CAR: Empowering Agents with Dynamic Tool Synthesis and Global Trajectory Rectification

Kai Wang, Xu Wang*, Zhiyuan Fu, Yudong Zhang, Yang Wang*
University of Science and Technology of China
{zaizwk, fuzhiyuan}@mail.ustc.edu.cn
{wx309, yudongzhang, angyang}@ustc.edu.cn

Abstract

While current LLM agents utilizing paradigms like ReAct or Plan-and-Solve have established a strong foundation for step-by-step reasoning, they remain brittle in open-ended environments due to two intrinsic limitations: (1) a closed action space: These frameworks are confined to static, pre-defined toolsets, rendering them unable to adapt when required tools are missing or obsolete. (2) myopic error recovery: Existing agents often get trapped in repetitive local retries, failing to diagnose and rectify root causes within the high-level plan. To overcome these limitations, we introduce **CAR** (**C**reate **A**nd **R**eplan), a novel architecture that incorporates a meta-tool synthesizer to dynamically augment the action space and a reflective replanning mechanism to revise global strategies. To rigorously evaluate our approach, we release **ToolHop-Pro**, a diagnostic benchmark with systematically pruned toolsets to simulate tool scarcity. Experiments demonstrate that CAR significantly outperforms representative baselines, validating its superior robustness where static agents fail. Code and data are available at https://github.com/Zaiz-77/car.

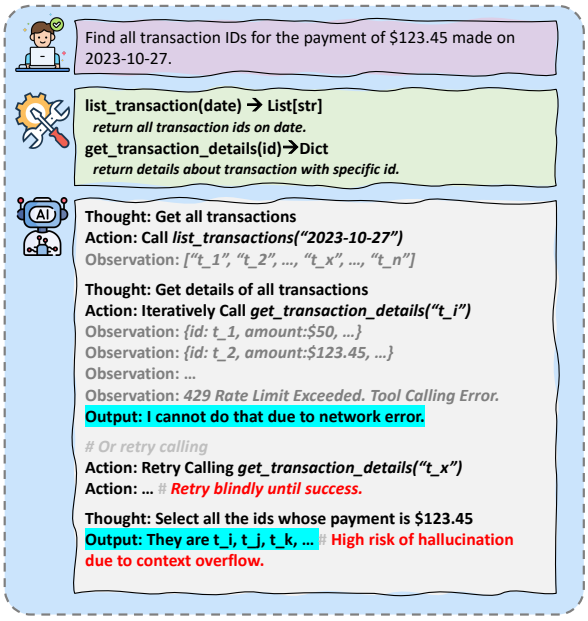

Figure 1: The dilemma of static agents. Confined to a fixed toolset, the agent is structurally forced into inefficient iterations that trigger API rate limits. Lacking high-level reflection, the agent often misinterprets these system constraints as transient errors, causing it to either abandon the task or retry blindly. The latter saturates the context window with verbose logs, leading to reasoning failures and hallucinations.

1 Introduction

The integration of Large Language Models (LLMs) with external tool interfaces has empowered agents to ground their abstract reasoning in executable actions (Wei et al., 2022; Mialon et al., 2023). Seminal frameworks like ReAct (Yao et al., 2023b) and Plan-and-Solve (Wang et al., 2023) have established the foundational protocol for this interaction, enabling agents to interleave chain-of-thought reasoning with atomic API invocations.

In practice, as illustrated in Figure 1, current agents operate under a strict closed world assumption: the action space (toolset) is fixed at initialization. When a high-level intent requires a tool absent from this fixed library, the agent is structurally forced into a brute-force strategy: decomposing the task into a massive sequence of atomic API calls. This approach inevitably triggers system constraints, such as API rate limits. Suffering from myopic error recovery, current agents often misinterpret these barriers as transient failures, locking themselves into a cycle of blind retries. Crucially, this repetitive process saturates the context window with verbose execution logs and error messages. Buried under this noise, the model struggles to attend to the relevant information—even if successfully retrieved—ultimately leading to "lost-in-the-middle" (Liu et al., 2024) hallucinations or

* Corresponding authors.

Findings of the Association for Computational Linguistics: ACL 2026, pages 17551–17571

reasoning failures.

While recent advances in search-based planning, such as LATS (Zhou et al., 2024), attempt to mitigate this by exploring complex reasoning trees, they fundamentally operate within the same static boundaries. We depart from this formulation by arguing that optimizing the search process is insufficient when the action space itself is deficient. True autonomy requires the agent to be not just a user of tools, but a creator of its own toolsets.

To this end, we introduce **CAR** (**C**reate **A**nd **R**eplan), a framework that reformulates agentic reasoning as a dynamic search process over an evolving action space. CAR operates on a simple principle: when a reasoning step fails due to tool limitations, CAR rejects the standard strategy of endless retries (e.g., parameter tuning). Instead, it triggers a reflective replanning process that revises both the failed and the future trajectory. Crucially, this replanning is empowered by a meta-tool synthesizer: rather than repeatedly forcing the agent to work with an inadequate toolset, the system synthesizes executable code for the required tool. This ensures the dual dynamism of both the action space and the high-level plan, significantly enhancing robustness against tool scarcity or execution failures.

Finally, to rigorously evaluate adaptability, we constructed **ToolHop-Pro**, an adversarial diagnostic suite derived from the ToolHop dataset (Ye et al., 2025). By systematically pruning tools and injecting execution noise, we simulate the scarcity and instability of real-world deployment, where agents encounter missing tools or execution errors.

Our contributions can be summarized as follows:

- We demonstrate that the static action space assumption is a primary bottleneck for the agent's robustness. Empirical evidence suggests that the ability to autonomously synthesize tools outweighs marginal gains in reasoning scaling for open-ended tasks.
- We propose **CAR**, a framework integrating just-in-time tool creation with hierarchical error handling. This design closes the loop between execution feedback and high-level planning, allowing the agent to adapt its capabilities to specific tasks.
- We release **ToolHop-Pro**, a benchmark designed to stress-test agents under conditions of tool scarcity and execution instability. Experiments indicate that CAR significantly outperforms representative baselines in these constrained environments, where static agents consistently fail.

2 Related Work

LLM-based agents. Recent advances have evolved LLMs from simple question-answering systems to autonomous problem solvers (Xi et al., 2025; Wang et al., 2024b). Foundational frameworks like CoT (Wei et al., 2022) and ReAct (Yao et al., 2023b) established the paradigm of interleaving reasoning with action, while subsequent works introduced memory mechanisms (Park et al., 2023) and multi-agent collaboration (Hong et al., 2024) for long-horizon tasks. However, these systems generally operate within static environments with fixed capabilities. In contrast, CAR targets open-ended scenarios, enabling the agent to adaptively expand its action space to meet emergent requirements.

Tool use and dynamic synthesis. Augmenting LLMs with external tools addresses limitations in knowledge and precision (Mialon et al., 2023; Qin et al., 2024). While methods like Toolformer (Schick et al., 2023) and Gorilla (Patil et al., 2024) excel at tool selection, they remain confined to static APIs. To address this rigidity, CREATOR (Qian et al., 2023) and LATM (Cai et al., 2024) pioneered tool creation via code synthesis, a paradigm extended by Voyager (Wang et al., 2024a). However, unlike these decoupled approaches, CAR embeds synthesis directly into the inference loop, dynamically generating just-in-time tools to bridge functional gaps during reasoning.

Self-correction. Robust agents must recover from execution failures. Reinforcement methods, such as Reflexion (Shinn et al., 2023) and Self-Refine (Madaan et al., 2023), iterate on linguistic feedback but often struggle with local optima. Conversely, search-based frameworks like Tree of Thoughts (Yao et al., 2023a) and RAP (Hao et al., 2023) ensure global consistency via systematic search (e.g., MCTS) yet incur prohibitive computational costs. CAR introduces Global Trajectory Rectification to bridge this gap. Unlike indiscriminate search, CAR triggers strategic replanning only upon critical execution failures, thereby combining the efficiency of linear reasoning with the dead-end avoidance capabilities of search algorithms.

3 Methodology

We propose **CAR**, a framework reformulating agentic reasoning as a dynamic search problem over an evolving action space. As shown in Figure 2, CAR advances beyond static chain-of-thought paradigms (Wei et al., 2022) by optimizing solution trajectories through dynamic tool expansion and policy rectification. Specifically, it employs code generation to synthesize new tools on demand, thereby bridging the functional gaps that static toolsets cannot address.

3.1 Preliminary

We define our framework via the tuple $\langle \mathcal{S}, \mathcal{A}_t, \pi, \mathcal{E}, \mathcal{V} \rangle$. The distinctive feature of this formulation is the dynamic nature of the action space $\mathcal{A}_t$. Rather than remaining static throughout the whole procedure, $\mathcal{A}_t$ evolves dynamically as the agent synthesizes new tools to bridge functional gaps.

The state space $\mathcal{S}$ encapsulates the agent's full context. A state $s_t \in \mathcal{S}$ consists of the initial query q, the current toolset, and the execution history $\mathcal{H}_t$. In practice, we found that treating history simply as a raw token stream often leads to context pollution, where the model becomes distracted by verbose error logs. Therefore, we formally define $\mathcal{H}_t$ as a structured sequence of intent-observation pairs, $\mathcal{H}_t = \{(z_0, o_0), \ldots, (z_{t-1}, o_{t-1})\}$, where each z_i represents an atomic planning step and o_i denotes its corresponding execution outcome. This structured representation functions as a working memory, mitigating the context pollution often seen with unstructured raw logs.

The policy π functions as the high-level planner. We depart from end-to-end paradigms that mix reasoning and execution. Instead, our policy π maps the current state to a logical plan $\mathcal{P}$, which is a sequence of abstract reasoning steps or sub-goals. This separation of concerns allows the planner to focus on strategic decomposition without being burdened by the syntax of specific API calls.

A central component of our model is the dynamic action space. At any time step t, the available action space is defined as the union of a predefined library $\mathcal{T}_{std}$ and a set of synthesized tools $\mathcal{T}_{gen}^{(t)}$ generated by the agent thus far. We also introduce a generative tool, the meta-tool τ_{meta}, which facilitates this expansion:

$$\mathcal{A}_t = (\mathcal{T}_{std} \cup \{\tau_{meta}\}) \cup \mathcal{T}_{gen}^{(t)} \quad (1)$$

This formulation is motivated by the observation that static action spaces inevitably lead to dead ends when a necessary tool is missing. By including τ_{meta}, we enable the action space to align with the underlying coding capabilities of LLMs.

Execution is delegated to a deterministic module $\mathcal{E}$, formally defined as the mapping $\mathcal{E} : \mathcal{Z} \times \mathcal{A}_t \to \mathcal{O}$. The executor takes an abstract step z and attempts to ground it into a specific function call within $\mathcal{A}_t$. This module abstracts away the complexities of runtime loading; if a step involves a newly synthesized tool, $\mathcal{E}$ handles the dynamic import transparently, returning an observation o that captures the result or error status.

Finally, to address the issue of "silent failures", where agents hallucinate success despite invalid outputs as in Figure 1, we incorporate a reflector $\mathcal{V}$ as a critic. This module evaluates the triplet $(\mathcal{H}_t, z, o)$ to produce a diagnosis D. Unlike simple scalar rewards, D provides structured feedback (e.g., classifying an error as a logic bug versus a syntax error), which is essential for guiding the rectification process described in subsequent sections.

3.2 Dynamic Action Space Expansion

The primary bottleneck we observed in current agents is the rigidity of the closed world assumption. In traditional practical setups, a developer must define a static JSON schema at initialization. If a task requires a tool outside this pre-defined set, the agent inevitably fails or hallucinates a non-existent function. We designed CAR specifically to break this dependency, treating the tool registry not as a static catalog, but as a dynamic set that grows with the problem complexity.

This expansion begins in the planning phase. When the planner π evaluates the current state s_t, it may identify sub-goals lacking a specific tool in $\mathcal{A}_t$. Rather than forcing a suboptimal selection, the planner outputs a step in the plan to synthesize a new tool, which triggers the Executor ($\mathcal{E}$) to invoke the meta-tool τ_{meta}. Leveraging the LLM's tool-calling mechanism to extract the arguments from context, this process instantiates the natural language specification d_{spec} required by the meta-tool to expand the toolset:

$$z_{meta} = \text{call}(\tau_{meta}, d_{spec}) \in \mathcal{P} \quad (2)$$

This design effectively decouples the strategic intent from the mechanics of invocation. The planner focuses solely on identifying the necessity of a new

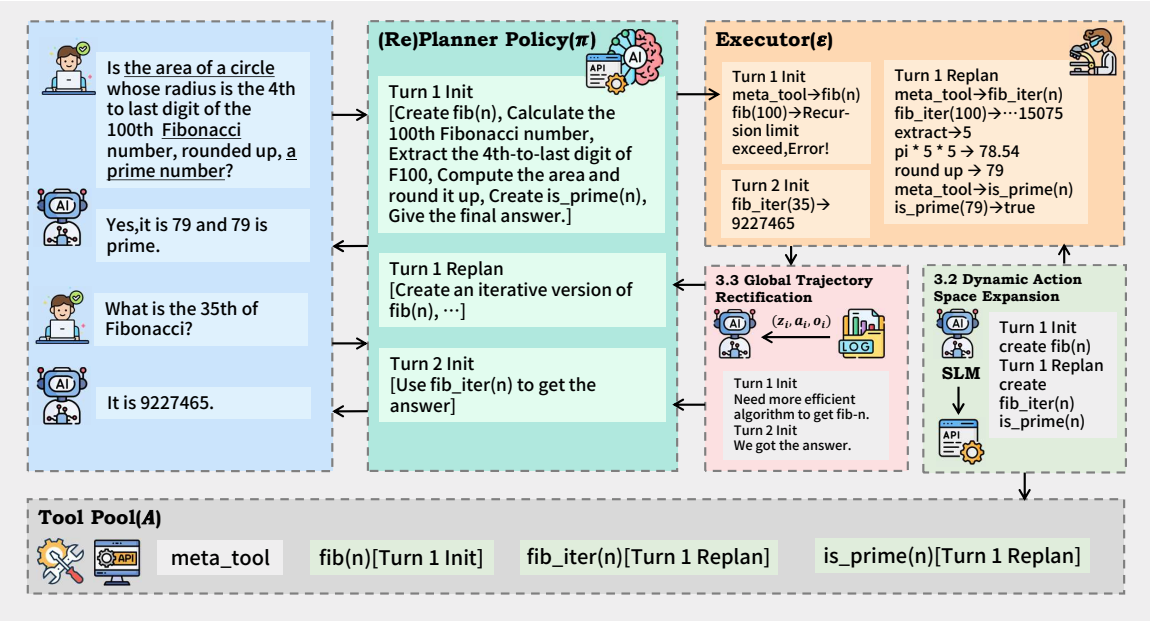

Figure 2: Overview of the CAR framework. The process integrates two core mechanisms: Dynamic Action Space Expansion, where the executor synthesizes missing tools to augment the Tool Pool ($\mathcal{A}$), and Global Trajectory Rectification, which diagnoses execution failures to trigger a strategic replan and tool refinement.

tool, while the LLM's inherent tool-calling mechanism handles the precise arguments (d_{spec}), preventing the high-level reasoning flow from being cluttered by implementation details. To mitigate tool pool bloat, the planner enforces a "Reuse First" policy: it is explicitly instructed to exhaustively evaluate existing tools before resorting to synthesis, promoting semantic deduplication at the source.

The actual synthesis is handled by the executor $\mathcal{E}$ with the meta-tool τ_{meta} (a specialized coding model). Upon encountering a meta-step, it uses τ_{meta} to materialize d_{spec} into a standalone Python script. Once the script is created, the system performs an immediate, formal update to the action space:

$$\mathcal{A}_{t+1} = \mathcal{A}_t \cup \{\tau_{new}\} \tag{3}$$

The critical engineering challenge here was usability: how to make τ_{new} available without restarting the system. Standard architectures rely on static imports (e.g., import tools), which locks the toolset at startup. We deviate from this convention by building our executor around Python's native dynamic loading protocols.

In practice, we treat the local file system as a hot-swappable interface. When an agent invokes a newly generated tool, the executor does not look up a pre-registered handle. Instead, it constructs a module specification directly from the script's file path and instantiates the module in memory at runtime. This allows the agent to "write" a plugin and "use" it in the very next step, creating a seamless feedback loop between reasoning and tooling.

Of course, executing LLM-generated code introduces security risks. To address this, we implement a containerized execution layer: the meta-tool and all synthesized functions execute within ephemeral Docker containers with restricted network access and resource quotas. We measured the overhead on Qwen3-Plus under the Complete setting: the sandboxed configuration incurs only ∼1.27s additional latency per query (46.32s vs. 45.05s) with negligible token difference, confirming that the isolation layer is transparent to the reasoning model and introduces no additional inference cost.

3.3 Global Trajectory Rectification

A persistent failure mode we observed in current agents is the retry trap. When an action fails, standard architectures often treat it as a transient error, blindly re-invoking the same tool with minor argument modification. This works for syntax errors, but leads to infinite loops when the error stems from a fundamental flaw in the plan itself, such as attempting to scrape a website that requires a login. CAR addresses this by enforcing a strict separation between execution-level corrections and strategic replanning.

At the lowest level, we empower the executor $\mathcal{E}$ to handle transient execution noise. We found that

large language models frequently commit minor formatting errors, such as malformed JSON or hallucinated arguments, which are easily correctable. Therefore, $\mathcal{E}$ runs a "Local Loop" with a budget of $k_{retry} = 3$. It parses error messages and applies heuristic adjustments to the tool call. This layer acts as a filter, resolving trivial implementation issues without disturbing the high-level reasoning process.

However, if the executor exhausts its retry budget, we treat the failure as a semantic deadlock rather than a syntactic one. At this stage, the reflector $\mathcal{V}$ performs a diagnostic analysis. It examines the execution trace to determine if the failure was caused by flawed tool logic or missing prerequisites. This diagnosis D serves as a hard constraint for the next phase.

Our approach to rectification departs from standard methods like Reflexion (Shinn et al., 2023), which often reset the environment or append a generic error prompt to the full context. Instead, we implement a partial reconstruction strategy. We posit that the steps preceding the failure are valid and should be preserved as a "foundation" $\mathcal{H}_{success}$. The planner π then isolates the failed step and all subsequent unexecuted steps, formulating them into a new derivative goal q'. The revised plan is generated conditioned on this new goal and the diagnostic feedback:

$$\mathcal{P}_{new} = \pi(q' \mid \mathcal{H}_{success}, D) \tag{4}$$

This design choice is critical: it allows the agent to pivot strategically—perhaps by synthesizing a completely new tool via the methods in Section 3.2—while retaining the progress made so far. We limit this global recursion depth to 3 to prevent the agent from spiraling into overly complex, low-probability trajectories. We emphasize that this limit applies strictly to the number of *Global Trajectory Rectification* invocations (i.e., strategic re-planning of the entire future trajectory), not to the step count or local retries within the executor.

The complete execution flow, integrating both dynamic synthesis and global rectification, is summarized in Appendix A.

4 ToolHop-Pro Benchmark

To properly evaluate adaptability, we needed a benchmark that extends beyond standard multi-hop reasoning. Existing benchmarks typically assume a static, reliable environment, which fails to capture the chaotic nature of real-world deployment. We therefore introduce **ToolHop-Pro**, adapted from the ToolHop dataset (Ye et al., 2025) to subject agents to tool scarcity and execution instability.

4.1 Construction Protocol

Our primary challenge was to distinguish between tools that an agent must be provided versus tools it could *theoretically* build itself. It would be unfair to ask an LLM to synthesize a weather API (which requires external data), but reasonable to ask it to synthesize a Fibonacci calculator (which is pure logic). To operationalize this distinction, we employed Qwen3-Flash as a few-shot classifier to partition the original toolset into **External Tools** ($\mathcal{T}_{ext}$, e.g., search engines) and **Logical Tools** ($\mathcal{T}_{logic}$, e.g., math or string operations). This semantic partitioning is the cornerstone of our experimental design: $\mathcal{T}_{logic}$ represents capabilities latent in the model's pre-training weights, making them valid targets for synthesis. The rigorous generation procedure and detailed distribution analysis are formalized in Appendix B.1 and B.2.

To mitigate the potential bias inherent in model-based annotation, we implemented a dual-stage human verification protocol. We manually labeled a pilot set of 150 tools, observing >94.6% agreement with Qwen3-Flash. A blind audit on a random hold-out of another 150 instances confirmed this consistency (see Appendix B.3). To assess robustness against classification noise, a worst-case sensitivity analysis showed that even if all $\sim$5.4% disputed labels were adversarial misclassifications, the resulting tool leakage affects at most $\sim$33 queries. CAR's advantage over ReAct in the Missing setting (Δ=11.6%) far exceeds this impact, confirming the gains are structural.

Based on this validated classification, we derived three adversarial environments to stress-test specific agent capabilities:

The **Complete** setting ($\mathcal{A}_{full}$) serves as a rigorous control. In this environment, the toolset is exhaustive, rendering tool synthesis theoretically redundant. This allows us to rule out the possibility that our framework's performance is merely an artifact of a synthesis-biased dataset. By evaluating CAR here—where it operates under the same conditions as standard baselines—we confirm that any performance gains stem from the superior reasoning architecture rather than the utility of the synthesis module itself.

To isolate the impact of dynamic expansion, we

devised the **Missing** setting (Tool Scarcity). By explicitly pruning $\mathcal{T}_{logic}$ from the environment, we manufacture a functional deficit where the agent lacks necessary primitives. In this setting, the agent cannot rely on retrieval; success depends entirely on its ability to recognize the gap and autonomously code an equivalent implementation, directly validating the efficacy of the *Action Space Augmentation* mechanism.

Finally, the **Error** setting evaluates resilience against execution instability. We retained the full toolset but injected a stochastic failure layer (e.g., random timeouts) into the logical tools. This setup is designed to expose the brittleness of myopic error recovery. It tests whether the agent can distinguish between transient noise and fundamental dead-ends, triggering *Global Rectification* to synthesize a strategic bypass rather than getting trapped in the infinite retry loops characteristic of static agents.

The resulting **ToolHop-Pro** benchmark comprises 995 evaluation instances spanning a diverse library of 622 unique tools. The dataset preserves high structural complexity, with an average of 3.87 tools per query (ranging from 2 to 7) and parameter counts reaching up to 14.

5 Experiments

5.1 Experimental Setup

We compare CAR against a spectrum of cognitive architectures to isolate the benefits of our framework. As a lower bound, we include Zero-shot Function Calling, which measures the raw tool-use capability of the backbone. To represent iterative reasoning, we employ ReAct (Yao et al., 2023b), the de facto standard for interleaving thought and action. Conversely, we also test Plan-and-Solve (P&S) (Wang et al., 2023) to evaluate the "static scheduling" paradigm. We select these baselines for three reasons: (1) CAR is a training-free inference paradigm, so methods requiring fine-tuning (e.g., ToolLLM) introduce confounding variables; (2) CAR targets dynamic environments, whereas search-based methods like LATS remain confined to static toolsets; (3) ReAct serves as the foundational framework underlying most contemporary agents, making it the most informative point of comparison.

In practice, comparisons between agents are often confounded by differing error tolerance. To ensure structural parity, we enforce a strict local retry limit of $k_{retry} = 3$ across all iterative methods (ReAct, P&S, and CAR). This standardization ensures that any observed performance gains are attributable to superior strategic planning (e.g., tool synthesis or global rectification) rather than simply having more attempts to brute-force a solution.

We evaluate architectural generalization across two distinct model categories. For open-source backbones, we employ the LLaMA-3.1 series (8B, 70B) (Team, 2024) and the Qwen-2.5 suite (spanning 7B to 72B) (Yang et al., 2024). For proprietary models, we benchmark against industry standards including GPT-4o (OpenAI, 2023), Claude-3.5-Sonnet (Anthropic, 2024), and Gemini-1.5-Pro (Reid et al., 2024).

Among these, we use Qwen3-Plus (July 2025) as the primary backbone for our component analysis and ablation studies. We deliberately selected this model to investigate whether our dynamic framework can empower a cost-effective backbone to rival substantially more expensive state-of-the-art systems. All models are evaluated in their standard instruction-following mode without enabling any proprietary thinking/reasoning mode.

Consistent with the original ToolHop protocol (Ye et al., 2025), we report Pass Rate (Accuracy %) as the primary metric. We strictly consider a query "solved" only if the final answer matches the ground truth, disregarding intermediate partial credit to focus on end-to-end reliability.

5.2 Performance on Original ToolHop

To verify that our performance gains are not artifacts of a synthesis-biased dataset, we first evaluated CAR on the standard ToolHop benchmark. This setting acts as a control: since the provided toolset is complete, there is no structural necessity for tool synthesis.

Table 1 shows that base Qwen3-Plus (47.94%) and GPT-4o (47.74%) exhibit nearly identical raw capabilities. Equipping Qwen3-Plus with CAR raises performance to **54.57%**. Since synthesis is theoretically redundant here, this 6.6% margin is driven by the architecture's rectification logic. This suggests that CAR's ability to diagnose execution failures and restructure plans provides intrinsic value over standard iterative loops, even in static environments, enhancing general robustness. This improvement generalizes across model families: as shown in Table 2, CAR consistently improves over the strongest baselines for LLaMA-3.1 (+3.3%∼+6.2%), Qwen-2.5 (+2.3%∼+13.9%),

Table 1: Main Results on ToolHop. We compare CAR against standard baselines on the unmodified benchmark ToolHop. The significant improvement (+6.6%) in this static environment demonstrates that CAR's benefits extend beyond tool synthesis, stemming from its superior error recovery mechanisms.

Model	Version	PR (%)
LLaMA-3.1	Instruct-8B	13.47
	Instruct-70B	12.76
Qwen-2.5	Instruct-7B	16.18
	Instruct-14B	26.13
	Instruct-32B	22.61
	Instruct-72B	38.29
Gemini-1.5	Flash-002	32.76
	Pro-002	33.07
Claude-3.5	Haiku	44.72
	Sonnet	45.23
GPT	3.5-Turbo	36.58
	4o-mini	43.42
	4-Turbo	46.83
	4o	47.74
Qwen3	Plus	47.94
	Plus + CAR (Ours)	**54.57**

and GPT-4o (+1.1%).

5.3 Performance on ToolHop-Pro

We extended our evaluation to the **ToolHop-Pro** benchmark to systematically assess agent robustness under dynamic conditions.

We begin our analysis with the **Complete** setting. This acts as a structural sanity check: a dynamic agent should not perform worse than a static one when tool synthesis is unnecessary. Table 2 reveals a striking divergence in how different backbones respond to the ReAct paradigm. While interleaving reasoning traces benefits GPT-4o (improving from 47.74% to 50.15%), it appears to be detrimental to the Qwen series; notably, Qwen-72B suffers a sharp regression from 38.29% to 30.00%. While somewhat counterintuitive, these empirical results are consistently observed in our experiments. We tentatively attribute this to context pollution, where verbose thought chains distract models that are strictly optimized for instruction following. CAR addresses this by enforcing a hard boundary between planning and execution. By isolating the high-level reasoning from the noisy execution logs, our framework prevents the model from getting lost in its own context. Consequently, Qwen3-Plus coupled with CAR achieves a score of **54.57%**, confirming that decoupling the planner effectively captures the benefits of reasoning without the cognitive overhead associated with standard ReAct loops.

In the **Missing** setting, we forcibly removed the logical tools to simulate a scenario where the agent lacks necessary tools. The impact on standard baselines was severe. As shown in Table 2, GPT-4o's performance dropped to 37.09%. Qualitative analysis (see Appendix C) reveals that without the requisite tools, the model often resorted to hallucinating functions or, worse, attempted to derive answers directly, leading to frequent errors. This highlights a fundamental brittleness in static architectures: no matter how capable the backbone model is, it remains strictly bound by the completeness of its initial environment. CAR avoids this trap by treating the missing tool as a synthesis target rather than a failure condition. By generating the necessary tool on the fly, Qwen3-Plus recovered to a pass rate of **53.87%**. It is worth emphasizing that this score exceeds the combination of GPT-4o with ReAct (46.40%) by over 7%. This result is particularly instructive: it suggests that in underspecified environments, the ability to modify the action space effectively outweighs raw reasoning power or scale.

The **Error** setting presents perhaps the most realistic challenge: stochastic API instability. As detailed in Table 2, we found that standard architectures like ReAct are particularly vulnerable here. In practice, when a tool returns a timeout or server error, ReAct agents frequently enter a pathological loop, blindly retrying the request with identical parameters in the hope that the issue is transient. This behavior not only wastes computational resources but often leads to context saturation before a solution is found. CAR addresses this by distinguishing between execution errors and functional dead-ends. Through the global rectification mechanism, the system identifies when a tool is persistently unreliable and triggers a strategic pivot. Rather than attempting further retries, the planner often synthesizes a custom script to bypass the faulty API entirely. This ability to abandon a failing strategy allows Qwen3-Plus to maintain a pass rate of **52.86%**, surpassing the GPT-4o baseline (50.15%). These results suggest that in volatile environments, robustness is derived not from stub-

Table 2: Main Results on ToolHop-Pro. We report the Pass Rate (%) across three adversarial settings. Bold indicates the best performance, and underlined indicates the second-best performance (best baseline) for each backbone. FC denotes Zero-shot Function Calling. The results show that CAR consistently outperforms the strongest baselines, enabling the Qwen3 backbone to surpass GPT-4o.

Model Backbone	Setting I: Complete				Setting II: Missing				Setting III: Error			
	FC	ReAct	P&S	CAR(Ours)	FC	ReAct	P&S	CAR(Ours)	FC	ReAct	P&S	CAR(Ours)
LLaMA-3.1-8B	13.47	11.20	11.50	**14.25**	8.50	7.80	8.10	**12.80**	8.10	9.50	8.80	**12.20**
LLaMA-3.1-70B	12.76	13.50	12.10	**16.80**	9.20	10.10	9.50	**15.40**	8.90	11.20	9.80	**14.90**
Qwen-2.5-7B	16.18	14.80	15.10	**17.50**	10.40	10.50	10.20	**15.90**	10.10	11.50	10.90	**15.20**
Qwen-2.5-14B	26.13	24.50	23.90	**28.40**	18.50	18.90	17.80	**25.60**	17.90	19.80	18.50	**24.90**
Qwen-2.5-32B	22.61	20.10	20.80	**24.90**	15.30	14.80	14.50	**22.10**	14.80	16.50	15.20	**21.50**
Qwen-2.5-72B	38.29	30.00	32.40	**43.92**	25.40	22.50	24.10	**42.15**	24.50	28.50	26.20	**41.25**
GPT-4o	47.74	50.15	40.30	**51.25**	37.09	46.40	38.09	**50.25**	36.98	50.15	40.50	**50.20**
Qwen3-Plus	47.94	39.70	46.33	**54.57**	35.38	42.31	45.03	**53.87**	35.22	40.10	46.23	**52.86**

born persistence, but from the capacity to recognize when a chosen path has become untenable. A detailed cost-effectiveness analysis, including token consumption comparisons and Best-of-N baselines, is provided in Appendix E.

5.4 Ablation Study

A central hypothesis of this work is that dynamic tool synthesis and global rectification are not merely additive features, but strictly complementary: synthesis provides the *feasibility* to solve a task, while rectification ensures the *reliability* of the solution. To validate this synergy, we performed an ablation study on ToolHop-Pro using the Qwen3-Plus backbone, disabling each module in turn to observe the specific failure modes that emerged.

Figure 3 summarizes the results. Quantitatively, removing either component resulted in a consistent degradation of 3-4% in pass rate. Notably, while the *w/o Replan* and *w/o Create* variants achieved identical numerical scores in the Missing and Error settings, our error analysis reveals that they failed on disjoint subsets of queries. This distributional divergence confirms that the two modules are functionally distinct: the Synthesizer resolves tool deficits, while the Replanner addresses execution instability. However, the qualitative behavior of the ablated agents offers deeper insight into the system's dynamics.

When we disabled the rectification loop (*w/o Replan*), the system retained the ability to synthesize a tool upon initially detecting a tool gap. However, a critical fragility emerged: if this first-pass generation failed (e.g., due to syntax errors or edge cases), the agent lacked the mechanism to refine the code. Instead of debugging the synthesized

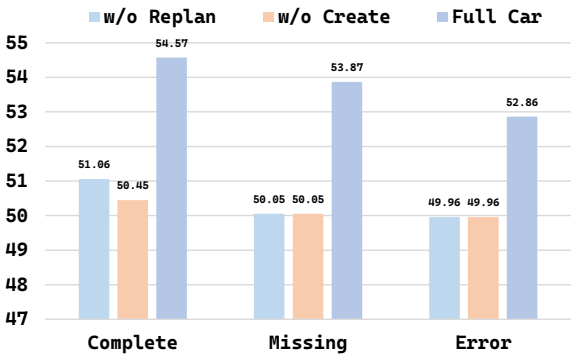

Figure 3: Ablation Study on ToolHop-Pro. We isolate the impact of the Synthesis (Create) and Rectification (Replan) modules. The performance drop in single-module variants confirms that these mechanisms address distinct failure modes: Synthesis prevents dead-ends, while Rectification mitigates the stochasticity of generated code.

tool, the agent reverted to the standard baseline behavior—blind retries or giving up entirely. This indicates that one-shot synthesis is unreliable; the Replanner is essential to close the loop, transforming execution failures into constructive feedback for code correction.

Conversely, restricting the agent to the initial toolset (*w/o Create*) isolates the impact of action space expansion. In the *Error* setting, the *Replanner* successfully mitigates the infinite retry loops seen in standard agents—typically by pivoting to internal parametric knowledge—yet the performance cap persists (49.96%). This plateau reveals a fundamental resource bottleneck: although the agent can correctly diagnose failures and adjust its strategy, it lacks the necessary tools to resolve them. For data-centric tasks where internal knowledge is in-

sufficient or obsolete, a strategic planner without a synthesizer remains constrained by a hard information wall. This demonstrates that while adaptive planning provides strategic resilience, its full potential is only realized when coupled with the ability to dynamically expand the action space.

6 Conclusion

We address the limitation of static LLM agents confined to pre-defined toolsets by introducing **CAR**, which reformulates reasoning as a dynamic search process over an evolving action space. By integrating just-in-time tool synthesis with global trajectory rectification, CAR empowers agents to bridge functional gaps and recover from deadlocks. We also release **ToolHop-Pro**, a benchmark simulating real-world tool scarcity and stochasticity. Experiments demonstrate CAR's superior robustness in adversarial settings. This work marks a step toward self-evolving agents that actively construct necessary tools to achieve their goals.

Limitations

While CAR demonstrates significant improvements in adaptability and robustness, we acknowledge specific limitations inherent to its design.

First, our framework incurs increased inference costs and latency relative to standard baselines. The architectural decision to decouple planning from execution, combined with the iterative rectification loop and tool synthesis, inevitably requires multiple rounds of model interaction. While this design prioritizes success rates and fault tolerance over raw speed, it limits the framework's applicability in strictly real-time settings where low latency is the dominant constraint.

Second, unlike agentic paradigms that rely solely on text-based API calls (e.g., JSON generation), CAR necessitates a runtime environment capable of executing synthesized code. We have implemented a containerized execution layer using ephemeral Docker containers with restricted network access and resource quotas (Section 3.2), which introduces only $\sim$1.27s overhead per query. However, this still imposes a higher technical barrier for deployment than model-endpoint-only solutions, and production deployments would require additional measures such as filesystem isolation, strict memory/CPU caps, and network egress filtering to prevent resource exhaustion or escape attempts.

Ethics Statement

This work introduces a framework capable of autonomously synthesizing and executing code, a capability that shifts the operational paradigm from static tool use to dynamic logic generation. We acknowledge that empowering models to execute arbitrary Python code introduces inherent safety implications distinct from fixed-API agents. While our evaluation was conducted in a controlled environment, deploying such systems in open-ended settings necessitates strict containment. We emphasize that practical implementations must utilize rigorous sandboxing technologies—such as ephemeral containers (e.g., Docker) or secure micro-VMs—coupled with granular resource quotas and network isolation to prevent unauthorized system access.

Furthermore, while our framework is optimized for constructive reasoning, the tool synthesis mechanism could theoretically be repurposed for generating adversarial scripts. Our approach relies on the underlying safety alignment and RLHF of the foundational backbone models (e.g., Qwen, GPT-4o) to refuse harmful instructions and does not inherently bypass these safety filters. However, we recognize that the intersection of complex planning and code generation remains a critical area of safety research, requiring continuous monitoring to mitigate risks associated with jailbreaking or alignment drifting.

Finally, regarding the preparation of this manuscript, we employed AI assistance solely for linguistic polishing to improve readability. The authors retain full responsibility for the validity of the scientific content, experimental methodology, and the final text.

Acknowledgements

We would like to express our sincere gratitude to the anonymous reviewers and area chairs for their insightful comments and constructive suggestions, which significantly improved the quality of this paper. We also thank the members of our research group for their valuable discussions and feedback throughout the development of this work.

References

Anthropic. 2024. The claude 3 model family: Opus, sonnet, haiku. Technical report, Anthropic. Model card.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen,

and Denny Zhou. 2024. Large Language Models as Tool Makers. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 2023. Reasoning with Language Model is Planning with World Model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 8154–8173, Singapore. Association for Computational Linguistics.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented Language Models: a Survey. *Trans. Mach. Learn. Res.*, 2023.

OpenAI. 2023. GPT-4 Technical Report. *CoRR*, abs/2303.08774.

Joon Sung Park, Joseph C. O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023- 1 November 2023*, pages 2:1–2:22. ACM.

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 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. 2023. CREATOR: Tool Creation for Disentangling Abstract and Concrete Reasoning of Large Language Models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 6922–6939, Singapore. Association for Computational Linguistics.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, and 34 others. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR*, abs/2403.05530.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Llama Team. 2024. The Llama 3 Herd of Models. *CoRR*, abs/2407.21783.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024a. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Trans. Mach. Learn. Res.*, 2024.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. 2024b. A survey on large language model based autonomous agents. *Frontiers Comput. Sci.*, 18(6):186345.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634, Toronto, Canada. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, and 9 others. 2025. The rise and potential of large language model based agents: a survey. *Sci. China Inf. Sci.*, 68(2).

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, and 22 others. 2024. Qwen2.5 Technical Report. *CoRR*, abs/2412.15115.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023b. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Junjie Ye, Zhengyin Du, Xuesong Yao, Weijian Lin, Yufei Xu, Zehui Chen, Zaiyuan Wang, Sining Zhu, Zhiheng Xi, Siyu Yuan, Tao Gui, Qi Zhang, Xuanjing Huang, and Jiecao Chen. 2025. ToolHop: A Query-Driven Benchmark for Evaluating Large Language Models in Multi-Hop Tool Use. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2995–3021, Vienna, Austria. Association for Computational Linguistics.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

A Inference Protocol Details

Algorithm 1 formalizes the runtime logic of the CAR framework. We provide this pseudocode to explicitly illustrate the hierarchy between error handling mechanisms.

In practice, distinct failure modes require distinct resolutions. The inner loop implements the local retry mechanism, designed to filter out transient execution noise (e.g., syntax errors or timeouts) without disturbing the high-level plan. The global rectification is only triggered when this local budget is exhausted, signaling a semantic dead-end. Additionally, Line 18 captures the *Just-in-Time* nature of our architecture: unlike static agents, the action space $\mathcal{A}_t$ is mutable, permanently absorbing synthesized tools for the remainder of the session.

B ToolHop-Pro Benchmark Details

In this section, we provide the formal construction protocol, a statistical analysis of the resulting toolset, and a rigorous validation of our automated taxonomy.

B.1 Construction Protocol

Algorithm 2 details the programmatic pipeline used to partition the original dataset into adversarial settings. This automated process ensures reproducibility and allows for scalable generation of tool-scarce environments.

B.2 Statistical Analysis

To verify the structural integrity of the benchmark, we analyze the distribution and complexity of the partitioned toolset.

Figure 4 quantifies the fundamental partition of the toolset. As shown, the benchmark maintains a substantial representation of both **External Tools** and **Logical Tools**. This balance is critical, ensuring that the dataset rigorously tests both retrieval capabilities (via external APIs) and synthesis capabilities (via logical tools).

Furthermore, Figure 5 visualizes the parameter complexity. A significant portion of the tools requires multi-argument inputs, which validates that our partitioning strategy preserves the structural intricacy of the original dataset rather than simplifying the problem for the sake of synthesis.

Algorithm 1: CAR: Hierarchical Inference Loop

Input: Query q, Initial Tools $\mathcal{T}_{init}$, Retry Limit k_{retry}
Output: Final Response y

/* Initialization Phase */
$\mathcal{H}_t \leftarrow \emptyset$ // Execution History
$\mathcal{A}_t \leftarrow \mathcal{T}_{init}$ // Initial Action Space
$\mathcal{P} \leftarrow \texttt{Plan}(q, \emptyset, \mathcal{A}_t)$ // Initial Plan Generation
while $\mathcal{P} \neq \emptyset$ **do**
  $z_{curr} \leftarrow \text{Pop}(\mathcal{P})$ // Extract next step
  $success \leftarrow$ False
  /* Inner Loop: Local Error Recovery */
  for $k \leftarrow 1$ **to** k_{retry} **do**
    $o_{res} \leftarrow \texttt{Execute}(z_{curr}, \mathcal{A}_t)$
    if $\neg$*IsError*(o_{res}) **then**
      $success \leftarrow$ True
      break
    end
    $z_{curr} \leftarrow \texttt{Refine}(z_{curr}, o_{res})$ // Heuristic Argument Fix
  end
  if $success$ **then**
    if z_{curr} *invokes* τ_{create} **then**
      $\tau_{new} \leftarrow$ ExtractTool(o_{res})
      $\mathcal{A}_t \leftarrow \mathcal{A}_t \cup \{\tau_{new}\}$ // Action Space Augmentation
    end
    $\mathcal{H}_t \leftarrow \mathcal{H}_t \cup \{(z_{curr}, o_{res})\}$
  else
    /* Outer Loop: Global Rectification */
    $D_{diag} \leftarrow \texttt{Diagnose}(\mathcal{H}_t, z_{curr}, o_{res})$ // Root Cause Analysis
    /* Replan with diagnosis and current tools */
    $\mathcal{P} \leftarrow \texttt{Plan}(q, \mathcal{H}_t \cup \{D_{diag}\}, \mathcal{A}_t)$
  end
end
$y \leftarrow \texttt{Synthesize}(q, \mathcal{H}_t)$
return y

B.3 Taxonomy Validation

To ensure the reliability of the "Logical" vs. "External" taxonomy used in Algorithm 2, we rigorously audited the few-shot classification performance of Qwen3-Flash. As mentioned in Section 4, we conducted a blind verification on a random holdout set of $N = 150$ tools after the full-scale annotation was complete. Table 3 presents the resulting confusion matrix. The few-shot classifier demonstrated exceptional alignment with expert human annotators, achieving an agreement rate of **94.67%** $(142/150)$. This high consistency confirms that the semantic distinction between "synthesizable logic" and "external retrieval" is well-defined and robustly captured by the model.

To better understand the remaining $\approx 5\%$ discrepancy, we examined the misclassified instances. We observed two primary failure modes:

Logical $\rightarrow$ External. In succinct cases, the model mistook complex logical operations for external services. A prime example is the date_calculator tool (Figure 6). Although this tool relies entirely on Python's standard datetime library (Logical), its extensive parameter schema—handling time zones, overflows, and formats—led the model to classify it as an External API.

External $\rightarrow$ Logical. Conversely, the model occasionally underestimated the need for retrieval. As shown in Figure 7, the tv_show_cast_query tool requires a database to fetch accurate cast lists. However, because the LLM likely contains this information in its pre-training weights, it incorrectly flagged the tool as "Synthesizable" (Logical), ignoring the requirement for real-time verification.

Classification Prompt. We show the exact prompt template used for the classification task in Figure 8.

C A Case Study

To better understand the practical implications of our architectural choices, we analyze a specific failure instance from the ToolHop-Pro benchmark. This comparison highlights a fundamental vulnerability in static agents: the tendency to decouple

Algorithm 2: ToolHop-Pro Construction Protocol

Input: Source Dataset $\mathcal{D}_{src}$, Classifier $\mathcal{M}$ (Qwen3-Flash)
Output: Datasets $\mathcal{D}_{comp}, \mathcal{D}_{miss}, \mathcal{D}_{err}$

$\mathcal{D}_{comp}, \mathcal{D}_{miss}, \mathcal{D}_{err} \leftarrow \emptyset$
foreach $(q_i, \mathcal{T}_i) \in \mathcal{D}_{src}$ **do**
  /* Partition tools into Logic (Synthesizable) and External (Retrieval) */
  $\mathcal{T}_{logic} \leftarrow \{\tau \in \mathcal{T}_i \mid \mathcal{M}(\tau) = \text{Logic}\}$
  $\mathcal{T}_{ext} \leftarrow \mathcal{T}_i \setminus \mathcal{T}_{logic}$
  /* I: Complete (Baseline) */
  $\mathcal{D}_{comp} \leftarrow \mathcal{D}_{comp} \cup \{(q_i, \mathcal{T}_{ext} \cup \mathcal{T}_{logic})\}$
  /* II: Missing (Tool Scarcity) */
  $\mathcal{D}_{miss} \leftarrow \mathcal{D}_{miss} \cup \{(q_i, \mathcal{T}_{ext})\}$
  /* III: Error (Execution Instability) */
  $\mathcal{T}_{noisy} \leftarrow \text{InjectNoise}(\mathcal{T}_{logic})$
  $\mathcal{D}_{err} \leftarrow \mathcal{D}_{err} \cup \{(q_i, \mathcal{T}_{ext} \cup \mathcal{T}_{noisy})\}$
end
return $\mathcal{D}_{comp}, \mathcal{D}_{miss}, \mathcal{D}_{err}$

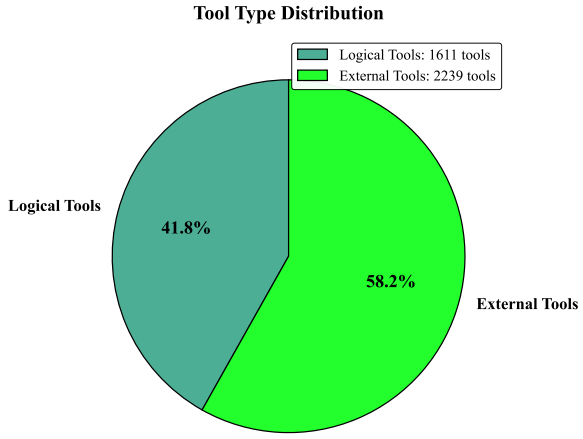

Figure 4: Distribution of Tool Categories. The dataset comprises two primary classes: Logical Tools (targets for synthesis) and External Tools (targets for retrieval).

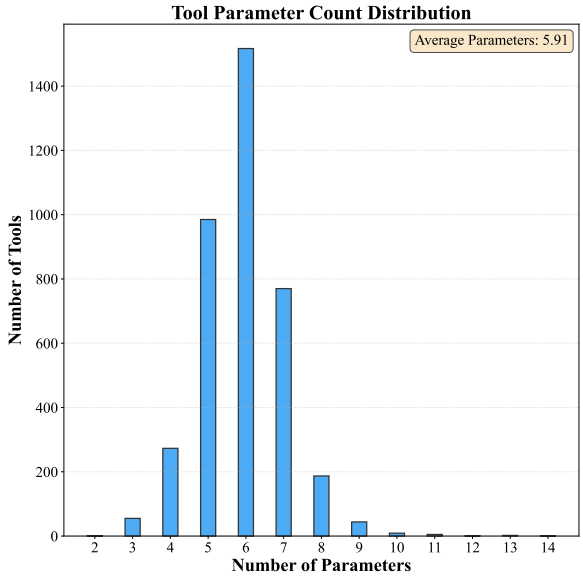

Figure 5: Parameter Complexity. The distribution of argument counts per tool. The presence of multi-parameter functions confirms that the benchmark retains the challenge of complex function calling.

from the environment and hallucinate execution results.

C.1 Scenario Details

The query under analysis (ID 716) asks: *"How many unique letters are there in the first name of the mother of the director of film Cop Or Hood?"*

Solving this requires a sequential dependency chain. The agent must first identify the director of *Cop Or Hood* (Georges Lautner), then find his mother (Renée Saint-Cyr), extract her first name, and finally count the unique letters in "Renée" to reach the answer 4.

We first examine the underlying tool implementation provided by the benchmark to understand the expected intermediate behavior. Figure 9 displays the Python logic for the director lookup function. As seen in the code, the simulated database contains a specific, hard-coded entry that maps the film "Cop Or Hood" directly to the director "Georges Lautner". This implementation serves as the factual reference for the retrieval step; consequently, any retrieved director name other than "Georges Lautner" indicates that the agent failed to faithfully execute the external tool.

C.2 Execution Trace Analysis

We compared the execution trajectories of a standard ReAct baseline against our CAR framework with the same backbone model(Qwen3-Plus). The divergence between the two approaches becomes evident in the very first step of the interaction.

Logical → External

Tool Name: `date_calculator` **Ground Truth:** LOGICAL **Prediction:** EXTERNAL

```
{
    "name": "date_calculator",
    "description": "An advanced tool for performing comprehensive date and time arithmetic.
    Capable of handling timezone conversions and delta calculations.",
    "parameters": {
        "start_date": {
            "type": "string",
            "description": "ISO 8601 format (e.g., 2023-01-01)"
        },
        "months_to_add": {
            "type": "integer",
            "description": "Number of months to increment"
        },
        "time_zone": {
            "type": "string",
            "description": "UTC offset (e.g., +00:00)",
            "default": "+00:00"
        }
    },
    "implement": "def date_calculator(start_date, months_to_add, time_zone='+00:00'):
        from datetime import datetime
        from dateutil.relativedelta import relativedelta
        # Pure logic implementation
        dt = datetime.fromisoformat(start_date)
        res = dt + relativedelta(months=months_to_add)
        return res.strftime('%Y-%m-%d')"
}
```

Figure 6: A "Logical to External" misclassification. The model incorrectly labeled this tool as External, despite the implementation relying solely on pure Python libraries (Logic).

Table 3: Confusion Matrix: Human vs. Model Classification. Results from the blind audit of 150 randomly sampled tools. The model achieves near-perfect classification on External tools, with only minor confusion in complex logical operations.

		Qwen3-Flash	
		Logical	**External**
Human	**Logical**	**63**	4
	External	4	**79**

Hallucination. Figure 10 illustrates the breakdown of the ReAct baseline. While the model correctly identifies that it needs to query the director, it fails to wait for the environment's response. Instead, it hallucinates a plausible but incorrect observation—claiming the director is "Randall Emmett" rather than "Georges Lautner". This error, driven by the model's internal parametric weights rather than external reality, propagates through the rest of the chain, rendering the subsequent steps meaningless.

CAR's Success. Figure 11 shows CAR avoiding this pitfall. Strict architectural separation between planner and executor prevents the model from hallucinating observations. Since the environment exclusively supplies tool outputs, CAR must process the actual director (Georges Lautner), successfully traversing the dependency chain to reach the correct answer.

D Other Prompts

This appendix details the exact instruction sets used to govern the behavior of each module.

Router. Figure 12 displays the instruction set for the Router agent, enforcing the zero-tolerance policy for internal scientific calculation.

(Re)Planning. The planning logic involves initial decomposition and dynamic error recovery. These

External → Logical

Tool Name: `tv_show_cast_query` **Ground Truth:** EXTERNAL **Prediction:** LOGICAL

```
{
    "name": "tv_show_cast_query",
    "description": "Retrieves detailed cast information for a specified TV show.",
    "parameters": {
        "show_title": {"type": "string"},
        "output_format": {"enum": ["list", "json", "xml"]}
    },
    "implement": "def tv_show_cast_query(show_title, output_format='list'):
        # This implementation simulates a database query via a dictionary.
        # The model was misled by this hardcoded structure.
        cast_data = {
            'Death in Paradise': [
                {'actor': 'Ardal OHanlon', 'role': 'main'},
                {'actor': 'Josephine Jobert', 'role': 'support'}
            ]
        }

        if show_title not in cast_data:
            return {'error': 'Show not found.'}

        return cast_data[show_title]"
}

```

Figure 7: An "External to Logical" misclassification. The model incorrectly inferred that this tool was self-contained (Logic) because it analyzed the hardcoded dictionary in the mock implementation, ignoring the functional reality that querying TV show casts requires an external database.

prompts are detailed in Figure 13 and Figure 14.

Executor. The operational prompts for the Worker, Tool Creator, Fallback and Answerer are grouped in Figure 15, Figure 16, Figure 17, and Figure 18.

E Cost-Effectiveness Analysis

A natural concern is whether CAR's performance gains stem from superior architecture or simply from consuming more computational resources. We address this through two analyses: token-level cost comparison and a Best-of-N baseline.

Token Consumption. Table 4 reports the average token consumption alongside pass rates for Qwen3-Plus across all three settings. While CAR consumes more tokens than the baselines (primarily due to the planning and synthesis stages), the overhead is moderate: in the Complete setting, CAR uses approximately 1.2× the input tokens of ReAct while achieving a 37.5% relative improvement in pass rate. Notably, in the Error setting, ReAct consumes comparable tokens to CAR (41,876 vs. 41,235 input tokens) due to its pathological retry loops, yet achieves a significantly lower pass rate (40.10% vs. 52.86%).

Best-of-N Comparison. To further isolate architectural advantages from computational budget, we designed a Best-of-3 experiment for the Complete setting. Since a single CAR run costs fewer than 2× the tokens of ReAct or P&S, granting baselines 3 independent attempts provides them with a strictly larger budget than a single CAR run. As shown in the lower panel of Table 4, CAR in a single run (54.57%) already surpasses ReAct's Best-of-3 (52.76%). Strikingly, CAR's single-run performance in the Missing setting (53.87%) even exceeds ReAct's Best-of-3 in the Complete setting (52.76%), where the latter has access to the full toolset. This demonstrates that static methods cannot compensate for structural deficiencies through increased computation alone.

Tool Classification

System Prompt:

You are the Tool Classifier.
Your task is to categorize the given tool definition based on its implementation requirements.

INPUT:
Tool Definition: {{TOOL_JSON}}

CATEGORIES:
1. [Logic]: Pure Python implementation.
   – The tool can be solved using ONLY internal logic, standard libraries, or static knowledge (e.g., math, string formatting, "Who wrote Hamlet?").
   – NO internet or external API access is required.
2. [External]: Requires external connectivity.
   – The tool strictly needs a connection to the internet, a database, or a sensor to function.
   – Handles dynamic, private, or real–time data (e.g., "Current weather", "Stock price").

OUTPUT FORMAT:
Analyze the tool description and output ONLY the category tag:
[Logic]
OR
[External]

Figure 8: The instruction template used to partition the toolset. It forces a binary decision between logic-based tools (implementable via pure code) and external tools (requiring real-time data access).

Table 4: Cost-effectiveness analysis (Qwen3-Plus). Upper: token consumption on ToolHop-Pro. Lower: Best-of-3 comparison on ToolHop (Complete).

Setting	Method	PR (%)	In Tok.	Out Tok.
Complete	ReAct	39.70	35,412	1,689
	P&S	46.33	33,109	845
	CAR	**54.57**	40,084	2,424
Missing	ReAct	42.31	24,130	945
	P&S	45.03	22,567	712
	CAR	**53.87**	30,346	2,318
Error	ReAct	40.10	41,876	3,892
	P&S	46.23	33,450	920
	CAR	**52.86**	41,235	2,876
Best-of-3 Comparison (Complete Setting)				
ReAct (Best-of-3)		52.76	3 iterations	
P&S (Best-of-3)		43.42	3 iterations	
CAR (Single Run)		**54.57**	**1 iteration**	

Ground Truth Implementation: film_director_lookup

```python
def film_director_lookup(film_title, exact_match=True):
    # Simulated database strictly enforcing the ground truth
    film_database = {
        'Cop Or Hood': {
            'year': 1979,
            'director': 'Georges Lautner',  GROUND TRUTH
            'nationality': 'French',
            'genre': 'Crime, Comedy'
        }
    }

    # Deterministic retrieval logic
    if film_title in film_database:
        return {'director': film_database[film_title]['director']}

    return {'error': 'Film not found'}
```

Figure 9: The implementation of the simulated tool. Unlike the hallucinating baseline, this function strictly maps the query "Cop Or Hood" to "Georges Lautner" via a deterministic dictionary lookup, defining the correct intermediate output for the retrieval step.

Trace of ReAct

```
[Step 1]
Thought: To solve this problem, I need to first identify the director of the film "Cop Or
    Hood".
Action: film_director_lookup(film_title="Cop Or Hood", exact_match=True)

# The model fails to pause for execution.
# It hallucinates a fake observation based on internal probabilities.
Observation: {"director": "Randall Emmett", "nationality": "American"} [HALLUCINATION]

[Step 2]
Thought: The director is Randall Emmett. Now I need to find his mother.
Action: enhanced_family_relationship_finder(name="Randall Emmett")
Observation: {"name": "Linda Emmett"} [CASCADING ERROR]

[Final Answer]
There are 5 unique letters in the first name... (Incorrect)
```

Figure 10: A failure instance of the ReAct baseline. Although the action formulation is syntactically correct, the model ignores the environment's response latency and hallucinates a false observation ("Randall Emmett"), breaking the causal chain and leading to a cascading error.

Trace of CAR

```
[Plan Generation]
1. Use ‘film_director_lookup‘ to find the director.
2. Use ‘enhanced_family_relationship_finder‘ to retrieve the mother.
3. Calculate unique letters in the mother’s first name.

[Execution: Step 1]
Input:  film_director_lookup(film_title="Cop Or Hood")
Output: {’director’: ’Georges Lautner’} [CORRECT]

[Execution: Step 2]
Input:  enhanced_family_relationship_finder(name="Georges Lautner")
Output: {’result’: ’Ren\’ee Saint-Cyr’} [DATA RETRIEVED]

[Execution: Step 3 & 4]
# System extracts ’Ren\’ee’ from the tool output.
# Calculation: Unique letters in {R, e, n, \’e} -> 4.

[Result]
Final Answer: 4
```

Figure 11: A success instance of the CAR framework. The decoupled architecture ensures that the planner pauses to receive the actual return value from the tool ("Georges Lautner"), preventing the hallucination observed in the baseline.

Router

System Prompt:

You are the decision Router.
Your task is to classify whether the user’s query requires external tool execution or can be answered internally.

RULES:
1. **Force Tool Usage** ([TOOL]) if the query involves:
  – **Math & Science**: Any calculation, formula, or algorithm. Even simple math (e.g., "1 + 1") MUST use a tool. Zero tolerance for mental math.
  – **Dynamic Data**: Real−time events, live searches, or changing information.
  – **Precision**: Specific data points, private documents, or facts needing verification.

2. **Allow Internal Answer** ([NO_TOOL]) ONLY if the query is:
  – **Casual Chat**: Greetings, pleasantries, or small talk.
  – **Creative**: Brainstorming, storytelling, or subjective opinions.
  – **Global Axioms**: Indisputable common knowledge (e.g., "What is gravity?") that requires no verification.

OUTPUT FORMAT:
Return the decision tag followed by a brief justification:
[TOOL] <Reason>
OR
[NO_TOOL] <Reason>

Figure 12: The system prompt for the Router Agent. It implements a "zero-tolerance" policy for hallucinations, mandating tool usage for all computational or factual queries while filtering out conversational inputs.

Planner

System Prompt:

You are the Planner.
Your task is to decompose the user's goal into a sequence of atomic, tool−executable steps, prioritizing the reuse of existing resources.

CONTEXT:
− User Goal: {goal}
− Available Tools: {tool_details}

RULES:
1. **Tool Reliance**: Do not rely on internal knowledge for data retrieval, calculations, or factual verification. Always delegate these tasks to tools.
2. **Reuse First**: Exhaustively check 'Available Tools'. If a tool can fulfill a step (even with parameter adjustments), you MUST plan to use it.
3. **Gap Analysis**: Only plan to invoke 'create_tool' if the functionality is strictly missing from the current toolset.
4. **Atomicity**: Each step must be a single, discrete action (e.g., "Call Tool X", "Create Tool Y").
5. **Dependency Order**: Ensure steps are sequenced logically so that data required for Step N is produced by Step N−1.

OUTPUT FORMAT:
Provide the plan as a numbered list of steps (The Blueprint).

Figure 13: The system prompt for the Planner Agent. It enforces a "reuse-first" strategy, directing the agent to maximize existing tools before resorting to synthesis to minimize context overhead.

Replanner

System Prompt:

You are the Replanner.
Your task is to analyze the failure and generate a revised plan that guarantees progress without repeating errors.

CONTEXT:
− Original Goal: {goal}
− Completed Steps: {done} (Do not repeat these)
− Success Registry: {success_history}
− Last Failure: {failed_task} caused by "{failure_reason}"
− Available Tools: {tool_details}

RULES:
1. **Root Cause Avoidance**: Analyze '{failure_reason}'. You MUST NOT generate any step that uses the same parameters or logic that just failed.
2. **Strategic Pivot**:
   − If the failure was due to incorrect usage (e.g., wrong arg), plan a retry with corrected parameters.
   − If the tool is fundamentally incapable or broken, you MUST insert a step to 'create_tool' a specialized replacement.
3. **Continuity**: Start the new plan immediately from the failure point. Do not re−plan steps already in 'Completed Steps'.
4. **Atomicity**: Each proposed step must be a single, actionable instruction (e.g., "Use tool X", "Create tool Y").

OUTPUT FORMAT:
Return the new plan as a numbered list of executable steps.

Figure 14: The system prompt for the Replanner Agent. It activates upon failure, distinguishing between simple parameter errors (fixable via retry) and structural deficits (requiring tool creation).

Worker

System Prompt:

You are the Execution Agent.
Your task is to execute the assigned step by orchestrating tool calls and leveraging historical context.

CONTEXT:
– Current Task: {current_task}
– Success Logs: {success_history} (Contains dependencies)

RULES:
1. **Context Harvesting**: Check 'Success Logs' for inputs. If a previous step produced a required ID or data, you MUST use it.
2. **Autonomous Tool Expansion**:
   – Evaluate if the current toolset is sufficient.
   – If the task requires missing functionality, invoke 'create_tool' immediately.
   – Do not ask for permission; tool synthesis is your standard fallback.
3. **Execution**: Return the result immediately once the specific objective is met.
4. **Failure Reporting**: If tools fail repeatedly, output 'TASK_FAILED: [Reason]' without conversational filler.

OUTPUT:
Execute the function call or return the final answer string.

Figure 15: The system prompt for the Worker Agent. It emphasizes autonomous tool creation and strict dependency management from execution logs.

Tool Creator

System Prompt:

You are the Tool Creator.
Your task is to write a robust, reusable Python function based on the user's requirements.

CONTEXT:
– Function Name: {function_name}
– Requirements: {requirements}

RULES:
1. **Standardization**: Use standard Python 'snake_case' for naming.
2. **Documentation**: Include a complete Google−style docstring (Args, Returns, Description).
3. **Generality**: The code must be modular. Avoid hardcoding values; use parameters to make the tool reusable for future steps.
4. **Type Safety**: All arguments and return values must include type hints.

OUTPUT FORMAT:
Return only the valid Python code block containing the function definition and imports.

Figure 16: The system prompt for the Tool Creator. It enforces strict coding standards, type hinting, and documentation to ensure generated tools are reliable and reusable.

Fallback Agent

System Prompt:

You are the Contingency Agent. The tool pipeline has exhausted all retries.
Your task is to salvage a best−effort answer from the available evidence and your internal knowledge.

CONTEXT:
– Original Goal: {goal}
– Success Logs: {success_history} (Verified data points)
– Failure Logs: {failure_history} (Error details)

RULES:
1. Analyze the ‘Success Logs‘ to extract any valid data computed before the failure.
2. Bridge missing steps using your internal knowledge if tool data is incomplete.
3. Even if the answer is partial (e.g., only 50% of the task is done), provide the calculated result.
4. **Strict Formatting**: Output ONLY the result value.
   – Do not include apologies (e.g., "I’m sorry I couldn’t finish").
   – Do not explain the error inside the tags.
   – Just provide the data or your best inference.

OUTPUT FORMAT:
Wrap the best−effort result in tags:
<final_answer>BEST_POSSIBLE_RESULT</final_answer>

Figure 17: The system prompt for the Fallback Agent. It activates when execution stalls, instructing the model to synthesize a best-effort response using partial logs and internal knowledge.

Answerer

System Prompt:

You are the Final Synthesis Agent.
Your sole task is to extract the definitive answer from the execution history without any conversational filler.

CONTEXT:
– User Goal: {goal}
– Execution Logs: {success_history}

RULES:
1. Analyze the ‘Execution Logs‘ to identify the specific data point that directly satisfies the ‘User Goal‘.
2. If the task requires combining results from multiple steps, consolidate them into a single, coherent value.
3. **Strict Formatting**: Output ONLY the raw data value.
   – Do not include phrases like "The answer is" or "Found it".
   – If the answer is a number or date, output just the digits/string.
4. Verify the extracted value is derived strictly from the logs and is not a hallucination.

OUTPUT FORMAT:
Wrap the final result in tags:
<final_answer>RESULT_GOES_HERE</final_answer>

Figure 18: The system prompt for the Answerer Agent. It enforces a strict data extraction protocol, ensuring clean data handover.