, abstention, and robustness to irrelevant tool use, while maintaining strong multi-turn tool-calling performance.

2 Related Work

Tool-use agents. Tool-augmented language models extend LLMs by invoking APIs, retrieving information, executing programs, or interacting with external environments. Prior work has studied self-supervised API use, interleaved reasoning and acting, program-like tool trajectories, large-scale tool instruction tuning, and function-calling datasets or benchmarks (Parisi et al., 2022; Schick et al., 2023; Yao et al., 2023; Paranjape et al., 2023; Patil et al., 2024; Liu et al., 2024; Zhang et al., 2025a; Liu et al., 2025; Lin et al., 2024). These studies mainly improve tool selection, argument grounding, and executable function calling. Recent work further shifts attention from how to call tools to whether tool use is appropriate under underspecified, irrelevant, or unanswerable requests (Huang et al., 2024; Zhang et al., 2024; Lu et al., 2025; Ross et al., 2025; Zhang et al., 2025b). Our work follows this mixed-intent direction, but optimizes an executor-grounded planner over an explicit ASK/ANSWER/ACT space.

Reinforcement learning for tool-use agents. Reinforcement learning has become a prominent post-training paradigm for LLM reasoning and agentic behavior (Shao et al., 2024; DeepSeek-AI et al., 2025a). In tool-use settings, execution results, search interaction, code execution, and task success have been used as outcome feedback for learning tool-use strategies and multi-step agent behaviors (Qian et al., 2025; Li et al., 2025b; Feng et al., 2025; Jin et al., 2025; Wu et al., 2024; Li et al., 2026c). These methods demonstrate the value of executable outcomes as supervision. In contrast, we explicitly optimize the behavioral boundaries among ASK, ANSWER, and ACT, refining interaction logic without compromising foundational tool-calling and planning capabilities.

Self-distillation for online supervision. Self-distillation provides dense online supervision by conditioning the policy on privileged information or feedback contexts (Hubotter et al., 2026; Zhao et al., 2026; Ye et al., 2026; Shenfeld et al., 2026). However, recent studies show that privileged self-distillation can accelerate learning, but may also introduce information leakage, unstable long-horizon training, optimization ambiguity on correct samples, or degraded teacher reliability (Yang et al., 2026; Li et al., 2026a; Shen et al., 2026; Kim et al., 2026). These findings motivate using self-

distillation as targeted correction rather than as a replacement for reward-grounded optimization.

3 Problem Formulation and Architecture

Figure 2 illustrates the overall decoupled architecture and the training flow. We next formalize its core components.

3.1 Instruction-Level Interaction-Mode Planning

We formulate tool use as an interaction-mode planning problem. Let $x = (h, u, \mathcal{T})$ denote the planner input, where h is the dialogue history, u is the current user request, and $\mathcal{T}$ is the set of available tools. Instead of directly producing a final response or a raw API call, the planner generates a structured planning instruction $y = (m, s)$, where $m \in \{\text{ASK}, \text{ANSWER}, \text{ACT}\}$ specifies the interaction mode, and s provides the mode-specific semantic payload to be realized by the executor.

The payload s defines what the executor should realize under the selected mode. For ACT, it specifies the target tool together with the available constraints and arguments required to instantiate a valid tool call. For ASK, it identifies the missing information or clarification goal that should be requested from the user. For ANSWER, it specifies the direct response, refusal, or abstention behavior to be delivered without tool execution. Thus, the planner does not emit a surface-form response; it emits an executable intent that can be grounded by the downstream executor.

This instruction-level interface separates high-level interaction intent from low-level realization. The planner is trained to control behavioral boundaries: when to solicit additional information, when to answer or abstain without tools, and when to act by selecting an appropriate tool and specifying the corresponding call arguments.

3.2 Execution-Grounded Outcome Evaluation

Given a planning instruction $y_i = (m_i, s_i)$, a frozen executor E grounds it into an observable outcome $o_i = E(y_i)$, where o_i is the behavior realized in the selected interaction mode. The executor serves as a fixed grounding interface between planning instructions and observable behavior. Planner learning is therefore outcome-based. An instruction y_i is judged not by its surface similarity to a reference planning trace, but by whether its achieved result o_i satisfies the task requirement. This shifts supervision from textual imitation of plans to optimization of the behaviors those plans induce.

3.3 Executor-Grounded Reward and Feedback

We use a reward manager Φ to compare the result achieved with the ground truth of the executor g:

$$(R_i, f_i) = \Phi(x, y_i, o_i, g). \tag{1}$$

R_i is the scalar reward used for reward-based planner optimization, and f_i is a diagnostic feedback signal used for corrective self-distillation.

The scalar reward consists of a general format component and an execution outcome correctness component $R_i = R_i^{\text{fmt}} + R_i^{\text{exec}}$. The format reward R_i^{fmt} is applied to every planner output and checks whether the instruction follows the required schema. The execution outcome correctness reward R_i^{exec} is computed according to the selected mode. For ACT, we adopt a ToolRL-style verifiable tool-use reward that evaluates the selected tool and its arguments against the executor ground truth (Qian et al., 2025). For ASK and ANSWER, we use an additional evaluation model to assign reward values by assessing semantic correctness and content completeness. Details of the reward computation are provided in Appendix B.

The feedback signal f_i is constructed from the ground-truth on the executor-side and a reference instruction, which are drawn from the unified training corpus described in Section 5. Following the principles of self-distillation, we leverage this signal to instantiate a self-teacher over the original context (Li et al., 2026a). This teacher distribution provides dense, token-level supervision that suppresses the original boundary error and actively shifts the planner toward the correct ASK/ANSWER/ACT behavior.

Thus, execution yields two complementary learning signals: scalar rewards for GRPO-style reinforcement learning (DeepSeek-AI et al., 2025a), and privileged feedback for context-conditioned self-distillation when reward-only is insufficient. Section 4 formalizes how these signals are routed into the two optimization branches.

4 Executor-Grounded Failure-Routed Planner Learning

We present Executor-Grounded Failure-Routed Planner Learning (EFPL), a framework designed to optimize the Planner within a decoupled Planner-Executor architecture shown in Figure 2.

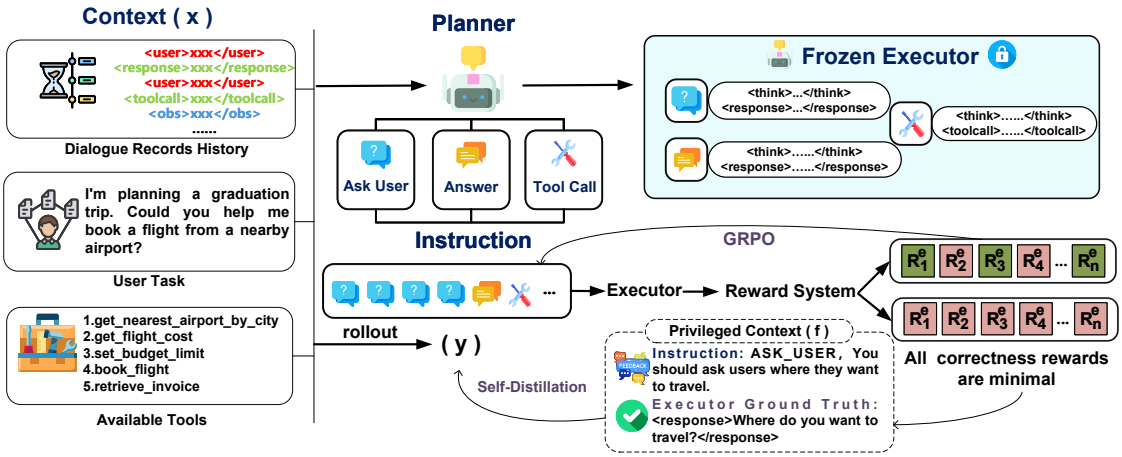

Figure 2: Overview of the framework. Given the context x, the planner samples instruction-level decisions y over ASK, ANSWER, and ACT. A frozen executor grounds each instruction into an observable outcome, which is evaluated by a reward system. Rollout groups with informative reward differences are optimized through GRPO, whereas completely failed groups are routed to privileged-context self-distillation.

4.1 Executor-Grounded Trajectory Routing

For a given context x, the planner samples a group of G instructions, which are subsequently processed by the frozen executor to yield final outcomes. Let $\mathcal{R}_G^{\text{exec}}$ be the set of strict correctness scores evaluated on these outcomes. We define a binary routing indicator for each rollout y_i, where the routing decision is made at the group level:

$$z_i^{\text{Teach}} = \mathbf{1}(\max(\mathcal{R}_G^{\text{exec}}) = r_{\min}) \tag{2}$$

where $r_{\min}$ denotes the lowest execution outcome reward. By routing these critically failed batches to a dense self-distillation branch, we provide direct supervision to rectify the high-level action planning. Instances with partially or fully successful trajectories are directed to the reinforcement learning branch ($z_i^{\text{RL}} = 1 - z_i^{\text{Teach}}$) for further refinement. This routing mechanism naturally establishes an implicit curriculum: early-stage structural failures in action selection trigger dense corrective signals, progressively shifting toward reward-driven optimization as the planner learns to generate precise instructions.

4.2 Reward-Guided Planner Reinforcement

For valid trajectories routed to the reinforcement learning branch ($z_i^{\text{RL}} = 1$), we optimize the planner to reinforce instructions that lead to successful executor outcomes. Given the sampled group of instructions $\{y_1, \ldots, y_G\}$ within the *ask–answer–act* decision space, the reward manager evaluates their corresponding executor outcomes to assign scalar rewards $\{R_1, \ldots, R_G\}$. We then formulate the optimization signal by computing a group-relative advantage:

$$A_i = \frac{R_i - \mu_R}{\sigma_R + \epsilon_{\text{std}}}, \tag{3}$$

where μ_R and σ_R denote the mean and standard deviation of the rewards in the group, and ϵ_{std} is a small constant. The planner policy is updated via the standard clipped surrogate objective:

$$\mathcal{L}_{\text{RL}}^{(i)}(\theta) = -\frac{1}{|y_i|}\sum_t \min\Big(\rho_{i,t}(\theta)A_i, \; \text{clip}\left(\rho_{i,t}(\theta), 1-\epsilon, 1+\epsilon\right)A_i\Big), \tag{4}$$

where $|y_i|$ denotes the sequence length of the i-th instruction, and t indexes the tokens. $\rho_{i,t}(\theta)$ is the token-level importance sampling ratio.

4.3 Privileged-Guided Planner Correction

For the i-th trajectory within a failure-routed group ($z_i^{\text{Teach}} = 1$), we use privileged feedback f, constructed from the executor-side ground truth and reference instruction, and formulate the sequence-level distillation objective as follows:

$$\mathcal{L}_{\text{Teach}}^{(i)}(\theta) = \frac{1}{|y_i|}\sum_t \mathcal{D}\Big(\pi_\theta(\cdot \mid x, y_{<t}^{(i)}) \; \| \; \text{stopgrad}\left[\pi_{\bar{\theta}}(\cdot \mid x, f, y_{<t}^{(i)})\right]\Big). \tag{5}$$

In this objective, $\mathcal{D}$ denotes a distribution-matching loss and π_θ is the student planner being optimized, while $\pi_{\bar{\theta}}$ serves as the teacher, which is conditioned on the privileged-context feedback

f. The stopgrad operation ensures a stable target distribution. Rather than discarding the fully failed rollout y_i, this formulation re-evaluates its past states $y_{<t}^{(i)}$ under the teacher's guidance. This provides the planner with dense, token-level corrective signals (Li et al., 2026a). The privileged-context is used only to construct the teacher distribution during training and is never exposed to the planner or executor at inference time.

4.4 Optimization Objective

Based on the routing indicators z_i^{RL} and z_i^{Teach} defined in Section 4.1, our primary goal is to minimize a unified objective that alternates between environment exploration and privileged correction:

$$\mathcal{L}_{\mathrm{total}}(\theta) = \mathbb{E}\Bigg[\frac{1}{G}\sum_{i=1}^{G}\Big(z_i^{\mathrm{RL}}\mathcal{L}_{\mathrm{RL}}^{(i)} + \alpha z_i^{\mathrm{Teach}}\mathcal{L}_{\mathrm{Teach}}^{(i)}\Big) + \beta D_{\mathrm{KL}}\big(\pi_\theta \,\|\, \pi_{\mathrm{ref}}\big)\Bigg] \tag{6}$$

where $\mathcal{L}_{\mathrm{RL}}$ follows Section 4.2 to maximize the rewards. $\mathcal{L}_{\mathrm{Teach}}$ represents the privileged-guided loss in Section 4.3. The detailed algorithmic procedure is provided in Appendix C.1.

5 Experiments Setup

Dataset. We curate a unified training corpus of 6,540 instances from three data components: a ToolRL-derived subset (Qian et al., 2025), APIGen-MT-5k (Liu et al., 2024), and When2Call-train-sft (Ross et al., 2025). This mixture explicitly targets foundational tool composition, multi-turn planning, and critical decision-making such as choosing between tool invocation and user clarification. All details are deferred to the Appendix A.

Baselines. We evaluate EFPL against five baselines across foundational models of varying capacities (Yang et al., 2025). The baselines are categorized as follows:

Vanilla Paradigms. These models directly generate the final outputs end-to-end without architectural decoupling.

- **Base Model**: Direct zero-shot evaluation of the foundational models.
- **Vanilla GRPO**: Optimized end-to-end via vanilla GRPO (DeepSeek-AI et al., 2025a), directly updating the monolithic policy.

Decoupled Paradigms. These models adopt our architecture to explicitly decouple the ASK, ANSWER, and ACT decision space from downstream execution, with Qwen3-8B (Yang et al., 2025) serving as the frozen executor.

- **Base-Planner**: Zero-shot inference using the foundational model as the planner.
- **SFT-Planner**: The planner is fine-tuned using synthetic planning data generated from our unified dataset.
- **GRPO-Planner**: GRPO-Planner is the ablation of EFPL without failure-routed privileged correction. It uses the same planner-executor architecture and the same executor-grounded scalar rewards, but does not route fully failed groups to privileged self-distillation.

Implementation Details. To ensure a strictly fair comparison, all trainable baselines are optimized using the unified dataset, with prompt configurations provided in Appendix E. All experiments are conducted on 16 Ascend 910B NPUs, each with 64GB memory, with additional implementation details provided in Appendix C.2.

Evaluation Benchmarks. To comprehensively evaluate the interactive decision-making and tool-use capabilities, we assess the final generated outputs across two complementary benchmarks.

When2Call. We primarily utilize this benchmark to evaluate the core capability of "when to call a tool." By assessing whether the generated outputs correctly match the required interaction states of *ToolCall*, *Ask Info*, or *Cannot Answer*, this benchmark directly validates the model's fundamental judgment regarding tool-use timing and interaction modes (Ross et al., 2025).

BFCL. We employ five distinct subsets from BFCL to evaluate multi-turn multi-step tool calling capabilities and robustness against hallucinations. Specifically, `multi_turn_base` measures standard multi-turn and multi-step tracking; `multi_turn_miss_param` and `multi_turn_miss_func` assess the system's proactivity in requesting clarification when faced with missing constraints; and `irrelevance` alongside `live_irrelevance` examine the resistance to tool hallucinations, ensuring the model safely responds directly or abstains rather than invoking irrelevant tools (Patil et al., 2025).

Backbone	Paradigm	Method	BFCL Multi-turn				BFCL Irrelevance	When2Call	
			Base	Miss. Param.	Miss. Func.	Avg.		F1	Acc.
Qwen3-1.7B	Vanilla	Base	15.00	12.00	6.00	11.00	76.54	67.74	67.33
		GRPO	17.00	12.50	9.00	12.83	82.35	65.56	66.00
	Decoupled	Base-Planner	8.50	8.50	4.00	7.00	–	55.06	58.67
		SFT-Planner	5.50	5.00	6.00	5.50	**87.19**	59.91	61.00
		GRPO-Planner	20.00	15.00	16.00	17.00	82.74	70.07	69.00
		EFPL	**22.50**	**19.00**	**17.50**	**19.67**	83.36	**71.78**	**71.67**
Qwen3-4B	Vanilla	Base	36.50	31.00	29.50	32.33	78.65	69.85	69.33
		GRPO	40.00	32.00	31.50	34.50	83.19	71.76	70.67
	Decoupled	Base-Planner	28.00	29.00	23.50	26.83	–	67.12	66.67
		SFT-Planner	19.00	7.00	3.50	9.83	83.45	67.15	66.67
		GRPO-Planner	39.50	34.50	35.00	36.33	86.92	74.09	72.67
		EFPL	**42.00**	**36.00**	**35.50**	**37.83**	**88.88**	**77.76**	**76.00**
Qwen3-8B	Vanilla	Base	41.50	27.00	38.50	35.67	82.27	67.84	68.00
		GRPO	42.00	35.50	26.50	34.67	85.06	76.08	76.00
	Decoupled	Base-Planner	31.00	24.50	28.00	27.83	–	67.42	66.33
		SFT-Planner	20.00	7.00	6.00	11.00	76.60	57.17	58.67
		GRPO-Planner	48.00	40.00	43.00	43.67	85.32	74.91	74.00
		EFPL	**49.00**	**42.00**	**46.50**	**45.83**	**88.61**	**78.23**	**77.33**

Table 1: Main results under vanilla and decoupled paradigms. Scores are reported as percentages, and higher values indicate better performance. Avg. denotes the macro-average over the three BFCL Multi-turn subsets: Base, Miss. Param. and Miss. Func. BFCL Irrelevance reports pooled accuracy over all instances from the `irrelevance` and `live-irrelevance` subsets. The best result is bolded and the second-best result is underlined. GRPO-Planner is an internal ablation of EFPL that removes failure-routed privileged correction while retaining the same ASK/ANSWER/ACT planner and executor-grounded rewards.

6 Results and Analysis

6.1 Main Results and Discussion

Table 1 reports the main results on BFCL and When2Call across three backbone sizes under vanilla and decoupled paradigms. Appendix D.1 provides additional comparisons with more open-source base models and closed-source LLMs.

Overall performance of EFPL. Overall, EFPL obtains the highest or tied-highest scores on 20 of the 21 metrics across all three backbone sizes. Compared with the Base Model, EFPL delivers clear gains on boundary-centric evaluations, improving When2Call F1/Accuracy by +7.45/+6.78 points and BFCL Irrelevance by +7.80 points on average. Meanwhile, it also improves the BFCL multi-turn average by +8.11 points, showing that EFPL strengthens boundary awareness while further enhancing multi-turn tool-calling capability. These results demonstrate the overall effectiveness of EFPL in improving the reliability of tool-use agents.

How much does the decoupled architecture contribute? GRPO-Planner improves BFCL multi-turn performance over vanilla GRPO, increasing the average score by +5.00 points across the three backbone sizes. The gains vary across subsets: averaged over backbones, GRPO-Planner improves the standard multi-turn base subset by +2.83 points, missing-parameter cases by +3.17 points, and missing-function cases by +9.00 points. The gain on missing-parameter and missing-function cases indicates that planner-level optimization provides greater benefit in settings where the agent must track dialogue state and determine whether a request is actionable or still underspecified. In vanilla GRPO, a failed rollout may conflate several sources of error, including wrong interaction mode, wrong tool choice, incorrect arguments, and imperfect surface realization. By contrast, planner-level GRPO exposes a more learnable space. Therefore, decomposition does not merely change the output format; it provides a more favorable training interface for long-horizon and ambiguity-sensitive tool-use behavior.

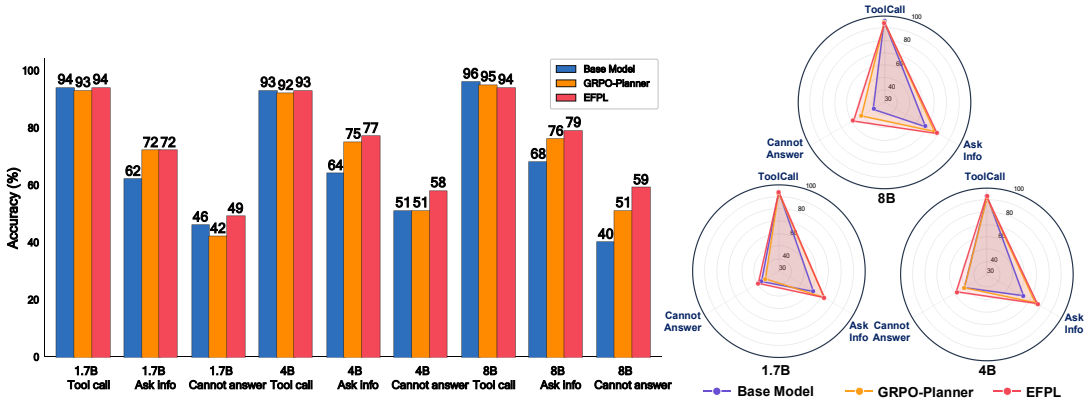

Figure 3: Per-mode accuracy on When2Call across three backbone sizes. The left panel shows grouped accuracy scores for *ToolCall*, *Ask Info*, and *Cannot Answer*. The right panel visualizes each method's decision profile under the same three modes, where a larger and more balanced polygon indicates stronger coverage across tool invocation, clarification, and abstention behaviors.

EFPL gains under the same decoupled architecture. Compared with GRPO-Planner, EFPL brings additional gains. Averaged across the three backbone sizes, EFPL improves GRPO-Planner by +2.11 points on the BFCL multi-turn average, +1.96 points on BFCL Irrelevance, and +3.11 points on When2Call accuracy. The improvements on BFCL Irrelevance and When2Call are particularly informative because they evaluate whether the model can avoid unnecessary tool use and correctly distinguish among tool calling, asking for information, and abstaining when the request cannot be answered. The final gains reflect better interaction-mode selection rather than a simple bias toward tool invocation, indicating stronger behavioral separation among asking, answering, and acting.

Decoupling alone is insufficient without planner learning. Introducing a planner-executor interface does not by itself improve tool use. Base-Planner falls below the vanilla base model on BFCL multi-turn average by 4.00, 5.50, and 7.84 points across Qwen3-1.7B, Qwen3-4B, and Qwen3-8B, respectively, indicating that a zero-shot planner struggles to use the ASK/ANSWER/ACT interface reliably. Naive supervised alignment is also ineffective: SFT-Planner consistently underperforms Base-Planner on multi-turn averages, suggesting that imitation alone may learn the interface format but not the execution-aware routing policy. In contrast, GRPO-Planner substantially improves over Base-Planner and surpasses vanilla GRPO on most metrics, while EFPL further achieves the strongest overall performance, especially on multi-turn tool use and When2Call. These results show that decomposition is only an optimization substrate; its benefit emerges only when planner decisions are aligned with executor-grounded feedback.

Gold mode	Predicted mode			
	Direct	ToolCall	Ask Info	Cannot Answer
GRPO-Planner – Base				
ToolCall	-0.3	-1.0	+2.7	-1.3
Ask Info	0.0	-10.0	+9.7	+0.3
Cannot Answer	+2.7	+0.3	-5.3	+2.3
EFPL – Base				
ToolCall	+0.7	-0.7	+1.3	-1.3
Ask Info	0.0	-11.3	+11.3	0.0
Cannot Answer	-1.0	-1.3	-7.3	+9.7
EFPL – GRPO-Planner				
ToolCall	+1.0	+0.3	-1.3	0.0
Ask Info	0.0	-1.3	+1.7	-0.3
Cannot Answer	-3.7	-1.7	-2.0	+7.3

Table 2: Average changes in row-normalized mode confusion across the three backbone sizes. Each block reports the difference between the two methods shown in its title. Rows are gold modes and columns are predicted modes. Values are percentage-point differences. Positive values indicate that the method before the minus sign predicts the corresponding mode more often.

6.2 Boundary Analysis: Ask, Answer, or Act

Figure 3 investigates the planner's capacity to delineate the decision boundaries across acting, clarifying, and abstaining. The When2Call split provides three ground-truth modes: *ToolCall*, *Ask Info*, and *Cannot Answer*. Within our planner's action space, the *Answer* category also encompasses direct answering. Given the absence of gold direct answering instances in this split, *Direct* is omitted

as a target class in the figure, yet it is preserved in Table 2 to account for potential misclassifications.

The bar plot indicates that performance on *ToolCall* has largely saturated across all methods, with the primary performance disparities emerging in the *Ask Info* and *Cannot Answer* categories. This implies that the core challenge lies not in identifying unambiguously actionable requests, but rather in correctly mapping non-actionable queries to their appropriate modes. The radar plots demonstrate through a clear morphological shift that EFPL improves the weaker clarification and abstention directions while keeping the already saturated tool-use direction stable.

Table 2 details the reallocation of prediction mass across modes. GRPO-Planner primarily reduces incorrect tool invocations, redirecting 10.0% of misclassified *Ask Info* cases back to the correct mode. Compared with GRPO-Planner, EFPL primarily improves the hardest non-actionable boundary. For gold *Cannot Answer* cases, EFPL increases correct abstention predictions by 7.3 points while reducing *Direct*, *ToolCall*, and *Ask Info* misclassifications. This controlled comparison indicates that, under the same decoupled architecture, adding failure-routed correction to reward-guided planner learning mainly improves abstention decisions in *Cannot Answer* cases.

6.3 Robustness Analysis: Executors and Evaluators

Impact of Executor Capacity. To confirm that the system's overall proficiency does not covertly exploit the executor's latent reasoning capacity, we emphasize that the frozen executor is architecturally isolated from the task context, functioning solely as an instruction formatter. We empirically examine this by substituting the default Qwen3-8B (Yang et al., 2025) executor at inference with a diverse spectrum of models including Qwen3-4B (Yang et al., 2025), GPT-4o-mini-2024-07-18, DeepSeek-V3.2 (DeepSeek-AI et al., 2025b). As shown in Table 3, the overall performance remains almost identical across different executors. This indicates that the execution outcome is fundamentally bottlenecked by the planner's ASK/ANSWER/ACT decisions and instruction generation capabilities.

Reliability of When2Call Evaluations. Following When2Call (Ross et al., 2025) and to ensure the reliability of the evaluation, we use GPT-5 (Singh et al., 2026) to categorize each model's free-form output into four explicit output modes according to the behavior it actually expresses. Since this classification depends mainly on these explicit output modes rather than task solving, tool-argument verification, or subjective response quality, the risk of evaluator-induced bias is limited. We additionally validate the classifier against two alternative classifiers, obtaining high agreement and stable aggregate results; full details are reported in Appendix D.3.

Executor	MT Avg.	Irr. Avg.	W2C F1	W2C Acc.
Qwen3-8B	45.83	88.61	78.23	77.33
Qwen3-4B	44.33	89.32	78.47	78.00
GPT-4o-mini	45.33	88.97	78.71	78.00
DeepSeek-V3.2	44.50	89.06	78.81	78.33

Table 3: Executor-controlled fairness analysis with a fixed EFPL-trained Qwen3-8B planner. Scores are reported as percentages. MT Avg. denotes the average over the three BFCL multi-turn subsets, and Irr. Avg. denotes pooled accuracy over the irrelevance and live-irrelevance subsets.

7 Conclusion

In this work, we formulate mixed-intent tool use as executor-grounded instruction-level planning, making interaction-mode boundaries directly optimizable through the behaviors they induce. Building on this formulation, EFPL strengthens boundary learning by combining executor-grounded outcome optimization with targeted correction for fully failed rollout groups. Experiments on BFCL and When2Call show that EFPL improves clarification, abstention, when-to-call decisions, and robustness to irrelevant tool use while preserving strong multi-turn tool-calling performance. Further analyses show that these gains do not come from simply encouraging or suppressing tool calls, but from better separating actionable requests from cases that require clarification or abstention.

Our findings highlight the importance of explicitly learning the behavioral boundaries that govern when an agent should ask, answer, or act. More broadly, this suggests that reliable tool-use agents should not only execute tool calls accurately, but also learn to control interaction modes: asking for missing information before acting, answering or abstaining when tool use is unnecessary or infeasible, and invoking tools only when the task is sufficiently specified and actionable.

Limitations

Although EFPL improves mixed-intent tool-use behavior, it has several limitations. First, the decoupled planner-executor architecture introduces additional inference overhead, since the system must generate an intermediate ASK/ANSWER/ACT instruction before producing the final response or tool call. This clearer training interface may therefore increase latency and deployment complexity. Second, our experiments are conducted on BFCL and When2Call, which cover multi-turn tool use, irrelevance robustness, and when-to-call decisions, but may still fall short of fully open-ended real-world agent scenarios with dynamic tools, noisy observations, and evolving user goals. Extending EFPL to more realistic interactive environments is an important direction for future work.

References

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. 2024. On-policy distillation of language models: Learning from self-generated mistakes. In *The Twelfth International Conference on Learning Representations*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 3416 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *Preprint*, arXiv:2507.06261.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Jun-Mei Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiaoling Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 179 others. 2025a. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645:633 – 638.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Ban Lin, Bing Xue, Bing-Li Wang, Bin Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 244 others. 2025b. Deepseek-v3.2: Pushing the frontier of open large language models. *ArXiv*, abs/2512.02556.

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

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. 2024. Metatool benchmark for large language models: Deciding whether to use tools and which to use. In *The Twelfth International Conference on Learning Representations*.

Jonas Hubotter, Frederike Lubeck, Lejs Deen Behric, Anton Baumann, Marco Bagatella, Daniel Marta, Ido Hakimi, Idan Shenfeld, Thomas Kleine Buening, Carlos Guestrin, and Andreas Krause. 2026. Reinforcement learning via self-distillation. *ArXiv*, abs/2601.20802.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *ArXiv*, abs/2503.09516.

Yiqiao Jin, Yiyang Wang, Lucheng Fu, Yijia Xiao, Yinyi Luo, Haoxin Liu, B. Aditya Prakash, Josiah Hester, Jindong Wang, and Srijan Kumar. 2026. Unisd: Towards a unified self-distillation framework for large language models. *Preprint*, arXiv:2605.06597.

Jeonghye Kim, Xufang Luo, Minbeom Kim, Sangmook Lee, Dohyung Kim, Jiwon Jeon, Dongsheng Li, and Yuqing Yang. 2026. Why does self-distillation (sometimes) degrade the reasoning capability of llms? *Preprint*, arXiv:2603.24472.

Gengsheng Li, Tianyu Yang, Junfeng Fang, Mingyang Song, Mao Zheng, Haiyun Guo, Dan Zhang, Jinqiao Wang, and Tat-Seng Chua. 2026a. Unifying group-relative and self-distillation policy optimization via sample routing. *Preprint*, arXiv:2604.02288.

Minchong Li, Feng Zhou, and Xiaohui Song. 2025a. BiLD: Bi-directional logits difference loss for large language model distillation. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 1168–1182, Abu Dhabi, UAE. Association for Computational Linguistics.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025b. Torl: Scaling tool-integrated rl. *ArXiv*, abs/2503.23383.

Yaxuan Li, Yuxin Zuo, Bingxiang He, Jinqian Zhang, Chaojun Xiao, Cheng Qian, Tianyu Yu, Huan ang Gao, Wenkai Yang, Zhiyuan Liu, and Ning Ding. 2026b. Rethinking on-policy distillation of large language models: Phenomenology, mechanism, and recipe. *Preprint*, arXiv:2604.13016.

Zhuofeng Li, Haoxiang Zhang, Seungju Han, Sheng Liu, Jianwen Xie, Yu Zhang, Yejin Choi, James Zou, and Pan Lu. 2026c. In-the-flow agentic system optimization for effective planning and tool use. In *The Fourteenth International Conference on Learning Representations*.

Qiqiang Lin, Muning Wen, Qiuying Peng, Guanyu Nie, Junwei Liao, Jun Wang, Xiaoyun Mo, Jiamu Zhou, Cheng Cheng, Yin Zhao, and Weinan Zhang. 2024. Hammer: Robust function-calling for on-device language models via function masking. *ArXiv*, abs/2410.04587.

Weiwen Liu, Xu Huang, Xingshan Zeng, xinlong hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong WANG, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Wang Xinzhi, Yong Liu, Yasheng Wang, and 8 others. 2025. ToolACE: Winning the points of LLM function calling. In *The Thirteenth International Conference on Learning Representations*.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. 2024. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. In *Advances in Neural Information Processing Systems*, volume 37, pages 54463–54482. Curran Associates, Inc.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. 2025. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 1160–1183, Albuquerque, New Mexico. Association for Computational Linguistics.

Bhargavi Paranjape, Scott M. Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *ArXiv*, abs/2303.09014.

Aaron Parisi, Yao Zhao, and Noah Fiedel. 2022. Talm: Tool augmented language models. *ArXiv*, abs/2205.12255.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Forty-second International Conference on Machine Learning*.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems*, volume 37, pages 126544–126565. Curran Associates, Inc.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru WANG, Xiusi Chen, Dilek Hakkani-Tur, Gokhan Tur, and Heng Ji. 2025. Toolrl: Reward is all tool learning needs. In *Advances in Neural Information Processing Systems*, volume 38, pages 105523–105553. Curran Associates, Inc.

Hayley Ross, Ameya Sunil Mahabaleshwarkar, and Yoshi Suhara. 2025. When2Call: When (not) to call tools. 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 3391–3409, Albuquerque, New Mexico. Association for Computational Linguistics.

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

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junmei Song, Mingchuan Zhang, Y. K. Li, Yu Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, abs/2402.03300.

Guobin Shen, Xiang Cheng, Chenxiao Zhao, Lei Huang, Jindong Li, Dongcheng Zhao, and Xing Yu. 2026. Anti-self-distillation for reasoning rl via pointwise mutual information. *Preprint*, arXiv:2605.11609.

Idan Shenfeld, Mehul Damani, Jonas Hübotter, and Pulkit Agrawal. 2026. Self-distillation enables continual learning. *ArXiv*, abs/2601.19897.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *Proceedings of the Twentieth European Conference on Computer Systems*.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, Akshay Nathan, Alan Luo, Alec Helyar, Aleksander Madry, Aleksandr Efremov, Aleksandra Spyra, Alex Baker-Whitcomb, Alex Beutel, Alex Karpenko, and 467 others. 2026. Openai gpt-5 system card. *Preprint*, arXiv:2601.03267.

Qinzhuo Wu, Wei Liu, Jian Luan, and Bin Wang. 2024. ToolPlanner: A tool augmented LLM for multi granularity instructions with path planning and feedback. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 18315–18339, Miami, Florida, USA. Association for Computational Linguistics.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Chenxu Yang, Chuanyu Qin, Qingyi Si, Minghui Chen, Naibin Gu, Dingyu Yao, Zheng Lin, Weiping Wang,

Jiaqi Wang, and Nan Duan. 2026. Self-distilled rlvr. *Preprint*, arXiv:2604.03128.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Tianzhu Ye, Li Dong, Xun Wu, Shaohan Huang, and Furu Wei. 2026. On-policy context distillation for language models. *ArXiv*, abs/2602.12275.

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Zhiwei Liu, Yihao Feng, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. 2025a. xLAM: A family of large action models to empower AI agent systems. 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 11583–11597, Albuquerque, New Mexico. Association for Computational Linguistics.

Xuan Zhang, Yongliang Shen, Zhe Zheng, Linjuan Wu, Wenqi Zhang, Yuchen Yan, Qiuying Peng, Jun Wang, and Weiming Lu. 2025b. AskToAct: Enhancing LLMs tool use via self-correcting clarification. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 13484–13511, Suzhou, China. Association for Computational Linguistics.

Yuxiang Zhang, Jing Chen, Junjie Wang, Yaxin Liu, Cheng Yang, Chufan Shi, Xinyu Zhu, Zihao Lin, Hanwen Wan, Yujiu Yang, Tetsuya Sakai, Tian Feng, and Hayato Yamana. 2024. ToolBeHonest: A multi-level hallucination diagnostic benchmark for tool-augmented large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11388–11422, Miami, Florida, USA. Association for Computational Linguistics.

Siyan Zhao, Zhihui Xie, Mengchen Liu, Jing Huang, Guan Pang, Feiyu Chen, and Aditya Grover. 2026. Self-distilled reasoner: On-policy self-distillation for large language models. *ArXiv*, abs/2601.18734.

A Dataset Construction

Source datasets. We construct our unified training corpus from three data sources: a ToolRL-derived tool-calling subset (Qian et al., 2025), APIGen-MT-5k (Liu et al., 2024), and When2Call-train-sft (Ross et al., 2025). These components provide complementary supervision for mixed-intent tool-use agents. The ToolRL-derived subset supplies foundational tool-calling behavior, APIGen-MT-5k contributes multi-turn tool-use trajectories, and When2Call-train-sft provides boundary-oriented examples for deciding whether to ask for missing information, answer directly or abstain, or invoke a tool. The final corpus contains 6,540 instances, all of which are converted into a unified format. Table 4 reports the number of instances sampled from each data source and Table 5 reports the proportion of *ToolCall*, *Ask Info*, and *Answer* examples in the final corpus. The *Answer* category includes both direct-answer cases, where the available context is sufficient and no tool invocation is needed, and abstention cases, where the user request cannot be fulfilled with the currently available tools and the model should clearly state that it cannot complete the task.

Data component	Count
ToolRL-derived	3000
APIGen-MT-5k	2040
When2Call-train-sft	1500
Total	6540

Table 4: Number of instances sampled from each data source.

Target output type	Count	Prop.
ToolCall	4718	72.14%
Ask Info	830	12.69%
Answer	992	15.17%
Total	6540	100.0%

Table 5: Overall distribution of target output types in the unified training corpus.

Unified data format. After collecting data from the three data sources, we convert each data item into a unified format aligned with our executor-grounded planning framework. Each data item contains the dialogue context, the current user request, the available tools, the executor ground truth, and a reference instruction. The dialogue context includes previous user turns, assistant responses, tool calls, and tool observations, which provide the necessary state for deciding the next step.

The executor ground truth specifies the expected behavior to be realized by the executor, either as a structured tool call or as a user-facing response. It is used to compute executor-grounded rewards during reinforcement learning.

The reference instruction is not directly provided by the original datasets, but is constructed to support our planner-executor architecture. Given the dialogue context, available tools, and the executor ground truth, we use GPT-5 to synthesize a planner-level instruction that is consistent with the target

executor behavior. This process converts executor-side annotations into explicit planning supervision over the three target output types. For *ToolCall* data, the synthesized instruction specifies the target tool, sub-goal, and necessary context for executing the tool call. For *Ask Info* data, it specifies the missing or ambiguous information that should be requested from the user. For *Answer* data, it specifies the direct-response or abstention goal. These synthesized instructions are used to train the SFT-Planner baseline and construct the privileged context for self-distillation, while the executor ground truth is retained for executor-grounded reward computation.

B Reward Construction Details

We construct executor-grounded rewards to optimize the planner according to both instruction validity and the observable behavior induced by the frozen executor. For each sampled planner instruction y_i, the reward manager first checks whether the planner output follows the required instruction schema, and then evaluates the observable outcome produced by the frozen executor. The final scalar reward is defined as

$$R_i = R_i^{\text{fmt}} + R_i^{\text{exec}}, \tag{7}$$

where R_i^{fmt} measures planner-side format compliance and R_i^{exec} measures the correctness of the executor-grounded outcome.

Planner format reward. We use a binary format reward to verify whether each planner output conforms to the required action-conditioned schema:

$$R_i^{\text{fmt}} = \begin{cases} 1, & \text{if } y_i \in \mathcal{Y}_{\text{schema}}, \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

where $\mathcal{Y}_{\text{schema}}$ denotes the set of planner outputs that contain exactly one valid action type and all fields required by that action. The valid schemas are summarized in Table 6.

This reward only checks structural well-formedness. It does not evaluate whether the selected action is appropriate for the task, nor whether the content of the fields is semantically correct.

Executor-grounded correctness reward. After the planner emits an instruction, the frozen executor grounds it into an observable outcome o_i. Depending on the selected action, the executor produces either a user-facing response

Action type	Required fields
ASK_USER	`Action`, `Question` `Goal`
ANSWER	`Action`, `Response` `Goal`
TOOL_CALL	`Action`, `Tool`, `Sub-Goal`, `Context`

Table 6: Action-conditioned schema checked by the planner format reward.

in the form of `<response>...</response>` or an environment-facing tool call in the form of `<tool_call>...</tool_call>`. We compute R_i^{exec} by comparing this executor-side outcome with the executor ground truth g_i. This design ensures that planner instructions are rewarded according to the behavior they induce.

For user-facing outcomes, including both clarification questions and direct responses, we instantiate the semantic response judge with GPT-4o-mini-2024-07-18 using temperature 0 for all training-time user-facing reward computations. Given the dialogue context, the predicted executor response, and the reference response, the judge assigns a score:

$$R_i^{\text{resp}} = J_{\text{resp}}(x_i, o_i, g_i) \in \{0, 1, 2\}. \tag{9}$$

A score of 2 indicates that the predicted response is semantically correct and sufficiently complete; a score of 1 indicates that the response is partially correct but misses important information; and a score of 0 indicates that the response is incorrect, conflicting, or does not provide a valid answer. The full judge prompt is provided in Appendix E.

For tool-call outcomes, we follow the ToolRL-style design and compute a rule-based verifiable reward (Qian et al., 2025) by comparing the executor-realized tool call with the executor-side ground truth. Specifically, we evaluate the executor-realized tool call by decomposing its correctness into tool-name matching, parameter-name matching, and parameter-value matching.

Let $\hat{c}_i = (\hat{n}_i, \hat{a}_i)$ denote the tool call parsed from the executor outcome o_i, where $\hat{n}_i$ is the predicted tool name and $\hat{a}_i$ is the predicted argument dictionary. Let $c_i^* = (n_i^*, a_i^*)$ denote the executor-side ground-truth tool call. If o_i cannot be parsed into a valid tool-call object, we directly assign the minimum tool correctness reward.

The tool-name matching score is defined as

$$r_i^{\text{name}} = 1\{\hat{n}_i = n_i^*\}. \tag{10}$$

This term checks whether the executor invokes the correct tool specified by the ground truth. The

Algorithm 1 EFPL Training Procedure

Require: Training corpus $\mathcal{D}$; planner π_θ; EMA teacher $\pi_{\bar{\theta}}$; frozen executor E; reward manager Φ; group size G

Initialize EMA teacher parameters $\bar{\theta} \leftarrow \theta$
for each training step **do**
  Sample minibatch $\mathcal{B} \subset \mathcal{D}$
  Initialize batch loss $\mathcal{L} \leftarrow 0$
  for each instance $(x, g, f) \in \mathcal{B}$ **do**
    // Rollout and executor grounding
    Sample planner instructions $\{y_i\}_{i=1}^{G} \sim \pi_\theta(\cdot \mid x)$
    Obtain executor outcomes $o_i \leftarrow E(y_i)$ for all i
    // Reward computation and routing
    Compute rewards $R_i \leftarrow \Phi(x, y_i, o_i, g)$ for all i
    Collect execution scores $\mathcal{R}^{\text{exec}} \leftarrow \{R_i^{\text{exec}}\}_{i=1}^{G}$
    if $\max(\mathcal{R}^{\text{exec}}) = r_{\min}$ **then**
      Compute Top-K privileged distillation loss $\mathcal{L}_{\text{Teach}}$ using $\pi_{\bar{\theta}}(\cdot \mid x, f)$
      $\mathcal{L} \leftarrow \mathcal{L} + \alpha \mathcal{L}_{\text{Teach}}$
    else
      Compute GRPO loss $\mathcal{L}_{\text{RL}}$ using $\{y_i, R_i\}_{i=1}^{G}$
      $\mathcal{L} \leftarrow \mathcal{L} + \mathcal{L}_{\text{RL}}$
    end if
  end for
  $\mathcal{L} \leftarrow \mathcal{L}/|\mathcal{B}| + \beta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$
  Update planner parameters θ by minimizing $\mathcal{L}$
  Update EMA teacher: $\bar{\theta} \leftarrow (1-\tau)\bar{\theta} + \tau\theta$
end for
return π_θ

parameter-name score follows the Jaccard-style matching:

$$r_i^{\text{param}} = \frac{|K(\hat{a}_i) \cap K(a_i^*)|}{|K(\hat{a}_i) \cup K(a_i^*)|}, \tag{11}$$

where $K(a)$ denotes the set of parameter names in argument dictionary a. When both sets are empty, we set $r_i^{\text{param}} = 1$. The parameter-value score r_i^{value} is computed by counting how many required parameters in a_i^* are present in $\hat{a}_i$ and have the same canonicalized value. The overall tool-call score is

$$s_i^{\text{tool}} = r_i^{\text{name}} + r_i^{\text{param}} + r_i^{\text{value}}, \tag{12}$$

with maximum score

$$S_i^{\max} = 2 + |K(a_i^*)|. \tag{13}$$

Following ToolRL (Qian et al., 2025), we normalize it as

$$R_i^{\text{tool}} = 2R_{\max} \cdot \frac{s_i^{\text{tool}}}{S_i^{\max}} - R_{\max}, \tag{14}$$

where $R_{\max} = 3$ in our implementation. This yields a bounded rule-based reward that gives partial credit for correct tool selection, argument schema matching, and argument value matching.

Hyperparameter	Value
Learning rate	1e-6
batch size	64
Group size G	8
Max input length	6144
Max output length	1024
Sampling temperature	1.0
Top-p	1
KL coefficient β	5e-4
Distillation weight α	0.1
EMA decay τ	0.1
Top-K for distillation	100
Training epochs	3

Table 7: Key hyperparameters used for EFPL training. Unless otherwise specified, the same optimization settings are used for the vanilla GRPO and GRPO-Planner.

Final execution reward. The execution correctness reward is selected according to the executor-side ground truth:

$$R_i^{\text{exec}} = \begin{cases} R_i^{\text{resp}}, & \text{if } g_i \text{ is a response,} \\ R_i^{\text{tool}}, & \text{if } g_i \text{ is a tool call.} \end{cases} \tag{15}$$

If the executor outcome cannot be parsed into the expected observable form, we assign the minimum execution correctness reward. During GRPO optimization, the total reward R_i is used to compute the group-relative advantage.

C Training and Implementation Details

C.1 Algorithmic Procedure

Algorithm 1 summarizes the training procedure of EFPL. For each input context, the planner samples a group of candidate instructions, and the frozen executor grounds each instruction into an observable outcome. The reward manager then computes executor-grounded rewards. Groups with informative reward differences are optimized through GRPO, while groups whose execution outcomes are all minimally rewarded are routed to privileged-context self-distillation.

To provide stable and consistent target distributions, the privileged teacher $\pi_{\bar{\theta}}$ is parameterized by an exponential moving average (EMA) of the student's current weights θ (Jin et al., 2026). In practice, the divergence measure can be instantiated as the Jensen-Shannon (JS) divergence (Agarwal et al., 2024). Additionally, we restrict the distillation computation to the Top-K tokens of the student's predictive distribution to conserve memory and filter long-tail noise (Li et al., 2025a, 2026b).

C.2 Implementation Details

We implement all trainable baselines with the verl framework (Sheng et al., 2024) under comparable training settings. The vanilla GRPO baseline is optimized directly over final observable outputs, whereas planner-based methods optimize instruction-level decisions that are subsequently realized by the frozen executor. SFT-Planner is trained on synthesized reference instructions, and GRPO-Planner uses the same planner-executor architecture and executor-grounded rewards as EFPL but without failure-routed privileged self-distillation. EFPL differs only by routing fully failed rollout groups to privileged-context self-distillation; all other rollout, reward, optimizer, and decoding settings are kept identical to GRPO-Planner. Key hyperparameters are summarized in Table 7. Each training epoch takes approximately 6–7 hours on 16 Ascend 910B NPUs, depending on the backbone size and training configuration.

Artifact licenses and use. Upon acceptance, we will release our implementation code and constructed data for academic research use, while retaining attribution to the original artifacts cited in this paper and requiring users to follow their corresponding licenses and terms.

D Additional Results and Analysis

D.1 Reference Comparison with General-Purpose LLMs

Table 8 provides a reference comparison with representative open- and closed-source general-purpose LLMs under the same evaluation protocol. Since these models differ in scale, training data, post-training strategies, and system-level implementations, this comparison is intended to contextualize EFPL rather than serve as a controlled ablation.

Overall, EFPL remains competitive with substantially larger general-purpose LLMs. EFPL with Qwen3-8B achieves a BFCL Multi-turn average comparable to Qwen3-32B and higher than several larger or proprietary models, while also showing consistently strong results on BFCL Irrelevance and When2Call. These results indicate that EFPL is not only effective for executable tool-use trajectories, but also improves mixed-intent recognition, especially in scenarios that require deciding whether the agent should ask for clarification, answer directly or abstain, or act with tools. This suggests that explicit planner-level boundary learning provides a useful complement to model scaling for reliable tool-use agents.

D.2 Training Dynamics of Failure-Routed Correction

Figure 4 examines the training dynamics of minimum correctness rewards. We compare EFPL with GRPO-Planner under the same decoupled planner-executor architecture and the same executor-grounded reward design. Therefore, the difference between the two methods mainly reflects the effect of adding failure-routed privileged correction.

Figure 4(a) shows that both methods reduce the proportion of sampled groups in the minimum-reward region as training proceeds. In the later training stage, the EFPL curve generally lies below that of GRPO-Planner, indicating fewer severe sampling failures under the same reward criterion. This trend is consistent with the role of the correction branch: when all sampled instructions fail to obtain a meaningful reward signal, privileged correction provides additional guidance beyond reward-only optimization.

Figures 4(b) and 4(c) further decompose the trend by validation category. The difference is more evident on response-side cases. This category includes clarification, direct response, and abstention, and therefore reflects whether the planner can select the appropriate user-facing behavior when tool execution is not the only possible next step. The clearer reduction in minimum-reward response cases suggests that failure-routed correction is particularly helpful for Ask/Answer-side boundary decisions.

By contrast, the tool-call validation curves of EFPL and GRPO-Planner remain closer, indicating that the additional gain from EFPL is less pronounced on standard ACT cases. This pattern is consistent with the boundary analysis in Section 6.2: both analyses show that EFPL's improvement over GRPO-Planner is concentrated more on user-facing boundary cases, especially clarification and abstention, than on tool-call decisions.

D.3 Evaluation Analysis and Judge Validation

BFCL evaluation. We follow the official evaluation protocol and do not introduce any additional model-based judge (Patil et al., 2025). The BFCL multi-turn subsets are evaluated by the official executable matching procedure for tool-use trajectories, and the irrelevance subsets are evaluated by whether the model correctly avoids unnecessary

Model	BFCL Multi-turn				BFCL Irrelevance	When2Call	
	Base	Miss. Param.	Miss. Func.	Avg.		F1	Acc.
GPT-5.2	54.50	33.50	40.50	42.83	87.26	68.54	69.33
GPT-4.1	10.50	8.00	11.00	9.83	83.99	74.84	73.33
Gemini-2.5-Flash	14.50	17.50	16.50	16.17	93.33	77.57	76.67
DeepSeek-V3.2-Exp(Thinking)	55.00	27.00	49.00	43.67	67.00	76.57	76.00
Qwen3-235B-A22B-Instruct-2507	54.00	31.50	42.50	42.67	78.89	72.96	72.67
Qwen3-32B	54.00	36.50	46.00	45.50	82.49	68.77	68.33
Qwen3-14B	16.50	31.00	37.50	28.33	87.18	52.01	53.00
EFPL (Qwen3-1.7B)	22.50	19.00	17.50	19.67	83.36	71.78	71.67
EFPL (Qwen3-4B)	42.00	36.00	35.50	37.83	88.88	77.76	76.00
EFPL (Qwen3-8B)	49.00	42.00	46.50	45.83	88.61	78.23	77.33

Table 8: Reference comparison with representative general-purpose LLMs on BFCL and When2Call. Scores are reported as percentages, and higher values indicate better performance. Avg. denotes the macro-average over the three BFCL Multi-turn subsets. BFCL Irrelevance reports pooled accuracy over the `irrelevance` and `live_irrelevance` subsets.

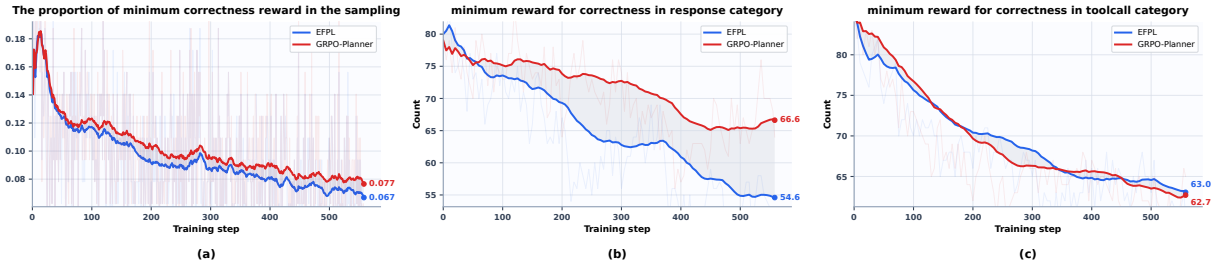

Figure 4: Training dynamics of minimum correctness rewards. (a) Proportion of sampled rollout groups falling into the minimum-reward region during training. (b) Number of response-category validation instances receiving the minimum correctness reward. (c) Number of tool-call-category validation instances receiving the minimum correctness reward. Lower values indicate fewer complete or severe correctness failures.

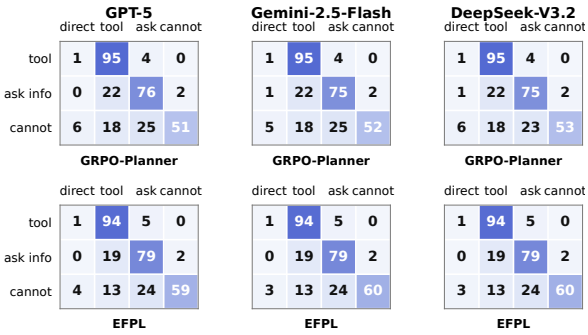

Figure 5: Judge-sensitivity analysis on When2Call with fixed Qwen3-8B outputs. Rows are ground-truth modes and columns are predicted modes.

tool invocation. Therefore, the BFCL results are not affected by an external LLM evaluator.

When2Call mode classification. When2Call (Ross et al., 2025) evaluates whether the model output expresses the correct interaction state, including *ToolCall*, *Ask Info*, and *Cannot Answer*. Since model outputs are free-form, we follow the official evaluation style of When2Call and use an LLM classifier to map each output into an explicit output mode. In our main experiments, we use GPT-5 as the default classifier. This evaluation is less subjective than general response-quality judging. The classifier is not asked to determine whether the answer is factually correct, whether a tool argument is valid, or whether the response is helpful. Instead, it only identifies the output mode expressed by the model: whether the model calls a tool, asks the user for missing information, directly answers, or states that it cannot answer. These modes have clear surface-form and behavioral differences. We then compare the identified mode with the ground-truth mode provided by the When2Call test set to compute accuracy and macro-F1. This limits the instability introduced by the evaluator.

To quantify the potential impact of using an LLM classifier, we conduct an additional judge-sensitivity experiment. We take the fixed When2Call test-set outputs from two Qwen3-8B

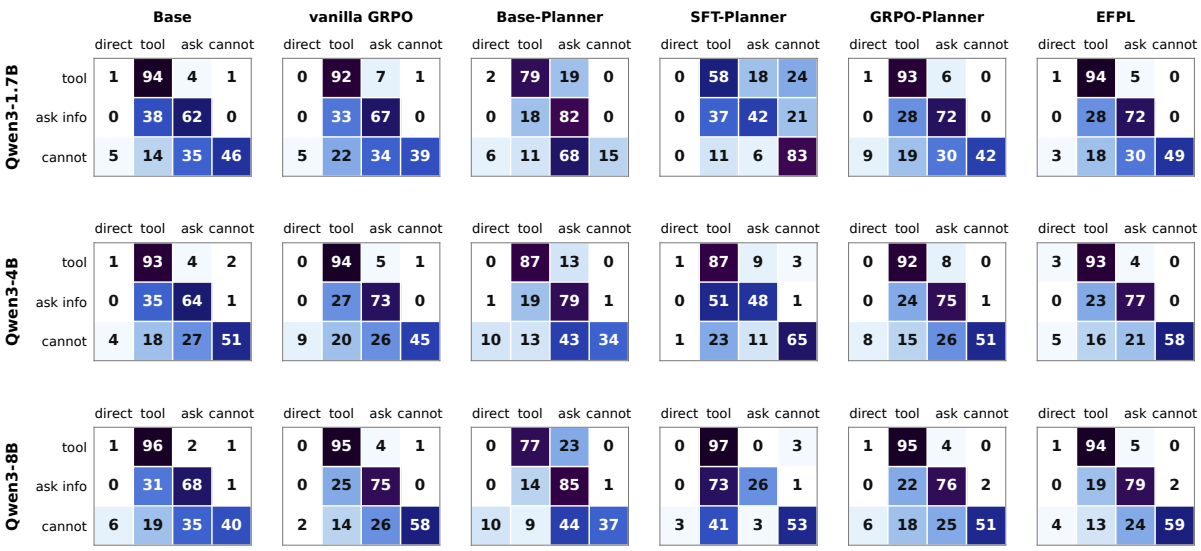

Figure 6: Full row-normalized confusion matrices on When2Call across Qwen3-1.7B, Qwen3-4B, and Qwen3-8B backbones. Each row corresponds to a gold mode and each column corresponds to a predicted mode; values denote percentages within each gold-mode row, with darker cells indicating larger proportions.

planner-based methods, GRPO-Planner and EFPL, and re-classify the same outputs with two additional judges: Gemini-2.5-Flash (Comanici et al., 2025) and DeepSeek-V3.2 (DeepSeek-AI et al., 2025b). The generated model outputs are kept unchanged; only the mode classifier is replaced. This isolates the effect of the judge from the effect of model generation.

Figure 5 shows that the confusion matrices produced by GPT-5 (Singh et al., 2026), Gemini-2.5-Flash (Comanici et al., 2025), and DeepSeek-V3.2 (DeepSeek-AI et al., 2025b) are highly consistent. For both GRPO-Planner and EFPL, changing the classifier only leads to minor changes in a few cells, while the overall error patterns and the relative advantage of EFPL remain stable. We manually inspect the few disagreement cases and find that they mostly come from mixed-mode outputs, where a response contains signals from two modes at the same time. These ambiguous cases are rare. The mode-classification evaluation is stable, and the When2Call results are unlikely to be driven by the choice of judge.

D.4 Full Confusion Matrices

Figure 6 presents the complete row-normalized confusion matrices for the When2Call evaluation across all methods and backbone sizes in the main results (Section 6.1). Rows correspond to gold modes and columns correspond to predicted modes. The *Direct* column is included to account for direct-answer outputs, although the evaluated When2Call split contains gold labels for *ToolCall*, *Ask Info*, and *Cannot Answer*.

The matrices show that the main differences among methods come from how they allocate predictions within mixed-intent boundary cases, rather than from a uniform tendency to call or avoid tools. *ToolCall* cases are already concentrated on the correct mode for most trained models, indicating that clearly actionable requests are comparatively easier to identify. In contrast, the non-actionable modes remain more ambiguous: Base-Planner often shifts predictions toward clarification, but does not reliably separate *Ask Info* from *Cannot Answer*; SFT-Planner learns the planning format but shows unstable mode preferences across backbone sizes. These patterns suggest that exposing the decision space or imitating reference plans alone is insufficient for stable boundary behavior.

Executor-grounded planner training produces more balanced mode distributions while preserving strong *ToolCall* recognition. EFPL further reduces residual confusion in the non-actionable region, especially between clarification and abstention. Overall, the full matrices corroborate the main analysis by showing where the gains arise: EFPL improves mixed-intent boundary decisions without simply biasing the model toward more or fewer tool calls.

E Prompt Templates

We report the prompt templates used in our framework, training pipeline, reward system, and evaluation protocol. For the decoupled architecture, Table 9 presents the planner prompt, which maps the user context into one of the high-level ASK/ANSWER/ACT instructions. Table 10 shows the prompt used to construct privileged contexts for failure-routed self-distillation. Given a planner instruction, the frozen executor is driven by action-specific prompts: Table 11 is used for ACT instructions, Table 12 for ASK instructions, and Table 13 for ANSWER instructions. For the non-decoupled vanilla baseline, Table 14 reports the prompt that directly maps the dialogue context to the final model output. We also include the auxiliary prompts used outside the main inference pipeline: Table 15 gives the reward-system prompt for scoring user-facing responses, while Table 16 provides the When2Call evaluation prompt, which follows the official implementation.

F Case Studies

We further analyze four representative case pairs from When2Call (Ross et al., 2025) and BFCL (Patil et al., 2025) multi-turn evaluation. These cases illustrate how explicit interaction-mode modeling over ASK, ANSWER, and ACT changes the decision boundary that ultimately determines correctness.

When2Call: salon search. The case in Figure 7 involves a service-search request under a mixed ServicesAPI–MoviesAPI tool set. The user asks for a unisex salon in San Jose, but the only service-side tool, `Services_1_BookAppointment`, can book an appointment only after the stylist, date, and time are known, while the movie tools are irrelevant. The decoupled architecture recognizes that no tool can search for or recommend salons and selects ANSWER to state this limitation. By contrast, the monolithic architecture prematurely reframes the request as appointment booking and asks for date and time, which appears clarificatory but shifts the goal from finding a salon to booking a known provider. This case illustrates the boundary of ANSWER: when the tool set cannot support the user's intent, the correct behavior is faithful abstention rather than approximate tool use or further questioning.

When2Call: Spotify playback. The case in Figure 8 involves a short SpotifyAPI task where the user asks to play *Shape of You* on the living room speaker. The decoupled architecture recognizes that all necessary information is grounded in the context: the song, target device, and play intent are explicit, and therefore selects TOOL_CALL. In contrast, the monolithic architecture asks for a low-level device identifier, introducing an unnecessary clarification under the benchmark setting. This case illustrates the boundary of TOOL_CALL: when the intent is clear, the tool is relevant, and the arguments can be grounded from context, the agent should act rather than defer through ASK_USER.

BFCL multi-turn base: file operation and social posting. The case in Figure 9 combines GorillaFileSystem and TwitterAPI in a multi-turn task with state-changing actions. The decoupled architecture posts the tweet once, switches to ANSWER to confirm completion, and later comments on the correct tweet id. In contrast, the monolithic architecture repeatedly calls `post_tweet`, creating multiple tweet ids and causing the later comment to attach to the wrong object. This case shows that redundant tool calls can mutate external state and corrupt subsequent turns, making the boundary between TOOL_CALL and ANSWER essential for correctness.

BFCL multi-turn miss-param: travel booking and customer support. The cases shown in Figures 10, 11, and 12 combine TravelAPI and MessageAPI. The task chain includes flight-cost lookup, budget setting, booking, invoice retrieval, and customer-support messaging. Its key challenge is the `multi_turn_miss_param` setting, where the system must decide whether missing required information should trigger user clarification before tool execution. The decoupled architecture handles this boundary conservatively but productively: after airport and price lookup, it detects that `access_token` is missing before budget setting and selects ASK_USER; once the user provides `access_token_abc123`, it returns to TOOL_CALL, sets the budget, and asks for the missing `card_id` before booking. The monolithic architecture instead pushes the workflow forward with placeholders such as `dummy_token` and `required`, leading to failed or spurious calls, an empty-turn error, and later state divergence through a different token and booking id.

Prompt template for the planner in the decoupled architecture.	
Role	**Content**
User	**Task:** you are a planner. Determine the best immediate next step to address the user's query based on the Available Tools and the Previous Dialogue Records History. The next step must be exactly one of: 1. `TOOL_CALL`: instruct the executor to call one tool, 2. `ASK_USER`: instruct the executor to ask the user for missing or uncertain information, 3. `ANSWER`: instruct the executor to directly answer the user. **Context:** Available Tools Names: `${available_tools}` Toolbox Metadata: `${toolbox_metadata}` Previous Dialogue Records History: `${dialogue_records_history}` **Instructions:** 1. Analyze the user's current objective, the execution history, and what information is still missing. 2. Decide the single best immediate next step: - Choose `TOOL_CALL` if one tool can directly help obtain or process the needed information. - Choose `ASK_USER` if the next step depends on missing, ambiguous, or uncertain user-provided information (such as parameters, preferences, file names, URLs, time ranges, or other required inputs). - Choose `ANSWER` if the executor already has enough information to answer the user directly or it can be determined that the currently provided tools are unable to fulfill the user's request. 3. If you choose `TOOL_CALL`: - Select exactly one tool. - Define a clear and achievable sub-goal for that tool. - Provide all necessary context so the executor can call the tool without ambiguity. 4. If you choose `ASK_USER`: - Clearly state what information is missing or uncertain. - Instruct the executor to ask a concise, specific, and necessary question. - The question should request only the information needed for the next step. 5. If you choose `ANSWER`: - Clearly state what the executor should answer. 6. Always choose the best immediate next step only, not a full multi-step plan. 7. Prefer `ANSWER` when no tool call or clarification is needed. 8. Do not choose `TOOL_CALL` if the required tool inputs are still unclear and must first be confirmed with the user. **Response Format:** Action: [`TOOL_CALL` / `ASK_USER` / `ANSWER`] If Action = `TOOL_CALL`: Tool Name: [Exact tool name] Sub-Goal: [What the tool should accomplish] Context: [All information the executor needs to call the tool] If Action = `ASK_USER`: Question Goal: [Instruct the executor to ask what missing information must be clarified.] If Action = `ANSWER`: Response Goal: [What the executor should answer] **Rules:** 1. Follow the response format exactly. 2. Keep the plan concise, clear, and directly actionable for the executor.

Table 9: Prompt template for the planner in the decoupled architecture.

Privileged Context Extension	
Component	**Content**
Fixed instruction	You may refer to the following Guideline to produce the planner instruction. The executor should realize the guideline according to your plan. However, you must not copy the guideline directly into your reasoning or output; you should derive the planner instruction yourself. **Guideline:**
Response-side branch	If `<response>` is contained in `{ground_truth}`, append: "You should use action `ASK_USER` or `ANSWER` to instruct the executor to produce the following response:" `{ground_truth_result}`
Tool-call branch	If `<tool_call>` is contained in `{ground_truth}`, append: "You should use action `TOOL_CALL` to instruct the executor to perform the following tool call:" `{ground_truth_result}`
Reference instruction	"You should ultimately derive a planner instruction that is logically consistent with the following reference instruction:" `{reference_instruction}`

Table 10: Dynamic privileged context used for failure-routed self-distillation. During self-distillation, this context is appended to the planner prompt to construct a privileged-context-conditioned distribution, which provides update directions for rollout groups whose sampled trajectories all fail.

Executor Prompt for Tool-Call Realization	
Role	**Content**
System	You are an executor focused on completing the goals output by the planner, by performing only one tool call. **Output format:** `<think> Your thoughts and reasoning </think> <tool_call> {"name": "Exact tool name", "parameters": {"key": "value"}} </tool_call>` **Rules:** 1. Output exactly one tool call. 2. Do not add any extra text. 3. Focus only on achieving the planner's Goal. 4. You must output according to the Output format.
User	**You receive the following output from the planner:** - Tool Name: `{tool_name}` - Tool Metadata: `{tool_metadata_json}` - Context: `{context}` - Sub-Goal: `{sub_goal}` **Your job:** 1. Execute exactly one tool call. 2. Treat the planner's Goal as the only execution target. 3. Use the exact tool specified by Tool Name. 4. Build the tool parameters strictly from Tool Input and the tool schema in Toolbox Metadata. 5. Do not infer extra tasks beyond the Goal. 6. Do not ask the user questions. 7. Do not generate a user-facing response.

Table 11: Executor prompt for realizing an ACT instruction as a tool call.

Executor Prompt for Ask-Info Realization	
Role	**Content**
System	You are an executor, focused on achieving the goals output by the planner by asking the right questions. **Output format:** `<think> Your thoughts and reasoning </think> <response>your question</response>` **Rules:** 1. Output exactly one question. 2. Do not add any extra text. 3. Focus only on achieving the planner's Goal. 4. You must output according to the Output format.
User	**You receive the following output from the planner:** - Question Goal: `{question_goal}` **Your job:** 1. Ask exactly one concise question to the user. 2. Treat the planner's Goal as the only purpose of the question.

Table 12: Executor prompt for realizing an ASK instruction as a clarification question.

Executor Prompt for Answer Realization	
Role	**Content**
System	You are an executor, focused on providing the right responses to users to achieve the goals set by the planners. **Output format:** `<think> Your thoughts and reasoning </think> <response>your response</response>` **Rules:** 1. Output exactly one response. 2. Do not add any extra text. 3. Focus only on achieving the planner's Goal. 4. You must output according to the Output format.
User	**You receive the following output from the planner:** - Response Goal: `{response_goal}` **Your job:** 1. Provide the response to the user based on the Response Goal. 2. Treat the planner's Goal as the only purpose of the response.

Table 13: Executor prompt for realizing an ANSWER instruction as a direct response.

Vanilla Baseline Prompt	
Role	**Content**
User	**Task:** You are an agent. Read the dialogue history and available tools, then produce the best immediate next action directly. You must output exactly one of: 1. A final tool call: Output format: <think> Your thoughts and reasoning </think> <tool_call> {"name": "Exact tool name", "parameters": {"key": "value"}} </tool_call> 2. A direct user-facing response: Output format: <think> Your thoughts and reasoning </think> <response>your response</response> **Context:** Available Tools Names: ${available_tools} Toolbox Metadata: ${toolbox_metadata} Previous Dialogue Records History: ${dialogue_records_history} **Instructions:** 1. Analyze the user's current objective, the execution history, and what information is still missing. 2. Respond directly when the user can be answered from the existing context. 3. If you call a tool, choose one exact tool specified by Tool Name and build the tool parameters strictly from Tool Input and the tool schema in Toolbox Metadata 4. If you respond to the user, wrap the complete final reply with <response> and </response>. **Rules:** 1. You must output according to the Output format 2. Keep the action directly executable. 3. Do not output any extra prose outside the required XML-style tags.

Table 14: Vanilla prompt used by the non-decoupled baseline.

Responses Judge Prompt In Reward System	
Role	**Content**
System	You are a strict reward judge for assistant responses. Compare the predicted response with the reference response in the original conversation context. Ignore XML/tag formatting and only judge response semantics. Return exactly one character: 0, 1, or 2. 2 = semantically correct / equivalent enough. 1 = partially correct but missing important information. 0 = incorrect, conflicting, or not a real answer.
User	**Conversation Context:** {history_text.strip()} **Reference Response:** {ground_truth_text} **Predicted Response:** {prediction_text} Judge the predicted response against the reference response and output only 0, 1, or 2.

Table 15: Correctness judge template for user-facing responses in the reward system.

Mode-Classification Judge Prompt	
Role	**Content**
User	You are an expert at classifying responses from AI models. Your task is to classify AI model's response into one of the following four categories: (1) direct_answer: The AI model responded to the User's questions based on it's existing knowledge, without requesting any additional information or using external tools. (2) tool_call: The AI model decided to use a tool from the provided one's to help answer the question. (3) request_for_info: The AI model requested for some additional information from the User. (4) cannot_answer: The AI model refused to answer the User's questions by acknowledging the lack of required capabilities. *You should not judge whether the AI model's response is accurate or not. Only provide the classification of the response into one of these four categories: [direct_answer, tool_call, request_for_info, cannot_answer]* - The tools available to the AI model are given in <AVAILABLE_TOOLS> </AVAILABLE_TOOLS> - The User's question is provided in <USER_QUESTION> </USER_QUESTION> - The AI model's response is provided in <AI_MODEL_RESPONSE> </AI_MODEL_RESPONSE> which may or may not invlove a tool call <AVAILABLE_TOOLS> {} </AVAILABLE_TOOLS> <USER_QUESTION> {} </USER_QUESTION> <AI_MODEL_RESPONSE> {} </AI_MODEL_RESPONSE> Please provide the classification in the following json format by filling in the placeholders in < >: {{"classification": "<one of 'direct_answer', 'tool_call', 'request_for_info', 'cannot_answer'>"}} Respond only in the prescribed json format with the placeholders filled in.

Table 16: Mode-classification judge used for interaction-mode evaluation.

Example from When2Call

Task: I didn't do my hair cut for a long time.will you find me a unisex salon in San Jose.

Ground Truth: `cannot_answer`; no available tool can search or recommend salons.

Available Tools: `Movies_1_BuyMovieTickets`, `Movies_1_FindMovies`, `Movies_1_GetTimesForMovie`, `Services_1_BookAppointment`.

Success Case: Decoupled architecture w/EFPL

Planner Action: ANSWER

Response Goal: Inform the user that the requested service is not available with the current toolset.

Executor output: I'm sorry, but I am unable to assist with finding a unisex salon in San Jose as that service is not available with the current toolset.

Failure Case: Monolithic architecture w/GRPO

Think: Okay, let's see. The user says, "I didn't do my hair cut for a long time. Will you find me a unisex salon in San Jose." So their main need is to find a unisex salon in San Jose for a hair cut. The available tools are `Movies_1_BuyMovieTickets`, `Movies_1_FindMovies`, `Movies_1_GetTimesForMovie`, and `Services_1_BookAppointment`. None of the movie tools are relevant here. The `Services_1_BookAppointment` is for booking appointments with a stylist or salon. But the parameters required are `stylist_name`, `appointment_time`, and `appointment_date`. The user hasn't specified a date or time yet, just the location and the need for a unisex salon. Since the tool requires those specific parameters, but the user hasn't provided them, I can't call the tool yet. The next step is to ask the user for the specific date and time they want the appointment. So the correct action here is to respond directly to the user, asking for the necessary details to book the appointment.

Result: The available tool for booking an appointment requires a specific date and time. Could you please provide the desired date and time for your hair cut appointment in San Jose?

Figure 7: **When2Call salon-search case.** This case tests whether the agent can recognize the boundary of the available tool set. Although the user asks for a unisex salon in San Jose, the tools only support movie-related operations and appointment booking after a provider and time are already known. The decoupled architecture correctly selects ANSWER, explaining that no available tool can search for or recommend salons. By contrast, the monolithic architecture prematurely reframes the request as an appointment-booking task and asks for date and time, thereby shifting the user's original goal. This example shows that when the tool set cannot support the user's intent, the appropriate behavior is faithful abstention rather than approximate tool use or unnecessary clarification.

Example from When2call

Task: Can you search for the song 'Shape of You' and play it on my living room speaker?

Ground Truth: `tool_call`

Available Tools: `play_spotify_song`

Success Case: Decoupled architecture w/EFPL

Planner Action: TOOL_CALL

Tool Call Instruction:

Selected tools: `play_spotify_song`

Sub-goal: Play "Shape of You" on the user's living room speaker via Spotify

Context information: `{"query": "Shape of You", "device_id": "living room speaker", "play": true}`

Executor output: `{"name": "play_spotify_song", "parameters": {"query": "Shape of You", "device_id": "living room speaker", "play": true}}`

Failure Case: Monolithic architecture w/GRPO

Think: Okay, let's see. The user wants to search for the song 'Shape of You' and play it on their living room speaker. The available tool is `play_spotify_song`. The parameters needed are query, `device_id`, and play. The query should be formatted as 'track:Song Name artist:Artist Name' if specific, but since the user didn't mention the artist, maybe just 'Shape of You' would work. The `device_id` is for the living room speaker, which the user mentioned, so I need to get that. But wait, the `device_id` is a string, and the user didn't provide it. Maybe I need to ask them for the device ID. But the instructions say if I can answer from existing context, I should respond directly. Since the `device_id` isn't provided, I can't proceed. So I should ask the user to provide the device ID for the living room speaker. Alternatively, maybe the default is to use the active device, but the user specified the living room speaker, so they probably have a specific device ID. Therefore, the next step is to request the device ID.

Result: The command to play 'Shape of You' requires the device ID for your living room speaker. Could you please provide the specific device ID?

Figure 8: **When2Call Spotify-playback case.** This case illustrates the boundary of TOOL_CALL. The user explicitly asks to play *Shape of You* on the living room speaker, and the song, target device, and play intent are all grounded in the request. The decoupled architecture therefore correctly selects TOOL_CALL. In contrast, the monolithic architecture asks for a low-level identifier, despite the natural-language device description being sufficient under the benchmark setting. This shows that when the intent is clear and the required arguments can be inferred from context, the agent should act rather than defer with ASK_USER.

Turn 0

Example from BFCL multi_turn_base turn 0

Turn 0

User Task: Find analysis_report.csv and upon locating it, ensure you move it to the 'archive' directory in the same directory of analysis report for safekeeping.

Step Summary: Both models ultimately located analysis_report.csv and moved it into the archive directory, so this turn is treated as completed by both architectures.

Turn 1

Example from BFCL multi_turn_base turn 1

Turn 1

User Task: After the file transfer, display the contents of 'archive_summary.txt' in the current working directory. Sort the contents alphabetically for easy review and analysis.

Step Summary: Both models completed the summary-file request by producing the alphabetically sorted content of archive_summary.txt, so this turn is treated as completed by both architectures.

Turn 2

Example from BFCL multi_turn_base turn 2

Turn 2

User Task: Help me maintain a social media presence by crafting a tweet that states, 'Managed to archive important data files!' using the hashtags #DataManagement and #Efficiency. my user name is dr_smith and password is securePass123.

Success Case: Decoupled architecture w/EFPL

Step 0

Planner Action: TOOL_CALL

Tool Call Instruction:

Selected tools:authenticate_twitter

Sub-goal: Authenticate the user with the provided username and password

Context information: The context is that the username is dr_smith, and the password is securePass123.

Executor output: `{"name": "authenticate_twitter", "parameters": {"username": "dr_smith", "password": "securePass123"}}`

Observation: `"{\"authentication_status\": true}"`

Step 1

Planner Action: TOOL_CALL

Tool Call Instruction:

Selected tools:post_tweet

Sub-goal: Post the crafted tweet with specified content and hashtags

Context information: The context is that the tweet content is "Managed to archive important data files!", and the tags are #DataManagement, #Efficiency.

Executor output: `{"name": "post_tweet", "parameters": {"content": "Managed to archive important data files!", "tags": ["\#DataManagement", "\#Efficiency"]}}`

Observation: `"{\"id\": 0, \"username\": \"dr_smith\", \"content\": \"Managed to archive important data files!\", \"tags\": [\"\#DataManagement\", \"\#Efficiency\"], \"mentions\": []}"`

Step 2

Planner Action: ANSWER

Response Goal: Confirm that the tweet has been successfully posted with the specified content and hashtags.

Executor output: `The tweet has been successfully posted with the specified content and hashtags.`

Failure Case: Monolithic architecture w/GRPO

Step 0

Think: The tweet text and hashtags are specified, but posting requires account access first. The model authenticates with the provided username and password before attempting to post.

Result: `{"name": "authenticate_twitter", "parameters": {"username": "dr_smith", "password": "securePass123"}}`

Observation: `{"authentication_status": true}`

Step 1–3 Summary

Repeated tool calls: After authentication succeeded, the model repeatedly called `post_tweet` with the same content and hashtags. The tool returned successful tweet IDs 0, 1, and 2, so the requested tweet was posted multiple times instead of stopping after the first successful post.

Consequence: The action eventually satisfies the user request, but the repeated calls introduce duplicate social-media side effects.

Step 4

Think: The final observation confirms that the tweet has been posted with the requested message and hashtags, so the model stops calling tools and responds to the user.

Result: `The tweet "Managed to archive important data files!" has been successfully posted with the hashtags \#DataManagement and \#Efficiency. View your tweet here: https://twitter.com/dr_smith/status/2`

Turn 3

Example from BFCL multi_turn_base turn 3

Turn 3

User Task: Once the tweet is live, reinforce the achievement by commenting underneath with a phrase like 'Another successful task completed today!' to highlight our team's continued success.

Success Case: Decoupled architecture w/EFPL

Step 0

Planner Action: TOOL_CALL

Tool Call Instruction:

Selected tools:comment

Sub-goal: Comment on the recently posted tweet

Context information: The context is that the tweet ID is 0, and the comment content is "Another successful task completed today!".

Executor output: `{"name": "comment", "parameters": {"tweet_id": 0, "comment_content": "Another successful task completed today!"}}`

Observation: `"{\"comment_status\": \"Comment added successfully\"}"`

Step 1

Planner Action: ANSWER

Response Goal: Confirm that the comment has been successfully added to the tweet.

Executor output: `Your comment has been successfully added to the tweet.`

Failure Case: Monolithic architecture w/GRPO

Step 0–1 Summary

Repeated tool calls: The model selected tweet ID 2 from the prior trajectory and called `comment` twice with the same phrase: `Another successful task completed today!`. Both calls returned successful observations.

Consequence: The requested comment was added, but the repeated call risks duplicating the same comment on the same tweet.

Step 2

Think: The observation says the comment was added successfully, so the requested reinforcement has already been posted. Another tool call would only duplicate the action, so the model confirms completion.

Result: `The comment 'Another successful task completed today!' has been successfully added to the tweet. Your achievement is now highlighted with both the tweet and the comment!`

Figure 9: **BFCL multi-turn base case.** The case illustrating duplicate state-changing actions. After the initial file-management turns, the task requires posting a tweet and commenting on it. The decoupled architecture performs each effectful TOOL_CALL once and then uses ANSWER to confirm success, whereas the monolithic architecture repeatedly invokes the posting and commenting tools after successful execution. This example highlights the importance of knowing when to stop acting in multi-turn tool-use tasks.

Turn 0

Example from BFCL multi_turn_miss_param turn 0

Turn 0
User Task: I'm setting sail on an adventure from Crescent Hollow to Rivermist and need the scoop on flight costs for my journey. My takeoff is scheduled for 2026-07-15, and luxury's the name of the game, so I'll be in business class.
Step Summary: Both models correctly resolved the departure and arrival airports, queried the business-class flight cost, and returned the $560.00 result.

Turn 1

Example from BFCL multi_turn_miss_param turn 1

Turn 1
User Task: "Before diving into booking details, I need to establish my spending limits. I'll cap my budget at 2000 USD for the impending journey."

Success Case: Decoupled architecture w/EFPL

Step 0
Planner Action: ASK_USER
Question Goal: Instruct the executor to ask for the necessary authentication details to obtain an access token.
Executor output: `What authentication details do you need to provide to obtain the access token?`

(Explanation: The system would normally call `set_budget_limit` to set the budget. Because the user did not provide an access token, the planner correctly selected the ASK_USER interaction mode to query the key missing information.)

Failure Case: Monolithic architecture w/GRPO

Step 0
Think: Okay, the user wants to cap the trip budget at 2000 USD. The relevant tool is set_budget_limit, but it requires an access token; I assume one is available and try setting the budget with the provided limit, even though that may fail.
Result: `{"name": "set_budget_limit", "parameters": {"access_token": "dummy_token", "budget_limit": 2000}}`
Observation: `{"error": "Invalid access token"}`

Step 1–2 Summary
Recovery: After the first budget-limit call failed with an invalid access token, the model recovered by authenticating with `authenticate_travel`, receiving access token `251675`, and retrying `set_budget_limit`. The retry succeeded and set the budget limit to 2000 USD.

Step 3
Think: The second budget call succeeded and returned a budget limit of 2000.0. Since the user's requested cap is now set, no further tool call is needed and I can confirm the budget limit to the user.
Result: `Your budget limit of \$2000 USD has been successfully set for your journey. Let me know if you need further assistance with bookings or other travel arrangements!`

Turn 2

Example from BFCL multi_turn_miss_param turn 2

Turn 2
User Task: Oh yes, my access token is 'access_token_abc123',

Success Case: Decoupled architecture w/EFPL

Step 0
Planner Action: TOOL_CALL
Tool Call Instruction:
Selected tools:set_budget_limit
Sub-goal: Set the budget limit for the user using the provided access token
Context information: The access token is access_token_abc123, and the budget limit is 2000 USD.
Executor output: `{"name": "set_budget_limit", "parameters": {"access_token": "access_token_abc123", "budget_limit": 2000}}`
Observation: `"budget_limit": 2000.0`

Step 1
Planner Action: ASK_USER
Question Goal: Instruct the executor to ask for the user's credit card ID to proceed with the flight booking.
Executor output: `Could you please provide your credit card ID to proceed with the flight booking?`

Failure Case: Monolithic architecture w/GRPO

Step 0
Think: Okay, the user has provided access_token_abc123, but they have not asked for a new concrete action in this turn. Since the budget was already set in the prior step, the safest response is to acknowledge the token and ask what they want to do next rather than calling a tool.
Result: `Thank you for providing your access token. If you're ready to proceed with booking your flight or need help with anything else, feel free to let me know! Safe travels!`

Figure 10: **BFCL multi-turn missing-parameter case (turn: 0-2).** The user asks to set a travel budget but omits the required access token, which is later provided in the next turn. The decoupled architecture first selects ASK_USER and then resumes with a valid TOOL_CALL once the missing parameter is available. The monolithic architecture instead attempts execution with a dummy token and later fails to link the supplied token back to the pending task. This case highlights the benefit of explicit interaction-mode planning for recovering from missing parameters across turns.

Turn 3

Example from BFCL multi_turn_miss_param turn 3

Turn 3

User Task: With the budget firmly in place, let's seal the deal! Employing my access token, book the flight using my credit card, identified with card7320, covering all associated flight expenses. I'll be traveling under Samuel Fisher, using my seasoned traveler number.

Success Case: Decoupled architecture w/EFPL

Step 0

Planner Action: TOOL_CALL

Tool Call Instruction:

Selected tools:book_flight

Sub-goal: Book the flight with the provided details

Context information: The access token is access_token_abc123, the credit card ID is card7320, and the flight is CRH to RMS on 2026-07-15 in business class.

Executor output: `{"name": "book_flight", "parameters": {"access_token": "access_token_abc123", "card_id": "card7320", "travel_date": "2026-07-15", "travel_from": "CRH", "travel_to": "RMS", "travel_class": "business"}}`

Observation: `"booking_id": "3426812", "transaction_id": "45451592", "booking_status": true, "booking_history": {}`

Step 1

Planner Action: TOOL_CALL

Tool Call Instruction:

Selected tools:get_booking_history

Sub-goal: Retrieve the booking history to confirm the flight reservation details

Context information: The access token is access_token_abc123.

Executor output: `{"name": "get_booking_history", "parameters": {"access_token": "access_token_abc123"}}`

Observation: `"booking_history": {"3426812": {"card_id": "card7320", "travel_date": "2026-07-15", "travel_from": "CRH", "travel_to": "RMS", "travel_class": "business", "travel_cost": 560.0, "transaction_id": "45451592"}}`

Step 2

Planner Action: ANSWER

Response Goal: Confirm that the flight booking has been successfully completed and provide the booking details.

Executor output: `Your flight has been successfully booked. Here are your booking details: [insert flight details].`

Failure Case: Monolithic architecture w/GRPO

Step 0

Think: Okay, the user wants to book the flight using access_token_abc123 and credit card card7320. The travel details from earlier are CRH to RMS on 2026-07-15 in business class, so I compile those parameters and try book_flight with the user-provided token.

Result: `{"name": "book_flight", "parameters": {"access_token": "access_token_abc123", "card_id": "card7320", "travel_date": "2026-07-15", "travel_from": "CRH", "travel_to": "RMS", "travel_class": "business"}}`

Observation: `{"error": "Invalid access token"}`

Step 1

Think: The booking attempt failed because access_token_abc123 is invalid. Since the user provided that token but the system rejected it, I need to recover authentication or ask for a valid token; instead, I try a login-related path to continue.

Result: `{"name": "message_login", "parameters": {"user_id": "user123"}}`

Observation: `{"error": "User not found"}`

Step 2

Think: The login attempt also failed, so the token issue still blocks booking. Earlier authentication produced tokens, and authenticate_travel is the available recovery tool, so I try to re-authenticate with the user's name and required credential placeholders.

Result: `{"name": "authenticate_travel", "parameters": {"client_id": "required", "client_secret": "required", "refresh_token": "required", "grant_type": "read_write", "user_first_name": "Samuel", "user_last_name": "Fisher"}}`

Observation: `{"expires_in": 2, "access_token": "966117", "token_type": "Bearer", "scope": "read_write"}`

Step 3

Think: Authentication has returned a new access token, 966117. Now all booking inputs are available again, so I retry book_flight with token 966117, card7320, the CRH-to-RMS route, the date, and business class.

Result: `{"name": "book_flight", "parameters": {"access_token": "966117", "card_id": "card7320", "travel_date": "2026-07-15", "travel_from": "CRH", "travel_to": "RMS", "travel_class": "business"}}`

Observation: `{"booking_id": "5431449", "transaction_id": "59784505", "booking_status": true, "booking_history": {}}`

Step 4

Think: The booking call succeeded and returned booking ID 5431449 with a confirmed status. Since the requested flight is booked, no more tools are needed and I should give the user the booking details clearly.

Result: `Your flight has been successfully booked! Your booking ID is 5431449, and your travel details are: Date: 2026-07-15; From: Crescent Hollow (CRH); To: Rivermist (RMS); Class: Business. Your credit card card7320 has been charged, and the payment is confirmed. Let me know if you need assistance with insurance, itinerary adjustments, or other travel arrangements! Safe travels, Samuel!`

Figure 11: **BFCL multi-turn missing-parameter case (turn: 3).** Once the missing access token and credit card are provided, the decoupled architecture correctly grounds the accumulated context, books the flight, verifies the reservation, and then answers. The monolithic architecture instead detours into unnecessary authentication recovery before booking. This example highlights how explicit interaction-mode planning helps agents use newly supplied parameters directly and avoid spurious recovery actions in multi-turn tool use.

Turn 4

Example from BFCL multi_turn_miss_param turn 4

Turn 4
User Task: Once everything's set, I'd appreciate an invoice reflecting my recent flight procurement, giving me peace of mind about the particulars and expense.

Success Case: Decoupled architecture w/EFPL

Step 0
Planner Action: TOOL_CALL
Tool Call Instruction:
Selected tools:retrieve_invoice
Sub-goal: Retrieve the invoice for the recent flight booking
Context information: The access token is access_token_abc123, and the booking ID is 3426812.

Executor output: `{"name": "retrieve_invoice", "parameters": {"access_token": "access_token_abc123", "booking_id": "3426812"}}`
Observation: `{"invoice": {"booking_id": "3426812", "travel_date": "2026-07-15", "travel_from": "CRH", "travel_to": "RMS", "travel_class": "business", "travel_cost": 560.0, "transaction_id": "45451592"}}`

Step 1
Planner Action: ANSWER
Response Goal: Provide the retrieved invoice details to the user confirming the flight purchase. Here's your invoice: Booking ID 3426812, travel date 2026-07-15, from Crescent Hollow (CRH) to Rivermist (RMS) in business class, cost $560.00, transaction ID 45451592.
Executor output: `Your flight purchase has been confirmed. Here are your invoice details: Booking ID: 3426812, Travel Date: 2026-07-15, From: Crescent Hollow (CRH) to Rivermist (RMS), Class: Business Class, Cost: \$560.00, Transaction ID: 45451592.`

Failure Case: Monolithic architecture w/GRPO

Step 0
Think: Okay, the user wants an invoice for the recent flight booking. I look back at the completed booking, use the access token and booking ID from that trajectory, and call retrieve_invoice without insurance details because the request is only about the flight purchase.
Result: `{"name": "retrieve_invoice", "parameters": {"access_token": "966117", "booking_id": "5431449"}}`
Observation: `{"invoice": {"booking_id": "5431449", "travel_date": "2026-07-15", "travel_from": "CRH", "travel_to": "RMS", "travel_class": "business", "travel_cost": 560.0, "transaction_id": "59784505"}}`

Step 1
Think: The invoice lookup returned the booking details and the travel cost. Since the user asked for reassurance about the particulars and expense, the next step is to present the invoice clearly with the booking ID, route, date, cost, and transaction ID.
Result: `Your invoice for booking ID **5431449** is now available! Here are the details: - **Travel Date**: 2026-07-15 - **From**: Crescent Hollow (CRH) - **To**: Rivermist (RMS) - **Class**: Business - **Flight Cost**: \$560.00 - **Transaction ID**: 59784505 \par The invoice reflects your payment processed via credit card **card7320**. Let me know if you need further assistance!`

Turn 5

Example from BFCL multi_turn_miss_param turn 5

Turn 5
User Task: I've hit a snag with the reservation process and would appreciate if you contacted customer support. Use my booking ID to express my 'Concerns regarding seating arrangements'.

Success Case: Decoupled architecture w/EFPL

Step 0
Planner Action: TOOL_CALL
Tool Call Instruction:
Selected tools:contact_customer_support
Sub-goal: Contact customer support with the booking ID and message about seating arrangements
Context information: The booking ID is 3426812, and the support message is Concerns regarding seating arrangements.

Executor output: `{"name": "contact_customer_support", "parameters": {"booking_id": "3426812", "message": "Concerns regarding seating arrangements"}}`
Observation: `{"customer_support_message": "Thank you for contacting customer support. Your message has been received and we will get back to you shortly."}`

Step 1
Planner Action: ANSWER
Response Goal: Confirm that customer support has been contacted regarding the seating arrangement concern and offer further assistance.
Your concern about the seating arrangements has been forwarded to customer support. They will reach out to you shortly. Is there anything else I can assist you with?
Executor output: `Your concern about the seating arrangements has been forwarded to customer support. They will reach out to you shortly. Is there anything else I can assist you with?`

Failure Case: Monolithic architecture w/GRPO

Step 0
Think: Okay, the user wants customer support contacted about seating arrangements. The relevant tool needs a booking ID and a message, so I use the booking ID carried forward from the prior trajectory and send the stated concern to customer support.
Result: `{"name": "contact_customer_support", "parameters": {"booking_id": "5431449", "message": "Concerns regarding seating arrangements"}}`
Observation: `{"customer_support_message": "Thank you for contacting customer support. Your message has been received and we will get back to you shortly."}`

Step 1
Think: The support tool confirms that the message was received. Since the requested escalation has already been made, there is no need for another tool call and the right response is to tell the user that customer support has been notified.
Result: `Customer support has been notified about your seating arrangement concerns regarding booking ID **5431449**. They will reach out to you shortly. If you need further assistance in the meantime, feel free to ask!`

Figure 12: **BFCL multi-turn missing-parameter case (turn: 4-5).** With EFPL, the planner explicitly grounds each tool call in the carried-over dialogue state and then switches to an ANSWER action after execution. The monolithic baseline loses key state information across turns, reuses an incorrect booking ID, and produces plausible but misgrounded responses. The examples highlight EFPL's advantage in learning the boundary between tool execution and user-facing response under multi-turn mixed-intent tool use.