CookBench: A Multimodal Interactive Benchmark for Long-Horizon Embodied Planning in Complex Cooking Scenarios

Anonymous ACL submission

Abstract

Recent multimodal agents have demonstrated substantial progress in digital environments, yet whether these capabilities seamlessly transfer to long-horizon, physically grounded planning remains an open question. To bridge this evaluation gap, we introduce CookBench, a high-fidelity multimodal interactive benchmark designed for long-horizon embodied planning in complex cooking scenarios. Featuring 133 single-dish tasks with an average length of over 130 steps, CookBench goes beyond existing benchmarks by incorporating large actions space, abundant interactive entities and high-fidelity physical dynamics. Through systematic evaluation of representative agent frameworks and state-of-the-art vision-language models via the Embodied Planning Machine evaluation adaptation framework, we expose a striking reality gap: even GPT-5.4-driven agents peak at just 8.35%. Ultimately, CookBench provides a crucial testbed for this digital-to-physical transition, tracing execution breakdowns to three fundamental capability deficits: absent physical grounding, poor closed-loop feedback comprehension, and severe embodied state amnesia.

1 Introduction

Recent multimodal agents excel in digital problem-solving, driving a community-wide push toward physically grounded environments. To evaluate this transition, existing research relies on digital-agent benchmarks (Koh et al., 2024; Xie et al., 2024; Rawles et al., 2024; Mialon et al., 2023) and embodied planning benchmarks (Shridhar et al., 2020a; Puig et al., 2018; Padmakumar et al., 2022; Zhang et al., 2025a). Yet, a critical misalignment persists: digital agents master extended planning but lack physical grounding, while embodied testbeds provide basic spatial constraints but lack planning depth.

Current embodied benchmarks abstract away real-world complexity, as shown in Table 1, suffering from five fundamental limitations: (1) **short task horizons**, which typically cap tasks at roughly 15 steps, severely underemphasizing extended workflows; (2) **scarce interactive entities**, offering a highly restricted vocabulary of objects while completely lacking functionally specific tools necessary for complex interaction; (3) **restricted action spaces** that rely on a narrow set of primitives (often fewer than 16) and fail to provide rich semantic feedback to guide error recovery; and (4) **simplistic physical simulation** that heavily abstracts away continuous state evolution (e.g., temperature, liquid volume) and ignores irreversible execution risks like spillage or overcooking; (5) **singular evaluation metrics** that primarily on binary task completion, fundamentally ignoring the continuous physical quality and overall efficiency of the execution. Consequently, a fundamental question remains unanswered: how do modern agents perform when long-horizon planning is inextricably coupled with persistent physical interactions?

To bridge this gap, we introduce CookBench (Figure 1), a multimodal interactive benchmark. Built on a high-fidelity Unity simulator, CookBench systematically overcomes the aforementioned limitations by providing: (1) **extended task horizons** through 133 complex single-dish tasks that average over 130 steps; (2) **abundant interactive entities** spanning an array of 221 objects and 40 tools; (3) **large action spaces** via an action library of 85 atomic actions and 13 skills supported by interactive feedback; and (4) **high-fidelity physical dynamics** that natively support continuous state evolution and diverse physical attributes (e.g., temperature, volume and time), shattering the assumption of perfect execution by introducing realistic anomalies (e.g., shattered containers and overcooked ingredients); and (5) **a comprehensive evaluation protocol** integrating a cognitive evaluation substrate with multi-dimensional metrics. Recognizing that genuine embodied planning in-

Figure 1: The CookBench environment. It combines diverse ingredients and tools, action library, continuous physical state changes, irreversible execution risks, and built-in automatic evaluation.

herently demands persistent physical grounding, rich functional tool use, and complex continuous state changes, we anchor our study in the cooking domain.

Experiments expose a striking reality gap: while humans reach ∼97%, even the strongest baseline (Code-as-Policy powered by GPT-5.4) peaks at just 8.35%. This collapse occurs because the physical execution relies on a continuous interaction loop. This loop unravels through a three-stage cascading failure: (1) a breakdown in physical grounding (the inability to map semantic intents to precise actions); (2) a failure in closed-loop repair (struggling to translate environmental friction into state-altering corrections); and ultimately, (3) a decay in embodied working memory (losing track of dynamic states over extended horizons). The main contributions of this paper are as follows:

1. A long-horizon benchmark for physical grounding. We introduce CookBench, a comprehensive testbed integrating high-fidelity physical simulation, a hierarchical action library, and comprehensive evaluation.

2. A comprehensive evaluation protocol. It integrates the Embodied Planning Machine for standardized environment adaptation with multi-dimensional metrics to rigorously quantify physical execution success.

3. A systematic diagnosis of technical gaps. We empirically analyze a spectrum of competitive agentic systems, exposing underlying structural failure modes to clarify future research directions in long-horizon interaction.

2 Related Work

Agentic Reasoning in Digital Environments. Current agents show remarkable success in digital domains, including web navigation (Koh et al., 2024; Wei et al., 2025), OS operation (Xie et al., 2024; Rawles et al., 2024), and general tool-use / reasoning tasks (Liu et al., 2023; Yoran et al., 2024; Yao et al., 2025; Mialon et al., 2023; Wang et al., 2026; Jin et al., 2024). While these benchmarks establish robust multi-step reasoning capabilities, they remain decoupled from the physical world. They typically bypass challenges like 3D spatial interaction, continuous state evolution (e.g., temperature), and irreversible execution failures.

Embodied Planning Benchmarks. Embodied planning research ranges from symbolic text/script abstractions (Shridhar et al., 2020b; Huang et al., 2022) to grounded 3D simulations (Shridhar et al., 2020a; Kim et al., 2024; Puig et al., 2018; Padmakumar et al., 2022; Deitke et al., 2022; Lin et al., 2025). However, tasks in these environments remain relatively short-horizon (avg. ∼10 steps) with coarse-grained action primitives and discrete state changes. The kitchen is a canonical domain for studying coordination and state interpretation (Bollini et al., 2013; Kanazawa et al., 2024; Mavrogiannis et al., 2024; Ivanova et al.,

	Benchmark	Task & Eval.		— Environment & Interaction —					— Core Capabilities —						
		Seq. Len.	Auto. Eval.	Simulator	3D Env.	Objs	Tools	Actions	Temp	Time	Volume	Shape	Physical State Change	Manip. Gran.	Knowledge
Planning	ALFRED (Shridhar et al., 2020a)	~10	✓	AI2-THOR	✓	84	✗	13	✓	✗	✗	✓	Discrete	Coarse	Common
	ReALFRED (Kim et al., 2024)	~14	✓	AI2-THOR	✓	119	✗	13	✓	✗	✗	✓	Discrete	Coarse	Common
	EB-ALFRED (Yang et al., 2025)	~13	✓	AI2-THOR	✓	84	✗	13	✓	✗	✗	✓	Discrete	Coarse	Common
	ALFWorld (Shridhar et al., 2020b)	~9	✓	Text-only	✗	84	✗	13	✓	✗	✗	✓	Discrete	Coarse	Common
	VirtualHome (Puig et al., 2018)	~12	✓	Unity3D	✓	308	✗	23	✗	✗	✗	✗	Discrete	Coarse	Common
	ET-Plan-bench (Zhang et al., 2025a)	~5	✓	Unity3D & Habitat	✓	127	✗	6	✗	✗	✗	✗	Discrete	Coarse	Common
	EmbRACE-3K (Lin et al., 2025)	~8	✓	UnrealCV	✓	–	✗	4	✗	✗	✗	✗	Discrete	Coarse	Common
	TEACh (Padmakumar et al., 2022)	–	✓	AI2-THOR	✓	92	✗	16	✗	✗	✗	✗	Discrete	Coarse	Common
	ProcTHOR (Deitke et al., 2022)	–	✓	ProcTHOR	✓	–	✗	6	✗	✗	✗	✗	Discrete	–	Common
	TaPA (Wu et al., 2023)	~15	✗	–	✗	–	✗	–	✗	✗	✗	✗	Discrete	–	Common
	ParaCook (Zhang et al., 2025b)	–	✓	Web	✗	–	✗	5	✗	✓	✗	✗	Discrete	Coarse	Common + Domain
Agentic Reason.	VisualWebArena (Koh et al., 2024)	–	✓	Web	✗	–	–	–	✗	✗	✗	✗	–	–	Common
	OSWorld (Xie et al., 2024)	–	✓	Computer	✗	–	–	–	✗	✗	✗	✗	–	–	Common
	AndroidWorld (Rawles et al., 2024)	–	✓	Android	✗	–	–	–	✗	✗	✗	✗	–	–	Common
	GAIA (Mialon et al., 2023)	–	✓	Web	✗	–	–	–	✗	✗	✗	✗	–	–	Common
	AgentBench (Liu et al., 2023)	–	✓	Mixed	✗	–	–	–	✗	✗	✗	✗	–	–	Common
	AssistantBench (Yoran et al., 2024)	–	✓	Web	✗	–	–	–	✗	✗	✗	✗	–	–	Common
	τ-bench (Yao et al., 2025)	–	✓	Tool API	✗	–	–	–	✗	✗	✗	✗	–	–	Tool
	BrowseComp (Wei et al., 2025)	–	✓	Web	✗	–	–	–	✗	✗	✗	✗	–	–	Common
	HORIZON (Wang et al., 2026)	–	✓	Mixed	✗	–	–	–	✗	✗	✗	✗	–	–	Common
	CookBench	**>130**	✓	**Unity3D**	✓	**221**	**40**	**98**	✓	✓	✓	✓	**Continuous**	**Fine**	**Common + Domain + Tool**

Table 1: A comparison of representative embodied ai, and agentic reasoning benchmarks. '-' indicates that the metric is not applicable or not explicitly reported. Key metrics include: Seq. Len. (Average number of steps per task); Tools (Number of tools with operation functions); and Manip. Gran. (Manipulation granularity, i.e., coarse-grained task primitives vs. fine-grained interactive actions).

2025). While existing benchmarks explore abstract multi-agent cooperation (Carroll et al., 2019) or asynchronous scheduling efficiency (Gonzalez-Pumariega et al., 2025; Zhang et al., 2025b), they often employ simplified environments or specialized task structures.

While prior work advances individual facets of autonomy, a critical gap remains in systematically evaluating whether digital agent capabilities can reliably transfer to long-horizon, physically grounded execution.

3 CookBench: Environment, Tasks, Knowledge, and Evaluation

In the following subsections, we detail the environment design, the task definitions, the structured knowledge base, and the evaluation protocol.

3.1 CookBench Environment

CookBench is built on *Cooking Simulator*[1], a high-fidelity 3D game featuring stable physical dynamics, rich kitchen objects, and persistent stateful interactions (Haas, 2014). However, since even the most advanced multimodal models struggle with precise zero-shot 3D spatial perception (e.g., exact positioning and morphological attributes), relying solely on raw vision often leads to cascading failures.

To isolate embodied planning capabilities from fundamental perception noise, we invest extensive effort into developing a dedicated state extraction interface via custom engine-level scripting, dubbed the **World Adapter**. This adapter actively provides agents with structured semantic observations, capturing the names, locations, and physical attributes of all entities, alongside the agent's real-time state. Furthermore, the World Adapter supports the real-time detection and semantic mapping of **97** products, **38** liquids, **37** spices, and **49** functional tools, along with diverse placement receptacles and tool-interaction sites.

Transforming this human-centric environment into a rigorous evaluation testbed for autonomous agents requires bridging a massive gap between raw keyboard and mouse inputs and semantic reasoning. To achieve this, we manually map the raw motor controls into a tool-callable semantic action space, comprising **25** action categories and **85** atomic actions. Furthermore, recognizing that relying solely on atomic primitives makes even basic execution (e.g., navigation) unnecessarily intractable, we manually encapsulate **13** functional skills, such as `auto_cut` and `auto_pour`—collectively constituting the **Cerebellum**. Crucially, we endowed the environment with closed-loop execution capabili-

[1] https://store.steampowered.com/app/641320/Cooking_Simulator/

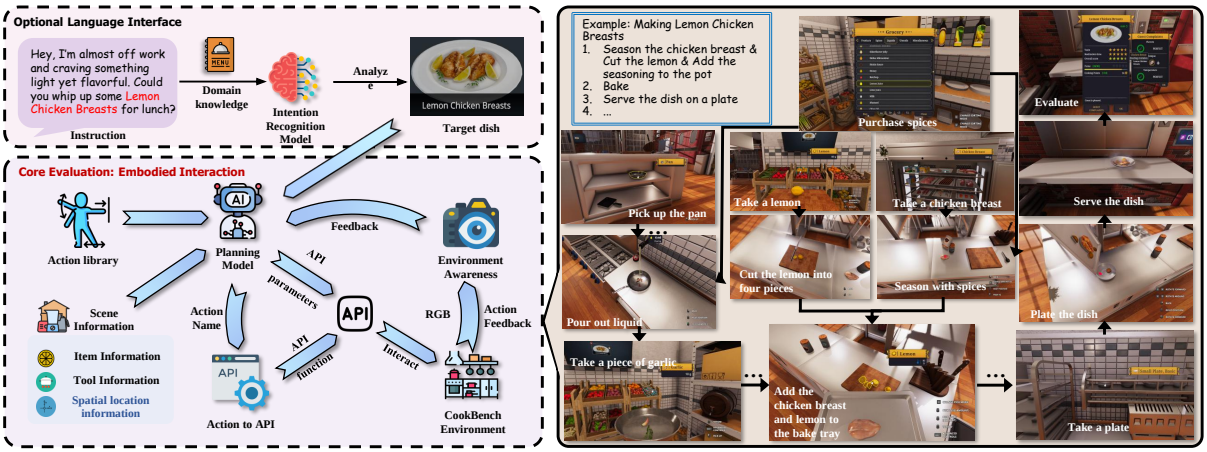

Figure 2: Overview of Cookbench task process. The core challenge of Cookbench is embodied interaction. The model will decide the interaction action with the environment according to the action library, scene information and perceptual feedback information until the task is completed. We also provide optional natural-language instructions (Appendix B.3.1).

ties by designing a diagnostic system that translates raw physical collisions into semantic, actionable textual feedback (see Appendix C.3 for detailed feedback mechanisms).

3.2 Task Definition

CookBench formulates tasks as embodied interactions (Figure 2): agents must engage in sequential decision-making within the environment based on recipe requirements, invoking actions to interact with a vast array of entities and deploy functionally specific tools. Based on their final physical execution outcomes, agents receive an objective evaluation score.

CookBench comprises 133 single-dish tasks (averaging over 130 steps each) alongside numerous multi-dish tasks automatically composed from the single-dish pool. To facilitate systematic analysis, the 133 single-dish tasks are further grouped into six cooking-style categories based on their dominant operations, component organization patterns, and dish styles. Detailed categories are provided in Appendix C.2. This paper focuses primarily on single-dish tasks, as they are already sufficient to expose core long-horizon embodied planning failures in current models.

3.3 CookBench Knowledge Base

Recognizing that current models struggle to autonomously explore and distill interaction heuristics like humans do, we manually collect a comprehensive structured knowledge base to assist agent planning. The knowledge base is augmented with 6 essential components: *action knowledge* detailing the interface protocols for tool-callable actions; *tool knowledge* specifying the functions and affordances of containers and kitchenware; *product knowledge* describing the physical and semantic attributes of ingredients and items; *common Sense* defining the agent's interaction constraints and physical rules; *experience knowledge* providing reusable heuristics to optimize task scores and execution efficiency; and *domain knowledge* specifying recipes for 133 dishes.

3.4 Evaluation Protocol

Cognitive Evaluation Substrate: Embodied Planning Machine To systematically investigate the cognitive mechanisms of multimodal agents transitioning into the physical world, we introduce the *Embodied Planning Machine* (EPM), as illustrated in Figure 3. To the best of our knowledge, EPM serves as the first dedicated cognitive infrastructure explicitly designed to support digital agents in long-horizon, physically grounded interactions. This standardized adaptation layer is strictly necessary; without EPM processing, directly applying original digital-agent implementations leads to complete zero-score runs in our pilot trials due to overwhelming physical feedback noise. Consequently, All agent frameworks are uniformly adapted through the EPM substrate.

Evaluation Metrics. CookBench evaluates agents along two complementary axes: **Dish quality** and **Execution efficiency**.

Dish quality is scored automatically against the

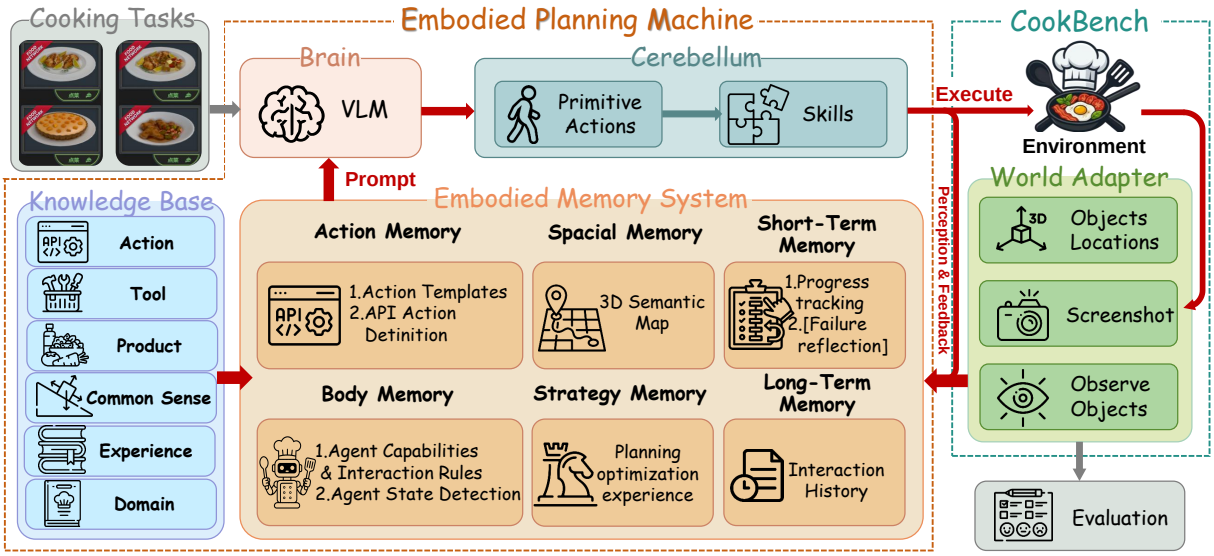

Figure 3: Overview of the Embodied Planning Machine. As a cognitive evaluation substrate, EPM coordinates the information flow across three dimensions: it aligns agent reasoning with a structured **knowledge base**, tracks long-horizon changes via an **embodied memory system**, and processes continuous environmental feedback through the **interaction interface** (via the Cerebellum and World Adapter).

recipe-defined ingredient composition and processing requirements. At the component level, the simulator aggregates four dimensions: taste and proportion, thermal processing, technique, and final serving temperature. Let the corresponding scores be α_k, β_k, γ_k, and δ_k, with weights λ_α, λ_β, λ_γ, and λ_δ. The component-level quality score is computed as

$$\psi_k = \frac{\lambda_\alpha \alpha_k + \lambda_\beta \beta_k + \lambda_\gamma \gamma_k + \lambda_\delta \delta_k}{\lambda_\alpha + \lambda_\beta + \lambda_\gamma + \lambda_\delta}.$$

The complete hierarchical scoring pipeline, including part-level weighting within each dimension, component-level multi-dimensional aggregation, and dish-level component weighting and penalties, is described in Appendix B.5.

We use the accumulated action execution time over the whole task as the main execution-efficiency metric, because it better reflects the agent's temporal scheduling ability in the environment. Auxiliary planning costs (API time and total token consumption) are detailed in Appendix Table 8.

4 Experiments and Analysis: Agent Performance in Physical Interaction

This section is organized around two complementary goals. First, we present the main benchmark results to quantify the overall performance of current agent frameworks on CookBench. Second, we conduct problem-driven analysis experiments to diagnose the underlying causes of their failures, revealing the fundamental capability gaps that emerge during the digital-to-physical transition.

4.1 Experimental Settings

Agent Harness Engineering: **Open Loop** (strict no-feedback execution), **ReAct** (short-horizon reactive prompting (Yao et al., 2023)), **Plan-and-Execute (PE)** (explicit rolling-horizon planning, following the broader planner–executor and feedback-aware embodied planning line (Huang et al., 2022; Ahn et al., 2022; Wang et al., 2023)), **Reflexion** (reflection-augmented correction (Shinn et al., 2023)), and **Code-as-Policy (CaP)** (programmatic code generation (Liang et al., 2022)). For this framework comparison, we fix the backbone model to GPT-5.4 (OpenAI, 2026).

Multi-modal Models: GPT-5.4, Gemini-3.1-Pro (Google, 2026), Kimi-k2.6 (Moonshot AI, 2026), Qwen3.6-Plus, and Qwen3.6-35B (Qwen Team, 2026). We deploy five models using *CaP*, the strongest-performing framework from our initial evaluation.

Human Performance: we recruited 20 volunteers to collectively complete all 133 single-dish recipes once.

To keep episodes comparable and prevent unbounded recovery loops, we enforce a strict 200-step submission cutoff. Each agent setting is run once on every dish. Additional implementation

details and runtime configurations are provided in Appendix B.1 and Appendix B.2.

4.2 Main Results: The Reality Drop in Physical Interaction

The Macro Reality Gap. The main benchmark results reveal a striking reality gap: while modern agents excel in digital environments, as shown in Table 2. Their performance drops precipitously when subjected to physical cooking tasks. Compared to the human upper bound of ∼97%, the best-performing agent framework achieves an average score of only 8.35%. This collective failure powerfully validates that semantic correctness does not equate to physical executability.

Agent Framework Comparison. Controlling for the underlying model (GPT-5.4), we observe how different control flows impact physical adaptability. Looking at the overall scores, performance ranks as *CaP* > *Reflexion* > *PE* > *ReAct*. First, comparing planning horizons, *PE* (explicit 10-step chunking) achieves a higher quality score (5.28%) and the shortest execution time (25.37 minutes) compared to *ReAct* (4.12%, 3-step short-horizon). This indicates that while *ReAct*'s high-frequency reactive planning is locally responsive, it easily loses track of global goals and traps the agent in deadlocks. *PE* provides better global structure, though its rigid planner-executor interface remains brittle to single-step grounding failures. Second, examining error recovery, *Reflexion* adds failure-triggered textual reflection to the *PE* base. However, the performance improvement is surprisingly marginal (from 5.28% to 6.05%). This limited gain exposes the boundary of text-based correction: pure textual reflection struggles to fundamentally resolve physical state deadlocks (e.g., a full container), as agents tend to propose parameter tweaks rather than executing state-altering physical repairs. Finally, *CaP* achieves the highest average score (8.35%). Instead of relying entirely on textual history, its programmatic control flow (e.g., variable binding, loop guards) acts as an externalized persistent memory. This structural advantage partially mitigates "embodied amnesia," allowing CaP to stably push forward high-value culinary operations despite execution leaks. Furthermore, the complete failure (0%) of the *Open Loop* baseline proves that one-shot semantic planning without closed-loop physical feedback is fundamentally unviable in stochastic environments.

Multimodal Comparison and Task Complexity. Evaluating diverse multimodal VLMs under the *CaP* framework reveals stark disparities in foundational physical cognition across different task structures. Overall, *GPT-5.4* establishes the absolute state-of-the-art upper bound (8.35% average), while *Gemini-3.1-Pro* emerges as the strongest alternative (6.77%). Crucially, performance varies drastically across dish categories, exposing heavily skewed physical commonsense among current models. Tasks with relatively linear, pipeline-style workflows (e.g., Pasta-Assembly, Soup-Stew) allow most models to make non-zero progress; for instance, the edge-deployable *Qwen3.6-35B* achieves a remarkable 12.73% on Pasta-Assembly. In stark contrast, tasks demanding intensive multi-component synchronization, frequent mode switching, and delayed state convergence (e.g., Grill-Combo, Salad-Cold) trigger massive failure spikes. On Salad-Cold, *Qwen3.6-35B* collapses to near zero (0.06%), whereas *Gemini-3.1-Pro* maintains partial success, demonstrating superior complex state management. This stark category split proves that the true difficulty of CookBench lies not merely in its long step horizons, but in the intricate structural complexity of embodied physical interactions.

4.3 Analysis: Unveiling the Digital-to-Physical Capability Gap

To diagnose where agents break the physical interaction loop, we analyze the natural cascade of failures: (Section 4.3.1) from translating semantic intents into physical actions, (Section 4.3.2) to processing environmental friction during failures, and (Section 4.3.3) ultimately to maintaining dynamic states over extended horizons.

4.3.1 Leaky Execution Chain: Where Does Semantic Planning Disconnect from Physical Reality?

To diagnose these breakdowns, we classify failures along a semantic-to-physical spectrum. Strikingly, abstract planning flaws (*Schema*, *Binding*) represent a distinct minority. Instead, execution is heavily dominated by physically grounded errors—specifically *Post-State* and *Mode Management* failures—confirming that agents' plausible semantic intentions are passively rejected by the 3D physics engine. This deficit in execution persistence plagues all control flows, albeit differently. While *ReAct* traps agents in local post-state dead-

Method		Grill-Single 22 dishes		Grill-Combo 24 dishes		Pan-Stir 23 dishes		Soup-Stew 25 dishes		Pasta-Assembly 23 dishes		Salad-Cold 16 dishes		Avg. 133 dishes	
Agent	**Model**	**Score % ↑**	**Time min ↓**	**Score % ↑**	**Time min ↓**	**Score % ↑**	**Time min ↓**	**Score % ↑**	**Time min ↓**	**Score % ↑**	**Time min ↓**	**Score % ↑**	**Time min ↓**	**Score % ↑**	**Time min ↓**
Agent Framework Comparison (Fixing Model to GPT-5.4)															
Open Loop	GPT-5.4	0.00	—	0.00	—	0.00	—	0.00	—	0.00	—	0.00	—	0.00	—
ReAct	GPT-5.4	4.21	36.85	2.05	34.95	2.76	37.93	2.66	32.87	9.86	42.83	3.11	26.97	4.12	35.61
Reflexion	GPT-5.4	3.99	29.83	4.61	37.68	**7.08**	37.15	6.63	31.23	9.09	33.31	4.30	33.68	6.05	33.82
PE	GPT-5.4	8.01	**25.44**	**4.73**	**25.97**	5.56	**25.37**	3.90	**24.97**	5.72	**25.80**	3.49	**24.78**	5.28	**25.37**
CaP	GPT-5.4	8.99	33.29	4.19	33.04	6.97	34.39	**9.02**	35.52	**13.72**	38.91	**6.91**	35.40	**8.35**	35.05
Multimodal Model Comparison (Fixing Framework to CaP)															
CaP	GPT-5.4	8.99	33.29	4.19	33.04	6.97	34.39	**9.02**	35.52	**13.72**	38.91	**6.91**	35.40	**8.35**	35.05
CaP	gemini-3.1-pro	**11.73**	47.80	1.96	55.40	6.63	57.88	7.85	49.70	8.27	59.52	3.50	55.46	6.77	54.22
CaP	Kimi-k2.6	1.36	61.44	2.04	60.55	1.79	49.92	5.27	70.30	10.75	55.68	3.13	46.24	4.13	58.22
CaP	Qwen3.6-Plus	3.38	45.55	4.43	53.49	3.50	51.03	4.46	50.14	5.87	58.15	3.16	55.19	4.20	52.14
CaP	Qwen3.6-35B	3.06	36.64	2.34	43.65	2.89	40.63	1.07	57.07	12.73	40.64	0.06	42.46	3.84	43.67
Human Performance Reference															
Human		97.63	6.64	97.47	7.66	96.10	7.90	96.74	7.23	96.00	7.74	96.10	6.61	96.70	7.34

Table 2: Main benchmark results across six CookBench dish categories. Among the nine non-human agent/model settings, bold values denote the best performance and underlined values denote the second-best performance.

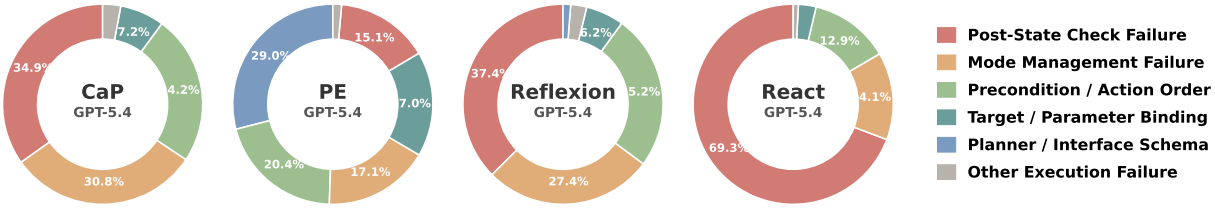

Figure 4: Phenomenon-level failure patterns across four closed-loop GPT-5.4 agent frameworks. *Schema* denotes invalid action formats, unsupported action names, or API-interface mismatches; *Binding* denotes valid action types with incorrect target objects, instance IDs, or parameters; *Precondition/Order* denotes missing physical prerequisites or invalid procedural ordering; *Mode* denotes violations of required body or interaction modes; *Post-state* denotes actions that pass initial checks but fail to realize the intended environment state transition; *Other* groups residual uncategorized failures.

locks (69.3%) and *PE* struggles with brittle schema contracts (29.0%), *Reflexion* proves that patching syntax with textual reflection merely inflates downstream physical obstructions. Even *CaP*, despite using programmatic variables as working memory, suffers from state drift between internal logic and external physical reality. Ultimately, these profiles demonstrate that without robust physical grounding, any high-level control flow inevitably collapses at the execution layer (See Appendix E.2 for detailed micro-mechanisms and framework-specific characteristics)

4.3.2 Closed-Loop Repair: Can Feedback Be Translated into Physical Recovery?

To assess whether agents possess effective **closed-loop repair** capabilities, we analyze two auditable failure signals following execution feedback and replanning. As shown in Figure 5, both signals remain significantly high across all settings (detailed metric construction and full reading are provided in Appendix E.1). The most severe negative case is *CaP@Qwen3.6-35B*, exhibiting a 68.3% blocked retry rate and an 86.0% post-replan error rate. This suggests that while feedback prompts local activity, it rarely translates into valid physical recovery. Qualitative logs expose the root cause: agents incorrectly interpret physical constraints (e.g., occupied hands or hot food that cannot be directly picked up by hand) as digital parameter errors. Rather than performing state-altering physical interventions (e.g., putting down the held object or transferring hot ingredients through a container), they attempt to resolve the issue by shifting coordinates or reissuing identical semantic intents. Ultimately, this highlights a severe deficit in **physical feedback**

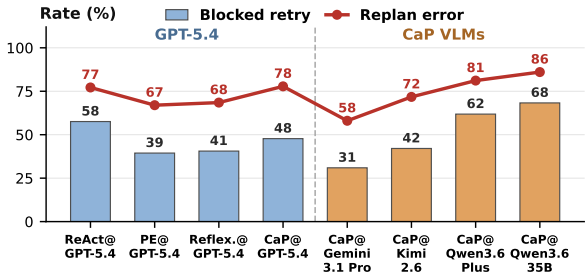

Figure 5: Failure signals for closed-loop repair. Bars (*Blocked Retry Rate*) denote the percentage of failed steps where the agent repeats the exact same failed action within five steps. The line (*Replan Error Rate*) shows the percentage of replans failing to prevent another error within the next five steps. Both metrics reveal a widespread inability to translate feedback into successful physical recovery.

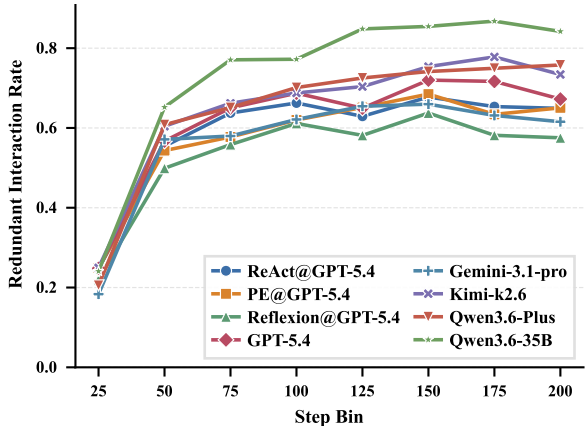

Figure 6: Redundant interaction rate over the 200-step execution horizon. The x-axis marks the end of each 25-step bin, and each line denotes one agent–model setting. Rising late-stage redundancy indicates that agents increasingly spend actions on repeated successful operations, repeated failed attempts, or local repair loops instead of advancing new cooking states.

comprehension: genuine embodied error recovery requires converting textual diagnostics into physical state transitions, not just resuming execution after a failure.

4.3.3 The Horizon Effect: Does Embodied Working Memory Decay Over Time?

Real-world cooking requires persistent state maintenance across many physical transformations. We probe two more direct symptoms of working-memory decay: whether unresolved physical states remain suspended for many steps before triggering a conflict, and whether agents spend more of the late horizon repeating already attempted interactions. Figures 6 and 7 isolate these two mechanisms.

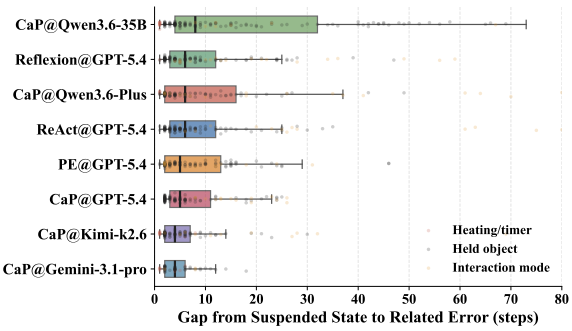

Figure 7: This chart tracks the temporal gap (in steps) between initiating a continuous physical state (e.g., holding an object, turning on a heater, or entering a specific mode) and subsequently triggering a state-conflict error. The substantial delays—often spanning dozens of steps—demonstrate that these failures stem from a degraded working memory of unmaintained states.

Figure 7 shows that many embodied-state failures are delayed rather than immediate: an agent may leave a hand, interaction mode, or heating/timer state unresolved for many steps before a later conflict exposes the mistake. Together with the rising redundancy in Figure 6, this indicates that long-horizon failures are not merely isolated local errors, but symptoms of degraded state maintenance over time. We identify this as **embodied state amnesia**. This progressive amnesia proves that agents cannot simply "look back" at past actions; they require a dedicated, structurally updated working memory that actively tracks continuous 3D environmental changes and resolves latent state conflicts.

5 Conclusion

This paper introduces CookBench to expose the critical gap between digital semantic reasoning and physically grounded execution. Taken together, our analysis experiments show that current agents mainly lack three core capabilities needed for digital-to-physical transfer: **physical grounding** for mapping semantic intentions to valid affordances and state transitions, **closed-loop physical feedback understanding** for turning diagnostic messages into concrete repairs, and **embodied working memory** for maintaining dynamic body, object, progress, and temporal states. CookBench provides the community with a clear diagnostic roadmap for developing embodied agents.

Limitations

While CookBench provides a comprehensive testbed for evaluating digital-to-physical transfer, our design choices introduce certain boundaries:

- **Abstraction of Low-Level Motor Control:** To focus on long-horizon semantic planning, CookBench abstracts low-level robotic kinematics into high-level skills. Consequently, it does not evaluate fine-grained motor control or contact-rich continuous manipulation required in real-world robotics.
- **Domain Specificity:** CookBench is strictly instantiated within kitchen environments. While an exceptional stress-test, the identified capability gaps may not entirely map to other embodied domains with distinct physical dynamics (e.g., outdoor navigation).
- **Computational and Temporal Cost of Evaluation:** The extremely long-horizon nature of the tasks and the necessity of high-fidelity closed-loop simulation make evaluations highly resource-intensive. This high inference cost may restrict massive-scale evaluations or extensive Reinforcement Learning (RL) training loops.

Data and Code Availability. The CookBench dataset and EPM framework will be made publicly available under the MIT license.

AI Assistance Disclosure

AI assistants (Google Gemini, OpenAI ChatGPT) were utilized for writing and coding assistance during the preparation of this manuscript. The human authors developed all scientific ideas, verified all analyses, and take full responsibility for the integrity and originality of the final work.

References

Michael Ahn, Anthony Brohan, Noah Brown, Yevgen Chebotar, Omar Cortes, Byron David, Chelsea Finn, Chuyuan Fu, Keerthana Gopalakrishnan, Karol Hausman, and 1 others. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*.

Mario Bollini, Stefanie Tellex, Tyler Thompson, Nicholas Roy, and Daniela Rus. 2013. Interpreting and executing recipes with a cooking robot. In *Experimental Robotics: The 13th International Symposium on Experimental Robotics*, pages 481–495. Springer.

Micah Carroll, Rohin Shah, Mark K Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca Dragan. 2019. On the utility of learning about humans for human-ai coordination. *Advances in neural information processing systems*, 32.

Matt Deitke, Eli Vander Bilt, Alvaro Herrasti, Luca Weihs, Jordi Salvador, Kiana Ehsani, Winson Han, Eric Kolve, Ali Farhadi, Aniruddha Kembhavi, and 1 others. 2022. Procthor: large-scale embodied ai using procedural generation. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, pages 5982–5994.

Gonzalo Gonzalez-Pumariega, Leong Su Yean, Neha Sunkara, and Sanjiban Choudhury. 2025. Robotouille: An asynchronous planning benchmark for llm agents. *arXiv preprint arXiv:2502.05227*.

Google. 2026. Gemini 3.1 pro: Announcing our latest gemini ai model. https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/. Accessed: 2026-05-23.

John K Haas. 2014. A history of the unity game engine.

Wenlong Huang, Fei Xia, Ted Xiao, Harris Chan, Jacky Liang, Pete Florence, Andy Zeng, Jonathan Tompson, Igor Mordatch, Yevgen Chebotar, and 1 others. 2022. Inner monologue: Embodied reasoning through planning with language models. *arXiv preprint arXiv:2207.05608*.

Anastasiia Ivanova, Eva Bakaeva, Zoya Volovikova, Alexey K Kovalev, and Aleksandr I Panov. 2025. Ambik: Dataset of ambiguous tasks in kitchen environment. *arXiv preprint arXiv:2506.04089*.

Emily Jin, Zhuoyi Huang, Jan-Philipp Franken, Weiyu Liu, Hannah Cha, Erik Brockbank, Sarah Wu, Ruohan Zhang, Jiajun Wu, and Tobias Gerstenberg. 2024. Marple: A benchmark for long-horizon inference. *arXiv preprint arXiv:2410.01926*.

Naoaki Kanazawa, Kento Kawaharazuka, Yoshiki Obinata, Kei Okada, and Masayuki Inaba. 2024. Real-world cooking robot system from recipes based on food state recognition using foundation models and pddl. *Advanced Robotics*, 38(18):1318–1334.

Taewoong Kim, Cheolhong Min, Byeonghwi Kim, Jinyeon Kim, Wonje Jeung, and Jonghyun Choi. 2024. Realfred: An embodied instruction following benchmark in photo-realistic environments. In *European Conference on Computer Vision*, pages 346–364. Springer.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 2024. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 881–905, Bangkok, Thailand. Association for Computational Linguistics.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. 2022. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*.

Mingxian Lin, Wei Huang, Yitang Li, Chengjie Jiang, Kui Wu, Fangwei Zhong, Shengju Qian, Xin Wang, and Xiaojuan Qi. 2025. Embrace-3k: Embodied reasoning and action in complex environments. *arXiv preprint arXiv:2507.10548*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, and 3 others. 2023. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*.

Angelos Mavrogiannis, Christoforos Mavrogiannis, and Yiannis Aloimonos. 2024. Cook2ltl: Translating cooking recipes to ltl formulae using large language models. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 17679–17686. IEEE.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2023. Gaia: a benchmark for general ai assistants. *Preprint*, arXiv:2311.12983.

Moonshot AI. 2026. Kimi-k2.6 model card. https://huggingface.co/moonshotai/Kimi-K2.6. Accessed: 2026-05-23.

OpenAI. 2026. Introducing gpt-5.4. https://openai.com/index/introducing-gpt-5-4/. Accessed: 2026-05-23.

Aishwarya Padmakumar, Jesse Thomason, Ayush Shrivastava, Patrick Lange, Anjali Narayan-Chen, Spandana Gella, Robinson Piramuthu, Gokhan Tur, and Dilek Hakkani-Tur. 2022. Teach: Task-driven embodied agents that chat. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 2017–2025.

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. 2018. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8494–8502.

Qwen Team. 2026. Visual understanding models. https://docs.qwencloud.com/developer-guides/getting-started/vision-models. Accessed: 2026-05-23.

Christopher Rawles, Sarah Clinckemaillie, Yifan Chang, Jonathan Waltz, Gabrielle Lau, Marybeth Fair, Alice Li, William Bishop, Wei Li, Folawiyo Campbell-Ajala, Daniel Toyama, Robert Berry, Divya Tyamagundlu, Timothy Lillicrap, Oriana Riva, and 1 others. 2024. Androidworld: A dynamic benchmarking environment for autonomous agents. *Preprint*, arXiv:2405.14573.

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*.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2020a. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10740–10749.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020b. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*.

Xinyu Jessica Wang, Haoyue Bai, Yiyou Sun, Haorui Wang, Shuibai Zhang, Wenjie Hu, Mya Schroder, Bilge Mutlu, Dawn Song, and Robert D Nowak. 2026. The long-horizon task mirage? diagnosing where and why agentic systems break. *Preprint*, arXiv:2604.11978.

Zihao Wang, Shaofei Cai, Anji Chen, Xiaojian Ma, Yitao Liang, and Yizhi Liu. 2023. Describe, explain, plan and select: Interactive planning with large language models enables open-world multi-task agents. *arXiv preprint arXiv:2302.01560*.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. 2025. Browsecomp: A simple yet challenging benchmark for browsing agents. *arXiv preprint arXiv:2504.12516*.

Zhenyu Wu, Ziwei Wang, Xiuwei Xu, Jiwen Lu, and Haibin Yan. 2023. Embodied task planning with large language models. *arXiv preprint arXiv:2307.01848*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Preprint*, arXiv:2404.07972.

Rui Yang, Hanyang Chen, Junyu Zhang, Mark Zhao, Cheng Qian, Kangrui Wang, Qineng Wang, Teja Venkat Koripella, Marziyeh Movahedi, Manling Li, and 1 others. 2025. Embodiedbench: Comprehensive benchmarking multi-modal large language models for vision-driven embodied agents. *arXiv preprint arXiv:2502.09560*.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2025. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*.

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

Ori Yoran, Samuel Joseph Amouyal, Chaitanya Malaviya, Ben Bogin, Ofir Press, and Jonathan Berant. 2024. Assistantbench: Can web agents solve realistic and time-consuming tasks? *arXiv preprint arXiv:2407.15711*.

Lingfeng Zhang, Yuening Wang, Hongjian Gu, Atia Hamidizadeh, Zhanguang Zhang, Yuecheng Liu, Yutong Wang, David Gamaliel Arcos Bravo, Junyi Dong, Shunbo Zhou, and 1 others. 2025a. Et-plan-bench: Embodied task-level planning benchmark towards spatial-temporal cognition with foundation models. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 21566–21573. IEEE.

Shiqi Zhang, Xinbei Ma, Yunqing Xu, Zouying Cao, Pengrui Lu, Haobo Yuan, Tiancheng Shen, Zhuosheng Zhang, Hai Zhao, and Ming-Hsuan Yang. 2025b. Paracook: On time-efficient planning for multi-agent systems. *arXiv preprint arXiv:2510.11608*.

A Detailed Architecture of the Embodied Planning Machine (EPM)

To bridge the persistent gap between digital semantic reasoning and physical execution, we design the *Embodied Planning Machine* (EPM) as a biomimetic cognitive substrate. EPM standardizes the interaction between agents and the CookBench environment through a decoupled architecture consisting of four primary modules: the Brain, the Cerebellum, the Memory System, and the World Adapter.

The Brain (VLM for Semantic Reasoning). The Brain serves as the central decision-making hub. Driven by modern Vision-Language Models (VLMs), it is responsible for high-level semantic planning, intention recognition, and error reflection. Instead of consuming low-level raw environment states directly, it operates on structured prompts assembled by the Memory System, which summarize the agent state, scene context, task progress, and recent execution feedback.

The Cerebellum (Execution Interface and Skill API). The Cerebellum serves as the strict execution engine for the structured action sequences generated by the Brain. It bridges the digital-to-physical gap by exposing an API library of 85 primitive actions (e.g., navigation, pickup) and 13 programmatic skills (e.g., `auto_cut`, `auto_pour`). Rather than unpacking vague high-level intents, the Cerebellum expects a precise sequence of action calls from the Brain. Its primary role is to faithfully execute these discrete commands within the environment, translating the API invocations into the underlying continuous physical dynamics of the simulator.

The Embodied Memory System (Persistent State Management). A critical bottleneck in embodied tasks is the decay of working memory over time. To combat *Embodied State Amnesia*, EPM features a structured Memory System that persistently tracks the agent's and the environment's states across five dimensions:

- **Action Memory:** Stores API definitions, schemas, and usage templates to help the Brain generate executable syntax.
- **Body Memory:** Tracks the agent's kinematic state, such as currently held items, posture, and active interaction modes, and enforces interaction rules such as empty-hand constraints.
- **Spatial Memory:** Maintains a semantic 3D map and topological relationships between objects and receptacles.
- **Short-Term Memory:** Archives the interaction history.
- **Long-Term & Strategy Memory:** Tracks current task progress against recipe milestones and records immediate failure reflections from closed-loop feedback, and stores reusable optimization heuristics from past executions.

The World Adapter (Perception and Feedback Interface). The World Adapter acts as the translation layer between EPM and the CookBench Unity3D simulator. For perception, it aggregates multimodal inputs, including RGB screenshots and structured object lists such as bounding boxes and distances. For execution feedback, it catches raw simulator signals, including overflow, collision rejection, and state conflicts, and translates them into actionable textual diagnostics. This enables the Brain to reason over physical constraints and perform grounded replanning, closing the execution loop.

A.1 Environment Encapsulation and Skill Abstraction

A key engineering step in CookBench is converting a human-centered cooking simulator into an agent-facing test platform. Directly providing only RGB screenshots makes the task dominated by basic localization and navigation failures: models often cannot reliably determine where an object is, which instance is being targeted, or whether an appliance, door, drawer, or container is currently open, closed, reachable, or interactable. To avoid evaluating low-level perception bottlenecks rather than long-horizon planning, the world adaptor encapsulates structured environment metadata, including object names, live instance IDs, 3D positions, object attributes, containment relations, and door/drawer open states. As shown in Figure 8, the 3D spatial map further uses color-coded object categories to expose the kitchen layout. Tables 3 and 4 summarize the low-level action space and the reusable high-level skills exposed through this interface. This metadata layer does not solve the cooking task; rather, it standardizes the observation

interface so that benchmark failures more directly reflect multimodal planning, state tracking, and physical feedback use.

On top of this metadata layer, CookBench exposes a mid-level control interface consisting of 85 atomic actions and 13 reusable skills. The motivation is similar: when agents are restricted to raw keyboard/mouse primitives or very low-level actions, they frequently fail even at basic navigation and alignment, which obscures the higher-level planning questions CookBench is designed to evaluate. The skill layer abstracts recurring physical routines such as navigation, cutting, pouring, sprinkling, mixing, ordering, purchasing, and submission, while still preserving physical preconditions, body-state constraints, action feedback, and failure modes.

B Additional Experimental Details

This appendix provides additional experimental details that are omitted from the main paper due to space limits.

B.1 Agent Framework Implementation Details

To support controlled comparison across heterogeneous agent designs in Section 4, we instantiate five representative agent frameworks through the EPM framework. Each method is lightly adapted to the same embodied action, feedback, and memory protocol. The comparison is mechanism-controlled because all methods share the same simulator access, action/skill API, execution feedback channel, and memory-store backbone, differing primarily in their control flow. In our pilot trials, directly applying the original digital-agent variants without EPM adaptation resulted in zero-score trajectories, mainly because those agents lacked stable interfaces for embodied state, physical feedback, and long-horizon recovery.

Most closed-loop variants can be viewed as modifications of a Plan-and-Execute (PE)-style outer loop: plan a short horizon, execute step by step, consume structured feedback, and replan when execution fails or the current plan is exhausted. The specific implementations are detailed below, with their enabled capabilities summarized in Table 5.

- **Open Loop**: This serves as the strict no-feedback lower bound. It requests a single multi-step plan once at the beginning of the episode, renumbers the returned steps, and then executes them sequentially. The executor deliberately passes an empty feedback string to the planner, ignores all later execution results, advances to the next planned step even after failure, and never requests another normal plan. If the one-shot plan is exhausted before the episode ends, the baseline does not replan; instead, it immediately arms the shared force-submission supervisor described below. We therefore mark its conceptual planning horizon as *unlimited*: the agent is allowed to propose the whole task once, but it has no reactive recovery ability.

- **ReAct**: Rather than a purely one-action textual agent, our instantiation is a short-horizon interleaved closed-loop variant with a chunk size of 3. Each planning round receives the current observation, recent memory, available tools, and execution feedback, then emits a structured three-step suffix. The executor keeps following the suffix while steps succeed. Once any step fails, the remaining suffix is discarded, the failure feedback is fed into the next prompt, and a new short suffix is generated. This realizes the observe–think/plan–act loop while keeping the same structured action schema as the other methods.

- **Plan-and-Execute (PE)**: This framework uses explicit rolling-horizon chunk planning. It generates 10-step plans, executes them sequentially, and triggers replanning only when the current chunk fails or is completely exhausted. On success, the executor advances the current index in the externally stored plan; on failure, it clears the entire pending plan and forces the next call to generate a fresh plan conditioned on the failure feedback and recent trace. PE is the reference closed-loop executor because its plan state is maintained outside the language model rather than hidden inside a generated program, making it a stable baseline for long-horizon embodied recovery.

- **Reflexion**: This builds upon the PE-style base executor, but adapts the original episode-level reflection mechanism to embodied long-horizon execution. In the original setting, the agent typically reflects after several full episode attempts. This is impractical in CookBench because each episode can consume up to 200 physical actions, making repeated full-

Figure 8: Environment metadata encapsulation for converting a human-centered simulator into an agent test platform. EPM exposes structured object names, positions, attributes, open/closed states, and a color-coded 3D spatial map. This design prevents the benchmark from being dominated by basic object localization and navigation errors caused by screenshot-only perception, and instead focuses evaluation on multimodal agent planning under physical constraints.

episode failures costly in time and model budget; moreover, a failed long cooking episode often entangles many local causes, so the final summary provides limited actionable repair information. We therefore use a step-level reflection trigger: when three errors occur within a rolling window of eight recent execution attempts, and at least eight executed steps have elapsed since the previous reflection, the agent reflects immediately instead of waiting for the whole episode to fail. Periodic reflection is disabled, and loop-triggered reflection is disabled in the reported configuration; the active trigger is failure density in the recent execution window. The reflection prompt summarizes the failed actions, structured execution diagnostics, recent short-term memory, current recipe progress, and likely corrective strategies. The generated reflection is stored in a sliding memory window of three recent reflections and injected into the next planning prompt together with a compact progress restatement. The action API, low-level skills, chunk size, and execution feedback channel remain the same as PE, so the PE–Reflexion comparison isolates the effect of more timely failure-triggered textual reflection rather than changing the executor itself.

- **Code-as-Policy (CaP)**: The original CaP formulation generates a Python policy program and executes it line by line, with local variables implicitly maintaining stage, item IDs, pot or container IDs, subgoals, recent failures, completed steps, and remaining action lists. This design is natural when the generated program can run without interruption, but it is unstable in CookBench because long-horizon cooking frequently requires replanning, local repair, and state recovery. Compared with PE, whose plan state is externally maintained and revised over the observed execution trace, the original CaP places much of the state bookkeeping inside program-local variables. Once execution fails and the policy must be regenerated, maintaining consistency between the new code, the already-executed variable environment, the physical world state, and the remaining recipe constraints becomes cumbersome and error-prone. The program format also introduces extra planning rules and variable-maintenance constraints on top of

Category	Action names
Navigate	move_backward, move_backward_left, move_backward_right, move_forward, move_forward_left, move_forward_right, move_left, move_right
Look	look_down, look_left, look_right, look_up
PickupObject	pick_less, pick_more, pick_up, pick_up_into_the_container
PutObject	put_down
RepairObject	repair
BlenderMode	blender_downward, blender_upward, enter_mixing_mode, exit_mixing_mode, toggle_off_blender, toggle_on_blender
Posture	kneel_down, stand_up
PutObject-DropThrowRelease	discard_into_garbage_bin, throw_away
Timing	countdown_time, wait_for_timer
DoorInteraction	close_drawer, close_door, open_drawer, open_door
Cut-object	enter_cutting_mode, exit_cutting_mode, knife_cut_down, exit_current_interaction_mode
Tongs-interaction	pick_up_with_tongs, release_with_tongs
Ladle-interaction	get_liquid_in_Ladle, get_liquid_out_Ladle
Pipette-interaction	change_content_5_or_10ml, pour_out_from_pipette, pour_in_pipette
SpatialControl	horizontal_movement, rotate_along_the_Yaxis, rotate_clockwise_along_the_Yaxis, rotate_clockwise_along_the_Zaxis, rotate_counterclockwise_along_the_Yaxis, rotate_counterclockwise_along_the_Zaxis, spatial_rotation, switch_to_advanced_control_mode, switch_to_normal_control_mode, vertical_movement
Liquid-pourout	enter_pouring_mode, exit_pouring_mode, bottleneck_move_downward, bottleneck_move_upward
Faucet-container-interaction	change_faucet_pipe_rotation, enter_the_faucet_filling_mode, exit_the_faucet_filling_mode, move_faucet_container, open_faucet_left_handle_in_faucet_filling_mode, open_faucet_right_handle_in_faucet_filling_mode, screw_faucet_left_handle, screw_faucet_right_handle
Bottle-interaction	screw_bottle, unscrew_bottle
DeepFryer-interaction	togggle_add_oil_into_Deep_fryer, toggle_pour_oil_out_Deep_fryer
ToggleOn/Off	toggle_off_switch, toggle_on_switch
Skewer-interaction	skewer_hang_on_string, skewer_remove_the_ingredients
Spices-sprinkle	adjust_sprinkle_weight, enter_spices_sprinkle_mode, exit_spices_sprinkle_mode
Spatula-interaction	enter_the_flipping_mode, exit_the_flipping_mode, flip_over, pick_up_with_spatula
Food-processor	toggle_off_food_processor, toggle_on_food_processor
Egg-crack	crack_open_the_egg

Table 3: Atomic action names exposed by CookBench. The 85 actions are organized into the 25 interaction categories defined by the action taxonomy.

the already difficult embodied task. Therefore, our adapted CaP does not execute the generated Python-like code as a fully autonomous script. Instead, it asks the model to use Python-like code as a structured chain-of-thought and policy representation, statically extracts the yielded action/skill queue, and executes the resulting embodied action sequence through the same interface as the other agents. Each generated chunk is capped at 10 steps. EPM keeps persistent state outside the generated program in a runtime state record and reinjects resume information, object bindings, ordered-dish state, recent failures, last yielded action, postconditions, and recent execution feedback when regenerating the policy. CaP also includes safeguards for repeated scene queries, repeated successful identical steps, and local repair failures: such patterns clear the current program, preserve the observed state, and trigger suffix regeneration with an explicit reason such as plan exhaustion, query loop, repeated-step loop, or repair after failure. This simple adaptation preserves CaP's useful programmatic structure while avoiding fragile line-by-line program execution and local-variable state drift.

B.1.1 Force-Submission Policy

CookBench episodes use a shared force-submission supervisor to prevent all agents from spending the entire horizon in unbounded local recovery loops. The supervisor is not an additional planner and is applied uniformly across methods. In the reported configuration, normal execution can use up to 200 steps before force submission is armed. Once the executed step index reaches this threshold, EPM records a deadline 50 steps later, yielding the 250-step hard cap. If the strict Open Loop baseline consumes its one-shot plan before the threshold, EPM also arms force submission immediately with the reason `open_loop_plan_exhausted`. The force-submission status, remaining step budget, deadline, and consecutive failure count are written into agent state so the prompt can treat the deadline as a first-class decision signal.

After force submission becomes active, EPM restricts the available interface to a minimal allowlist of finalization primitives. These specifically include: basic manipulation (e.g., `pick_up`, `put_down`, `throw_away`), mobility and posture adjustments (`auto_navigation`, `kneel_down`, `stand_up`), state transitions (`enter_pouring_mode`, `auto_pour`,

Skill	Role
auto_cut	Cut a specified ingredient for a target number of cuts and exit cutting mode after completion.
auto_flip	Flip a target food item with a spatula and place it onto the specified surface or container.
auto_mix	Mix the contents of a target container with the blender and restore the normal holding state.
auto_navigation	Navigate to a target object and align the agent's field of view to center precisely on the target.
auto_perception	Return structured visible-scene observations from the current environment state.
auto_pour	Pour a target volume of liquid into a specified container and stop near the requested amount.
auto_sprinkle	Sprinkle seasoning onto a specified target for a requested number of repetitions.
gui_buy_new_item	Navigate to the shop interface and purchase a missing ingredient or tool.
gui_order_dish_via_computer	Start an episode by ordering the target dish through the in-game computer interface.
gui_submit_dish_via_checkout_stand	Submit the final dish at the checkout stand and collect score/feedback.
query_scene_objects	Retrieve live scene instances, including instance IDs, containers, distances, and visibility.
query_product_catalog	Query static product metadata such as available items, forms, and reference attributes.
query_tool_manual	Query cookware/tool usage metadata from the tool manual.

Table 4: Reusable high-level skills exposed by EPM. These skills abstract repeated physical routines while preserving simulator feedback and failure states.

`exit_current_interaction_mode`), and the terminal execution (`gui_submit_dish_via_checkout_stand`). Tool schemas and tool-calling manifests are dynamically rebuilt from this restricted allowlist, and the normal agent pipeline is bypassed by a deterministic force-submit decision routine. This design makes the late-stage behavior comparable across agents: they can still recover, transfer, and submit plausible final containers, but they cannot start unrelated exploration or open-ended recipe execution after the deadline begins.

The force-submit routine is a state machine over the current held object, interaction mode, candidate serving containers, and container probe results. It first exits unsafe interaction modes unless the current stage intentionally requires pouring. If the agent is holding a directly submittable food item, it submits it immediately; if it is holding a non-container object, it throws the object away; if it is holding a valid container selected for submission, it navigates to the checkout stand and submits. When no object is held, the routine probes candidate vessels, records whether each probe is nonempty, counts observed contents, checks name matches and pickup signals, and selects the best confirmed nonempty submission vessel. If a nonempty transfer-source vessel is found while the preferred final vessel such as the Big Pot is empty, the routine navigates to the source, picks it up, enters pouring mode, transfers contents into the destination, drops the source safely, and then submits the destination. Repeated force-submit failures invalidate the current candidate and trigger recovery; if the deadline is overdue and the number of consecutive force-submit failures reaches the configured failure limit, the episode is terminated with a fatal force-submit error. Thus, force submission is a bounded finalization policy rather than a hidden scoring shortcut: it only operates through the same embodied action/skill interface and submits containers that the environment makes reachable and observable.

B.2 Experimental Configuration

Table 6 summarizes the main runtime and pipeline settings used in the experiments. Table 7 further lists the agent-specific pipeline settings used in the framework comparison. These values are derived from the shared EPM configuration files in `epm/configs`. Local machine paths and private API keys are excluded.

B.3 Evaluation Protocol

In the main setting, agents receive structured cooking goals rather than relying solely on natural-language interpretation. This decoupled protocol allows the benchmark to isolate embodied execution failures from instruction understanding errors.

B.3.1 Natural-Language Instructions

CookBench also provides natural-language instructions as an optional front-end resource for studying intent recognition before embodied execution.

Agent	Plan horizon	Execution feedback	Failure reflection	Progress summary	Code-style policy
Open Loop	unlimited	×	×	×	×
ReAct	3	✓	×	×	×
PE	10	✓	×	×	×
Reflexion	10	✓	✓	✓	×
CaP	10	✓	×	×	✓

Table 5: Capability comparison of the five adapted agent frameworks. Check marks indicate that the capability is explicitly enabled in our EPM-based implementation.

Setting	Value	Purpose
Task set	133 single-dish recipes	Main benchmark suite used for both agent-framework and model comparisons. All benchmark tasks are instantiated in the **Classic** kitchen scene.
Goal protocol	Oracle-goal structured recipes	Separates embodied planning/execution failures from natural-language instruction understanding noise.
Backbone for framework comparison	GPT-5.4	Controls model capacity when comparing Open Loop, ReAct, PE, Reflexion, and CaP.
Backbones for model comparison	GPT-5.4, Gemini-3.1-pro, Kimi-k2.6, Qwen3.6-Plus, Qwen3.6-35B	Tests whether different VLM backbones transfer better under the same CaP/EPM interface.
Human reference	20 volunteers, 133 recipes completed once collectively	Provides a score and completion-time reference under human cooking behavior.
Primary cutoff	200 steps	Starts the force-submission window to avoid unbounded recovery loops.
Hard runtime cap	250 steps	Maximum configured episode length after the force-submission window.
Short-term memory window	200 steps	Shared runtime memory window for recent interaction traces.
Perception mode	Oracle structured observation with VLM screenshots enabled	Provides stable object/state access while preserving visual grounding requirements through screenshots.
Model decoding	Temperature 0.2, max tokens 128000, timeout 600s, retries 3	Keeps generation relatively deterministic while allowing long planning contexts.

Table 6: Summary of experimental runtime and pipeline settings.

This resource, denoted as CookBench-IR, contains 14,394 bilingual instructions: 7,197 in Chinese and 7,197 in English. The purpose is to evaluate whether a model can map user utterances to the intended target dish list and, when relevant, the correct dish order. In contrast, the main embodied-interaction experiments in this paper use oracle structured goals, so the reported execution scores are not confounded by natural-language instruction parsing errors. We therefore describe these instructions as an auxiliary benchmark resource rather than as a new interface contribution.

The instruction set is constructed at two levels. For single-dish requests, we define three difficulty types. *Direct instructions* explicitly mention the target dish name and mainly test basic named-entity recognition. *Interference instructions* include multiple options, distractors, or conflicting conditions, covering realistic cases such as referential resolution and logical branching; the model must reason over the utterance to identify the final intended dish. *Ambiguous instructions* omit the dish name and describe the request through ingredients, taste, cuisine, dietary effects, or other domain cues, requiring the model to ground fuzzy language into the structured recipe knowledge base.

For multi-dish requests, the construction is balanced across 4-, 5-, and 6-dish scenarios, with approximately 1,482 instances per scenario. We control the mixture of simple and interference-based clauses and distribute dish mentions to reduce shortcut bias. The instructions also cover different ordering patterns, including *batch orders*, where all dishes are specified at once, and *sequential orders*, where dishes are specified incrementally across the utterance. In addition, the evaluation set includes single-turn and multi-turn dialogue forms, different dish repetition patterns, and shuffled 4-, 5-, and 6-dish cases. The Chinese version follows the English construction but is further strengthened

Agent	Pipeline	Plan horizon	Main adaptation
Open Loop	`open_loop`	unlimited	Generates one initial plan and does not consume later execution feedback.
ReAct	`react`	3 steps	Uses a short closed-loop planning suffix and discards the suffix after failure.
PE	`planner_executor`	10 steps	Maintains plan state externally and replans after failure or chunk exhaustion.
Reflexion	`reflexion`	10 steps	Converts episode-level reflection into step-level repair: reflection is triggered after three errors within eight recent execution attempts.
CaP	`code as policy`	10 steps	Uses Python-like code as structured CoT before emitting action sequences, while EPM externally maintains persistent state and object bindings.

Table 7: Agent-pipeline settings used for the framework comparison.

with additional aliases, synonyms, and semantically denser contexts, especially in multi-person dialogue scenarios. For CookBench-IR evaluation, a prediction is counted as correct only when all target dish names and the required order exactly match the ground truth.

B.4 Planning Cost and Token Usage

Table 8 reports auxiliary planning-cost statistics. Unlike the main efficiency metric, which uses action execution time to reduce sensitivity to network fluctuations, this table explicitly records API-side thinking time and language-model token consumption. The token metric is computed as input tokens plus output tokens per run and is reported in millions. Human and Open Loop are omitted because they do not use closed-loop API planning in this comparison.

The framework comparison reveals that planning cost and final embodied performance are only partially aligned. *PE@GPT-5.4* uses the smallest average token budget among the four closed-loop GPT-5.4 agents (2.421M), and it also achieves the fastest end-to-end execution time in the main benchmark (25.37 min). This makes PE the most economical choice when the primary constraint is full embodied runtime or raw planning-token budget. However, its average score (5.28%) still trails both *Reflexion@GPT-5.4* (6.05%) and especially *CaP@GPT-5.4* (8.35%), indicating that lower planning cost alone does not imply better physical task completion. By contrast, *ReAct@GPT-5.4* is expensive on both planning dimensions (16.65 API min, 3.811M tokens) while remaining the weakest GPT-5.4 closed-loop framework in score (4.12%), making it the least attractive point in the cost–quality space. *Reflexion@GPT-5.4* occupies a more favorable middle ground: compared with ReAct, it reduces token cost by about 32% (3.811M $\rightarrow$ 2.581M), slightly reduces API thinking time (16.65 $\rightarrow$ 15.49 min), and simultaneously improves score from 4.12% to 6.05%. This suggests that failure-triggered reflection is substantially more budget-efficient than frequent short-horizon reactive replanning.

The most notable frontier point in the framework comparison is *CaP@GPT-5.4*. It achieves the best score (8.35%) while also recording the lowest average API-side planning time (11.75 min), showing that its program-style reasoning format is highly efficient on the language-model side. Yet its full execution time in the main benchmark is still 35.05 min, much slower than PE's 25.37 min. This gap is important: fast planning does not automatically translate into fast embodied completion, because overall runtime is dominated not only by thinking latency but also by how many physical recovery steps, retries, and interaction detours are induced during execution. Even so, if we jointly consider quality and cost, CaP remains the strongest GPT-5.4 configuration overall. It provides the highest score per API minute, and it also yields the best score per unit of end-to-end runtime among the closed-loop GPT-5.4 agents, meaning that its longer execution traces still convert into more useful task progress than competing frameworks.

The multimodal comparison under a fixed *CaP* controller exposes even sharper efficiency tradeoffs. *CaP@GPT-5.4* remains the clear Pareto frontier: it delivers the highest average score (8.35%), the lowest planning latency (11.75 min), and a moderate token budget (3.351M), while also being the fastest model in full embodied runtime (35.05 min). Among the non-GPT alternatives, *Gemini-3.1-Pro* is the strongest score-wise (6.77%) and uses a sim-

Method		Grill-Single 22 dishes		Grill-Combo 24 dishes		Pan-Stir 23 dishes		Soup-Stew 25 dishes		Pasta-Assembly 23 dishes		Salad-Cold 16 dishes		Avg. 133 dishes	
Agent	**Model**	**API min ↓**	**Tok. M ↓**	**API min ↓**	**Tok. M ↓**	**API min ↓**	**Tok. M ↓**	**API min ↓**	**Tok. M ↓**	**API min ↓**	**Tok. M ↓**	**API min ↓**	**Tok. M ↓**	**API min ↓**	**Tok. M ↓**
Agent Framework Comparison															
ReAct	GPT-5.4	14.14	3.352	18.59	4.088	13.87	3.524	15.82	4.022	15.43	3.645	23.47	4.252	16.65	3.811
Reflexion	GPT-5.4	14.61	2.601	15.59	2.749	16.38	2.569	14.80	2.489	15.89	2.557	15.84	2.504	15.49	2.581
PE	GPT-5.4	20.58	2.405	19.14	2.626	18.68	2.564	17.69	2.316	17.91	2.268	16.97	2.372	18.46	2.421
CaP	GPT-5.4	13.99	3.532	9.17	3.314	7.31	3.049	16.58	3.366	11.48	3.464	12.09	3.415	11.75	3.351
Multimodal Model Comparison															
CaP	gemini-3.1-pro	27.51	3.166	31.57	3.542	26.87	3.291	26.34	3.014	31.79	3.518	28.74	3.402	28.70	3.309
CaP	Kimi-k2.6	19.96	3.323	19.87	3.349	18.79	3.180	17.72	2.864	18.37	3.053	19.30	3.313	18.96	3.168
CaP	Qwen3.6-Plus	34.94	4.193	41.59	4.515	38.33	4.256	34.64	3.974	42.39	4.296	42.93	4.771	38.88	4.307
CaP	Qwen3.6-35B	20.91	5.245	20.67	5.218	21.56	5.620	21.63	5.450	21.61	5.548	22.66	5.793	21.44	5.460

Table 8: Auxiliary planning-cost comparison across six CookBench dish categories. API min denotes average language-model thinking time per run in minutes. Tok. M denotes average input-plus-output token usage per run in millions.

ilar token budget to GPT-5.4 (3.309M vs. 3.351M), but it is much slower in both API planning time (28.70 min) and total execution time (54.22 min). This means Gemini is competitive in quality but not in efficiency. *Kimi-k2.6* has slightly lower planning cost than Gemini (18.96 min, 3.168M tokens), yet its score drops sharply to 4.13% and its total runtime is even longer (58.22 min), so its lower token use does not convert into better task-level efficiency. *Qwen3.6-Plus* is the least favorable latency point: it has the highest API planning time (38.88 min) and a high token budget (4.307M), but its score remains only 4.20%. *Qwen3.6-35B* presents a different failure mode. Its API planning time (21.44 min) is substantially lower than Gemini and Qwen3.6-Plus, but it consumes by far the most tokens (5.460M) while still obtaining only 3.84%, making it the worst token-to-quality tradeoff in the model comparison.

Taken together, Table 8 and Table 2 support three practical conclusions. First, **planning-token efficiency and embodied quality are not identical**: PE minimizes token cost, but CaP@GPT-5.4 converts each unit of model-side computation into higher final score. Second, **planning latency and wall-clock runtime are also not identical**: CaP@GPT-5.4 is the fastest planner but not the fastest executor, whereas PE plans more slowly but finishes sooner overall because its embodied interaction trajectory is shorter and more stable. Third, **high cost is not sufficient for high quality**: Qwen3.6-Plus and Qwen3.6-35B both incur large planning budgets without corresponding score gains, while Gemini-3.1-Pro pays a large latency premium for only a partial recovery in quality. Therefore, auxiliary planning-cost statistics should be interpreted jointly with final score and embodied execution time, rather than as standalone indicators of agent usefulness.

B.5 Scoring Details

CookBench evaluates each submitted dish through the simulator's built-in recipe scoring pipeline. After an agent submits a container as the final dish, the simulator compares the submitted physical state against the target recipe and returns a structured feedback record. This record contains a normalized quality score, a dish-level penalty, component-level scores, and human-readable complaints. Following the benchmark protocol used in the main paper, we retain the simulator-returned *quality* score as the official benchmark target and do not add realization-time scores into the reported benchmark score. Complaint fields are used only for diagnostic analysis.

Simulator feedback structure. The feedback record has three levels. First, the quality block reports the final normalized quality score and the percentage score used in our tables. Second, the component block lists each recipe component with a component weight and four dimension-specific scores: taste, thermal processing, technique, and serving temperature. Each dimension also has a corresponding score weight, so dimensions that are irrelevant to a component do not affect its compo-

nent score. Third, the complaint block records the concrete failure reasons exposed by the simulator, including flavor complaints, technique complaints, temperature complaints, and unwanted products. Technique complaints include messages such as bad cutting, missing products, wrong amount, product wrapping, unhealthy products, unwanted products, or appliance-related mistakes. These complaints are not manually assigned by us; they are generated by the simulator's scoring logic from the submitted dish state.

Official benchmark score. Let $\mathcal{Q} \in [0, 1]$ denote the simulator-returned quality score. CookBench reports the percentage-scaled benchmark score

$$\mathcal{P} = 100\,\mathcal{Q}.$$

Any realization-time signal shown in the game UI is treated as an auxiliary log field rather than part of the official benchmark objective.

Three-level score aggregation. The simulator uses a three-level hierarchy. At the lowest level, each recipe component k contains product parts i with part weights q_i. For each scoring dimension $d \in \{\text{taste}, \text{thermal}, \text{technical}, \text{temperature}\}$, let $s_i^{(d)} \in [0, 1]$ be the derived score of part i. The component's single-dimension score is

$$\text{singleScore}_k^{(d)} = \frac{\sum_{i \in k} q_i\, s_i^{(d)}}{\sum_{i \in k} q_i}.$$

This is the first level of aggregation: multiple product parts are averaged within one dimension according to their part weights. The four dimensions capture different physical requirements. Taste compares submitted flavors against recipe-defined target flavor ranges and is capped by thermal correctness when poor heating invalidates flavor quality. Thermal processing checks whether required heating operations such as frying, baking, boiling, grilling, or related treatments were correctly applied and penalizes unwanted heat traces. Technique checks operation-specific requirements such as cutting, chopping, blending, mixing, wrapping, peeling, or garnishing. Temperature checks whether the final serving state matches recipe expectations such as hot or cool serving.

At the second level, the four dimensions are aggregated inside each component. For compactness, let $c_{k,t}$, $c_{k,h}$, $c_{k,\tau}$, and $c_{k,\theta}$ denote the taste, thermal, technical, and temperature scores of component k, respectively, and let $(w_t, w_h, w_\tau, w_\theta)$ denote their recipe-defined weights. Then

$$\text{CF}_k = \frac{w_t c_{k,t} + w_h c_{k,h} + w_\tau c_{k,\tau} + w_\theta c_{k,\theta}}{w_t + w_h + w_\tau + w_\theta}.$$

At the third level, recipe components are combined by component weights ω_k:

$$\bar{C} = \frac{\sum_k \omega_k\, \text{CF}_k}{\sum_k \omega_k}.$$

The final dish quality score subtracts a dish-level penalty π:

$$\mathcal{Q} = \max\left(0,\ \bar{C} - \pi\right).$$

Here CF_k is the final score of component k, and $\bar{C}$ is the weighted component average of the whole dish. The three weighting levels remain distinct: (q_i) aggregate multiple product parts within one dimension, $(w_t, w_h, w_\tau, w_\theta)$ determine which physical dimensions matter for a component, and (ω_k) determine how much each component contributes to the whole dish.

Dish penalty. The penalty term π is derived from the simulator's dish-level scoring logic and is not simply another component-average term. It penalizes whole-dish mismatches that are not fully captured by component averaging, especially unwanted ingredients and composition-level contamination. For brevity, let S denote wanted product mass, T thermally scored wanted mass, U unwanted product mass, and U_T thermally scored unwanted mass, with $T \le S$ and $U_T \le U$. The simulator uses constants $A = 2.5$ and $B = 10$, together with the helper functions $C_1(x) = \min(\max(x, 0), 1)$ and $E(x) = C_1(1 - (1 - x)^2)$. The penalty is piecewise defined as follows.

If $S \le 0$, then

$$\pi = 0.$$

If $S \ge A \cdot T$, meaning that thermally scored products do not dominate the dish composition, then

$$\pi = \begin{cases} C_1\left(\left(\frac{U}{S}\right)^2\right), & \frac{U}{S} < 0.1, \\ E\left(\frac{U}{S}\right), & \frac{U}{S} \ge 0.1. \end{cases}$$

If $S < A \cdot T$ and $T > 0$, the simulator additionally penalizes non-thermal unwanted residue inside a thermally dominated dish. Let $u = U/S$; when

$S \neq T$, also let $v = (U - U_T)/(S - T)$. Then

$$\pi = \begin{cases} C_1(u^2) & \\ \quad +E\left(\frac{U-U_T}{S}\right) & u < 0.1, \\ E(u) + E(v), & u \geq 0.1,\ S \neq T, \\ C_1\big(2 - (1-u)^2\big), & u \geq 0.1,\ S = T. \end{cases}$$

Two details matter here. First, the simulator does not clamp the sum after addition, so π should be interpreted as a penalty-strength scalar rather than a strict probability-like score in $[0, 1]$. Second, this term explains why a dish can receive partially reasonable component-level scores but still lose substantial quality due to extra products or global composition violations.

Temperature dimension. For the temperature dimension, let E denote the expected serving temperature, R the tolerance range, and z the actual product temperature. The simulator first computes

$$\begin{aligned} T^- &= \mathrm{Clamp}(E - R, 20, 100), \\ T^+ &= \mathrm{Clamp}(E + R, 20, 100). \end{aligned}$$

If the rule flag `checkGoodOrGreater` is true, the product is treated as acceptable as long as it is not colder than the lower bound:

$$s_i^{(\theta)} = \begin{cases} 1, & z > T^-, \\ \frac{z}{E}, & z \leq T^-. \end{cases}$$

Otherwise, the product is expected to lie inside the target interval:

$$s_i^{(\theta)} = \begin{cases} 1, & T^- \leq z \leq T^+, \\ 1 - \frac{z-E}{100-E}, & \text{otherwise}. \end{cases}$$

Technical dimension. For the technical dimension, the simulator compares each product part against a recipe-specific full-score interval $[x_s, y_s]$ and a broader half-score interval $[x_b, y_b]$:

$$s_i^{(\tau)} = \begin{cases} 1, & x_s \leq q_i \leq y_s, \\ 0.5, & x_b \leq q_i \leq y_b,\ q_i \notin [x_s, y_s], \\ 0, & \text{otherwise}. \end{cases}$$

Thermal dimension. For the thermal dimension, the simulator compares the performed heating trace against the recipe-required cooking types. If a product does not participate in the thermal dimension, then

$$s_i^{(h)} = 1.$$

Otherwise, the score is computed as $s_i^{(h)} = C_1(V - P)$, where V is the value contributed by required thermal types and P is the penalty contributed by unwanted thermal types:

$$V := \sum_{j \in \mathcal{R}} \left(\frac{r_j}{\sum_{k \in \mathcal{R}} r_k} \cdot \sigma_j \right), \qquad P := \sum_{g \in \mathcal{U}} \rho_g.$$

Here $\mathcal{R}$ is the set of required heating types, $\mathcal{U}$ is the set of unwanted heating types, r_j is the recipe target magnitude for required type j, σ_j is its achieved score, and ρ_g is the penalty of unwanted type g. At the product-region level, the simulator maintains cumulative thermal intensities $P_{a,g}$ for region a and thermal type g, and converts them into scored values through a thresholded average-value mechanism. For the benchmark interpretation, the main effect is simple: missing a required heating type drives V down, while applying an unwanted heating type increases P.

Taste dimension. The taste dimension compares observed flavor contributions against recipe-specific target flavor profiles. For one flavor channel, let d be the scaled deviation from the target value, M the maximum tolerated deviation for the current quality band, and $(V_{\min}, V_{\max})$ the score range assigned to that band. Denoting the band score by β, the simulator uses

$$\begin{aligned} b &= V_{\max} - \tfrac{|d|}{M}(V_{\max} - V_{\min}), \\ \beta &= \mathrm{Clamp}(b,\ V_{\min},\ V_{\max}). \end{aligned}$$

The resulting taste score is then capped by the thermal score of the same product:

$$s_i^{(t)} \leftarrow \mathrm{Clamp}\Big(s_i^{(t)},\ 0,\ s_i^{(h)}\Big).$$

This cap is important for interpretation: a product with failed thermal handling can lose taste score even if its flavor composition would otherwise be close to the target.

Complaint fields for diagnosis. The simulator's complaint record is used to explain why a submitted dish lost points. Flavor complaints indicate seasoning or taste-profile mismatch. Technique complaints identify product-level operation failures, such as bad cutting, missing product, wrong amount, unwanted product, unhealthy product, product wrapping, or appliance mismatch. Temperature complaints identify serving-temperature or heat-state errors. The unwanted-products list records extra products present in the submitted dish

that do not belong to the target recipe. Together, these complaint fields distinguish whether an agent never attempted a required ingredient, interacted with it but failed to finalize it, used the wrong technique, overheated or underheated it, or added extra products into the final container.

C Additional Benchmark Details

C.1 Task Construction

CookBench recipes are stored as compact semi-structured entries. Each entry specifies a dish identifier, dish name, required ingredients, one or more recipe components, and a reference video identifier. Table 9 shows one representative entry from the single-dish recipe pool.

C.2 Single-Dish Cooking-Style Taxonomy

For analysis, we further group the 133 single-dish tasks into six cooking-style categories. This grouping is not used to define the benchmark itself; rather, it provides a balanced and interpretable lens for reporting category-level results and diagnosing where current models break down. The grouping emphasizes preparation style and workflow structure, while keeping the category sizes reasonably balanced.

Grilled Mains. This category centers on a single grilled or roasted main item without an explicit multi-component platter structure. The main splitting criterion is that the task focuses on one dominant hot main rather than a coordinated "main + side" composition. **Dish IDs:** 1, 11, 13, 15, 27, 32, 33, 48, 52, 53, 55, 64, 98, 109, 115, 117, 121, 122, 124, 126, 127, 138.

Grilled Platters. These tasks present a grilled or roasted main together with sides, sauces, or plated accompaniments. The key criterion is explicit multi-component coordination and final plate assembly rather than single-main completion. **Dish IDs:** 5, 6, 8, 12, 14, 23, 24, 28, 34, 38, 50, 51, 62, 63, 66, 96, 99, 100, 116, 118, 123, 125, 128, 137.

Pan Dishes. This category includes dishes dominated by skillet frying, sauteing, stir-frying, reduction, or other pan-centered hot cooking processes. The main criterion is that the core difficulty lies in local pan-level ordering and short-window thermal control. **Dish IDs:** 17, 18, 25, 26, 31, 36, 37, 44, 60, 61, 65, 92, 93, 94, 95, 101, 102, 103, 104, 107, 108, 110, 111.

Soups & Stews. These tasks are organized around soups, stews, curries, broths, and other pot-based liquid-heavy workflows. The splitting criterion is that liquid management, boiling time, ingredient order, and multi-stage convergence dominate the task complexity. **Dish IDs:** 19, 20, 29, 30, 40, 41, 54, 79, 80, 81, 82, 83, 84, 86, 87, 88, 89, 90, 91, 97, 105, 106, 112, 113, 114.

Assembled Staples. This category includes pasta-centered dishes as well as breakfast, burger, tart, and other staple-like tasks whose difficulty lies in component preparation and final assembly. The main criterion is that the task is structured around a staple base or carrier that must be matched with additional prepared components. **Dish IDs:** 3, 4, 9, 10, 39, 42, 58, 59, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 120, 129, 131.

Cold Dishes. This group contains salads, gazpacho-style cold preparations, and lighter dishes that rely more on cutting, ingredient selection, and cold assembly than on sustained hot cooking. The key criterion is that light preparation and final combination dominate over long heating processes. **Dish IDs:** 2, 7, 16, 21, 22, 35, 43, 45, 46, 47, 49, 56, 57, 85, 119, 130.

The complete ID-to-name mapping for all single-dish tasks is listed in Table 10. Table 11 further provides representative examples from the interactive object vocabulary exposed by the environment.

C.3 Action and State Design

Beyond the action interface, CookBench returns rich, structured execution feedback during interaction. This feedback mechanism shifts the benchmark from a static action-scoring setup to a realistic closed-loop execution environment. The feedback primarily consists of four categories:

- **Precondition checks.** Before an action is simulated, the engine verifies whether physical prerequisites are met, such as ensuring that the agent is holding a knife before triggering `cut`, or that its hands are empty before `pick_up`.
- **Post-state verification.** After execution, the environment checks whether the intended state transition actually occurred, such as verifying whether an object was successfully dropped into a container rather than remaining in hand or falling outside the target area.

Field	Example content
Dish ID	3
Dish name	Lemon Tart
Ingredients	Banana; Orange; Strawberry; Lemon; Tart; Honey; Coconut Milk; Brown Sugar
Recipe component	Tart
Recipe workflow	Take Tart [260g]. Bake for 40s. Add to the Food Processor: Coconut Milk [180ml], Honey [20ml], Brown Sugar [5g]. Blend. Cut: Banana [100g] into Pieces [20g], Lemon [80g] into Pieces [10g], Orange [60g] into Quarters [15g], Strawberry [60g] into Halves [5g]. Transfer into a Tart: Mixture [200ml], Lemon, Banana, Orange, Strawberry. Serve cool.
Video ID	video_001

Table 9: Example recipe entry from the CookBench recipe repository. The recipe text is intentionally semi-structured: it preserves natural cooking instructions while exposing quantities, operations, containers, and serving conditions that agents must ground into executable actions.

- **Measurement and progress feedback.** For continuous tasks, the simulator returns quantitative progress signals, such as the exact volume of liquid poured (`poured_ml`) or the number of cuts completed, helping agents monitor partial completion and resource usage.

- **Retrieval and disambiguation.** When an agent queries the environment or attempts to interact with non-unique objects, such as multiple plates on a table, the engine provides plausible candidates and their `instance_ids`, requiring explicit target disambiguation.

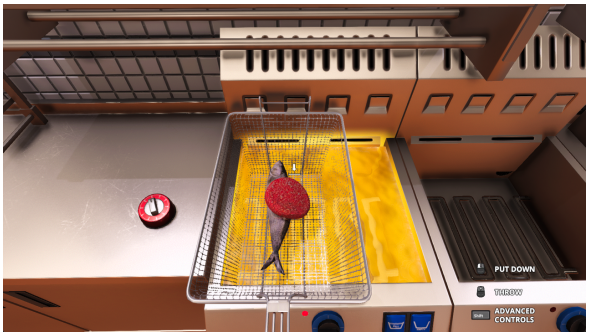

Figure 9: Tool–appliance affordance. Operating the deep fryer requires the fryer basket rather than direct placement, so a semantically correct "deep-fry" intention still fails without the correct tool chain.

D Detailed Challenge Analysis of CookBench

This section provides a more detailed description of the core challenges introduced by CookBench. These materials can support the high-level summary in the main paper while keeping the main narrative concise.

To illustrate its formidable difficulty, CookBench dismantles the assumption of perfect execution prevalent in prior benchmarks, introducing true execution uncertainty through simulated physical dynamics. A seemingly correct semantic action might fail due to subtle spatial misalignments, missing preconditions, or capacity limits. This non-determinism, coupled with the benchmark's massive state space and extended horizons, forces agents to rely heavily on closed-loop feedback, fundamentally elevating the overall task difficulty.

D.1 Large Action Space and Dense Object State

CookBench introduces a substantially larger and more fine-grained control space than short-horizon household benchmarks. Agents must choose among 25 action categories, 85 atomic actions, and 13 reusable skills, while interacting with hundreds of ingredients, liquids, seasonings, tools, containers, appliances, and placement points. The resulting difficulty is not merely that the action list is long: the same semantic recipe step may require different executable routines depending on the current object instance, held item, container state, spatial position, and appliance status. This is why the benchmark stresses action-space decision making, instance binding, and embodied working memory rather than only instruction following.

D.2 Operation Logic and Affordance Constraints

Cooking requires domain-specific affordance reasoning. Agents must know which tool can operate which appliance, which container can hold which material, and which body-state constraints forbid direct interaction. These constraints are simple for humans but difficult for agents that treat the environment as a symbolic API. Figures 9, 10, and 11 show representative cases.

ID	Dish	ID	Dish	ID	Dish
1	Lemon Chicken Breasts	46	Smokey German Potato Salad	91	Ukrainian Borscht
2	Lemon Chicken Breasts with Fruit Salad	47	Baked Cod with Greek Salad	92	Fast Ratatouille
3	Lemon Tart	48	Grilled Tuna Steak	93	Ratatouille
4	Mango Tart	49	Grilled Tuna Steak with Orange Salad	94	Fast Shakshuka
5	Brussels Sprouts with Pancetta & Chicken Wings	50	Salmon with Asparagus	95	Shakshuka
6	Brussels Sprouts with Pancetta & Grilled Swordfish	51	Salmon in Butter Sauce with Asparagus	96	Pork Chops with Fried Egg and Lemon Baked Potatoes
7	Fiesta Corn with Tomatoes	52	Salmon Steak and Boiled Potatoes	97	Ukrainian Borscht with Boiled Egg
8	Fiesta Corn with Tomatoes & Spicy Chicken	53	Salmon Steak, Potatoes, Grilled Tomatoes	98	Grilled T-bone Steak and Baked Red Potatoes
9	Sausage & Egg Muffin	54	Fried Shrimp Boil	99	Barbecue T-bone Steak with Potatoes and a Corn on the Cob
10	Sausage & Egg Muffin with Bacon	55	Baked Shrimp Boil	100	Barbecue T-bone Steak with Potatoes and a Hot Corn on the Cob
11	Melting Potatoes	56	Chicken Leg with Caprese Salad	101	Sweet and Sour Pork
12	Melting Potatoes with Egg de Provence	57	Marinated Chicken Leg with Caprese Salad	102	Marinated Sweet and Sour Pork
13	Grilled Buffalo Wings	58	Hamburger	103	Kung Pao Chicken
14	Grilled Buffalo Wings with French Fries	59	Honey-Mustard Burger with French Fries	104	Marinated Kung Pao Chicken
15	Spicy Sparkling Pork	60	Duck Breast with Apples	105	Easy Chinese Egg Drop Soup
16	Spicy Sparkling Pork with Spring Salad	61	Duck Breast with Roasted Mushrooms	106	Chinese Egg Drop Soup
17	Mango Salsa Chicken	62	Grilled Beef Chuck with Baked Potatoes	107	Orange Pork Stir-Fry
18	Mango Salsa Chicken with French Fries	63	Beef Chuck with Potatoes and Garlic Sauce	108	Orange Pork Stir Fry with Brussels Sprouts
19	Chicken Tortellini Soup	64	Pork Chops and Lemon Baked Potatoes	109	Helles Marzenbier Roasted Pork
20	Chicken Tortellini Soup with Croutons	65	Pork Tenderloin in Mustard Sauce	110	Dunkles Marzenbier BBQ Chicken
21	Double Potato Salad with Pesto	66	Steak with Barbecue Sauce and Vegetables	111	Currant-Glazed Pork Tenderloin with Red Cabbage and Thyme Dumplings
22	Double Potato Salad with Button Mushroom Sauce	67	Blended Fusilli Aglio, Olio e Peperoncino	112	Simple Chicken Pumpkin Stew
23	Baked Potatoes with Feta	68	Fast Beef Stroganoff over Fusilli	113	Chicken Pumpkin Stew
24	Baked Potatoes with Feta & Tomatoes	69	Fast Beef Stroganoff over Buttered Fusilli	114	Chicken Pumpkin Stew with Tomatoes
25	Italian Home Fries	70	Beef Stroganoff over Buttered Fusilli	115	Roasted Christmas Ham
26	Italian Home Fries with Cucumber Salad	71	Fusilli with Neapolitan Sauce	116	Roasted Christmas Ham with Garlic Fried Potatoes
27	Grilled Swordfish Provencal	72	Fusilli with Blended Neapolitan Sauce	117	Lobster Tail Cooked in White Wine
28	Grilled Swordfish with Hot Sauce	73	Fast Pasta alla Genovese	118	Lobster tail Cooked in White Wine with Grilled Asparagus
29	Chicken Tikka Masala	74	Pasta alla Genovese	119	Grilled vegetables
30	Chicken Tikka Masala with Potatoes	75	Penne in Broccoli Sauce	120	Grilled cabbage burger
31	Fried Shrimp	76	Penne in Broccoli and Mushroom Sauce	121	Grilled white sausage
32	Baked Cod	77	Penne in Salmon Sauce	122	White sausage roasted in beer
33	Baked Trout	78	Penne in Fragrant Salmon Sauce	123	Cowboy steak with baked potato and broccoli
34	Steak with French Fries	79	Fast Caldo Verde	124	Sweet and smoky salmon
35	Gazpacho	80	Caldo Verde	125	Coffee-seasoned steak with corn on the cob
36	Pork Chops with Baked Potatoes	81	Corn Chowder	126	Grilled lobster tail in lime sauce
37	Pork Tenderloin with Caramelized Vegetables	82	Corn, Scallop and Bacon Chowder	127	Roasted pork ribs
38	Baked Trout with Roasted Brussels Sprouts	83	Duck Broth	128	Grilled pork ribs with french fries
39	Fusilli Aglio, Olio e Peperoncino	84	Duck Consomme	129	Fruit salad
40	Fast Pumpkin Soup	85	Chunky Gazpacho	130	Salad with eggplant, halloumi, and tomatoes
41	Simple Ukrainian Borscht	86	Pumpkin Soup	131	Halloumi burger
42	Tomato Bruschetta	87	Pumpkin Soup with Croutons	137	New York Steak with Garlic
43	Shrimp Salad with Tomato Bruschetta	88	Tomato Soup	138	Simple New York Steak
44	Fried Garlic Shrimp	89	Red Pepper and Tomato Soup		
45	German Potato Salad	90	Red Pepper and Tomato Soup with Toast		

Table 10: Complete list of the 133 single-dish tasks used in CookBench. IDs correspond to the cooking-style taxonomy in Appendix C.2.

ID	Object	ID	Object	ID	Object	ID	Object
0	T-bone Steak	63	Asparagus	126	Ground coffee	189	Sunflower Oil
1	Salmon Fillet	64	English Muffin	127	Curry powder	190	Honey
2	Star Anise	65	Apple	128	Allspice powder	191	Soy Sauce
3	Cheddar	66	Eggplant	129	Ginger	192	Sour Cream
4	Swordfish	67	Strawberry	130	Turmeric powder	193	Lime Juice
5	Pumpkin	68	Button Mushroom	131	Cumin powder	194	Pesto
6	Tuna	69	Feta	132	Marjoram dried	195	Avocado Oil
7	Halloumi	70	Scallion	133	Garlic dried	196	Chicken Broth
8	Potato	71	Mint Leaf	134	Herbs de Provence dried	197	Dunkles Marzenbier
9	Potato Dumplings	72	Shrimp	135	Oregano dried	198	Square Plate
10	Bacon	73	Honey Mango	136	Lovage dried	199	Large Plate
11	Serrano Pepper	74	Fusilli	137	Thyme dried	200	Plate
12	Jalapeno	75	Broccoli	138	Basil dried	201	Small Plate
13	Garlic	76	Zucchini	139	Dill dried	202	Deep Plate
14	Goat Cheese	77	Horseradish	140	Fenugreek Leaves dried	203	Casserole
15	Parmesan	78	Rosemary Twig	141	Mint dried	204	Bowl
16	Tortellini	79	Penne	142	Horseradish dried	205	Barrel of dark beer
17	Pancetta	80	Lime	143	Rosemary dried	206	Barrel of light beer
18	Gorgonzola	81	Green Bell Pepper	144	Bay Leaf dried	207	Bucket
19	Scallop	82	Bread	145	Cilantro Leaves dried	208	Knife holder
20	Fingerling Potato	83	Sausage	146	Lemon Pepper	209	Trash can
21	Brussels Sprouts	84	Banana	147	Brown Sugar	210	Carton Box
22	Whole Ginger	85	Mozzarella	148	Mustard Seeds	211	Repair Phone
23	Parsley Leaves dresh	86	Anchovy	149	Starch	212	Phone Holder
24	Basil Leaf dresh	87	Cod	150	Smoked Pepper powder	213	Computer
25	Dill dresh	88	Trout	151	Sweet Bell Pepper powder	214	Refrigerator
26	Bay Leaf dresh	89	Chicken Wing	152	White Pepper	215	Refrigerator Left Door
27	Cilantro Leaves dresh	90	Chicken Breast	153	Salt	216	Refrigerator Right Door
28	Sage Leaf dresh	91	Chicken Leg	154	Sugar	217	Refrigerator Left Drawer
29	Pork Breakfast Sausage	92	Egg	155	Cinnamon ground	218	Refrigerator Right Drawer
30	Mooncake	93	Duck Breast	156	Nutmeg powder	219	Automatic Cutter Button
31	Lemon	94	Yellow Bell Pepper	157	Sichuan Pepper powder	220	Automatic Cutter Inlet
32	Raspberry	95	Cucumber	158	Cayenne Pepper powder	221	Auto Cutter Container
33	Pear	96	Lobster Tail	159	Chili Flakes	222	Fryer-Slot a
34	Radish	97	Automatic Cutter Container	160	Vanilla powder	223	Fryer-Slot b
35	Orange	98	Bake Tray	161	Black Pepper	224	Fryer Switch
36	Parsnip	99	Big Pot	162	Cola	225	Fryer oil filling switch
37	Parsley Root	100	Blender	163	Bavarian Beer Vinegar	226	Fryer oil discharge switch
38	Burger Meat	101	Colander	164	Balsamic Vinegar	227	Griddle Switch
39	Bottom Burger Bun	102	Cutting Board	165	Elderflower Jelly	228	Grill Switch
40	Top Burger Bun	103	Food Processor Container	166	Dusseldorf Mustard	229	Faucet Base
41	Onion	104	Fryer Basket	167	Lemon Juice	230	Cool Faucet Switch
42	Ham	105	Grill Pan	168	Coconut Milk	231	Hot Faucet Switch
43	Steak	106	Knife	169	Olive Oil	232	Right Sink Place Point
44	Beef Chuck	107	Ladle 150 ml	170	Water	233	Left Sink Place Point
45	Oregano Twig	108	Ladle 50 ml	171	Dijon Mustard	234	Faucet Pipe
46	Pork Chops	109	Large Skewer	172	Hoisin Sauce	235	Checkout Stand
47	Pork ribs	110	Masskrug	173	Helles Marzenbier	236	Food Processor button Off
48	Pork Shoulder	111	Medium Skewer	174	Barbecue Sauce	237	Food Processor button On
49	Pork Tenderloin	112	Paella Pan	175	Milk	238	Top Oven Switch
50	Cob of Corn	113	Pan	176	Ketchup	239	Bottom Oven Switch
51	Tart	114	Pipette	177	White Wine	240	Oven Top place point
52	Beetroot	115	Plastic Bowl	178	White Vinegar	241	Oven Bottom place point
53	Tomato	116	Small Pot	179	Rice Wine	242	Baking tray rack point
54	White sausage	117	Small Skewer	180	Rice Vinegar	243	Stove Place Point
55	Thyme Twig	118	Spatula	181	Red Wine	244	Stove Switch
56	Red Cabbage	119	Tongs	182	Redcurrant Jelly	245	Hook Point
57	Red Bell Pepper	120	Stove	183	Clarified Butter	246	Hook Point a
58	Red Potato	121	Fryer	184	Sesame Oil	247	Hook Point b
59	Chili Pepper	122	Oven	185	Mustard	248	Fusilli Box
60	Chives	123	Griddle	186	Peanut Oil	249	Penne Box
61	Cinnamon stick	124	Grill	187	Salsa	250	Potato Dumplings Box
62	Carrot	125	Cloves ground	188	Wine Vinegar	251	Tortellini Box

Table 11: nteractive object vocabulary supported by the CookBench environment. The table lists the English object names used by the agent interface

Figure 10: Container-material constraint. A solid-only container cannot serve as a liquid receptacle, which forces agents to distinguish generic "container" semantics from physically valid containment affordances.

Figure 12: Volume and capacity constraint. Pouring is a continuous process: if the agent does not track the target volume and container capacity, liquid overflow becomes an irreversible source of score loss.

Figure 11: Body-rule constraint. Hot ingredients cannot be directly picked up by hand; successful recovery requires using an appropriate tool or container rather than retrying the same pickup action.

Figure 13: Temporal state-memory failure. Compared with Figure 11, this state is reached after additional heating; the popup shows that the heating progress has changed. If heating continues while the agent plans or performs other steps, failure to remember and close the active heating state can burn ingredients even when the recipe-level plan is correct.

D.3 Continuous State Changes and Temporal State Memory

Many CookBench states evolve continuously after an action is triggered. Liquids accumulate until they reach a container capacity, food heats or burns over time, and appliances may require timers, waiting, switching off, or repair. Figures 12, 13, and 14 illustrate these evolving physical processes. These cases are a temporal dimension of embodied working memory rather than a separate capability category: an agent must remember that a physical process is still active, monitor how long it has been active, stop it at the right time, and recover if the process has crossed a failure threshold.

D.4 Physical Simulation Anomalies and Irreversible Failures

The simulator also introduces physical anomalies that are hard to capture with pure recipe logic. Figures 15, 16, 17, and 18 show representative cases in which object fragility, tool geometry, articulated-object interference, and embodied self-interference make a semantically plausible action physically fail. These failures matter because they change the world state rather than merely returning an invalid API call. Once the state changes, agents must repair, replace, discard, or replan around the damaged physical situation.

D.5 Knowledge Requirements in Cooking Tasks

These challenge cases show why CookBench failures cannot be reduced to missing recipe steps. Successful execution requires five intertwined forms of knowledge: recipe knowledge for ingredient composition and target dish requirements; object knowledge for forms, locations, and state-dependent properties; tool knowledge for capacities and tool–operation pairings; embodiment-rule knowledge for hand limits, modes, placement rules, and navigation-before-interaction; and strategy knowledge for timing, recovery, and score-

Figure 14: Appliance degradation and recovery. Some devices can become unusable after extended operation and require repair, introducing an exception-handling branch that must be integrated into the remaining cooking plan.

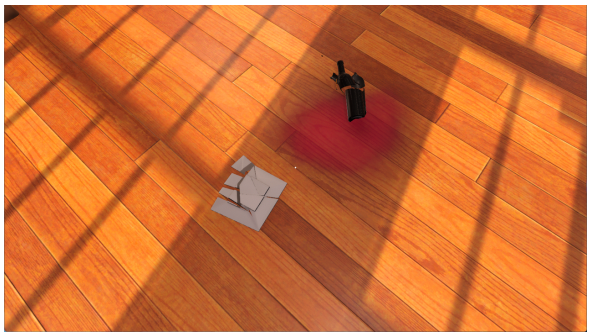

Figure 15: Fragile-object failure. Bottles and plates may break after collision or dropping, turning a local placement or navigation mistake into an irreversible loss of tools, containers, or ingredients.

Figure 16: Geometry-sensitive tool use. Object shape affects whether a tool can successfully manipulate the target, so agents must reason about physical geometry rather than only object categories.

Figure 17: Articulated-object interference. A door may fail to close when blocked by an open drawer, requiring the agent to identify and resolve the blocking state before repeating the door action.

oriented prioritization. The same surface failure may arise from different mechanisms, as summarized in Figure 19.

E Additional Quantitative Results

E.1 Extended Analysis Questions

The main paper compresses the analysis into the phenomenon–mechanism–capability funnel. Here we retain only the figure-supported analysis questions that are directly backed by the appendix or main-text visualizations.

Can agents exploit waiting states for effective temporal scheduling? Figures 20 and 21 provide the timer-specific evidence for this question. We parse every successful `countdown_time` call as a timer window and distinguish three aspects of temporal scheduling. *Timed-step coverage* measures whether explicit timed recipe operations are covered by timer calls, rather than the fraction of all actions that are timers. *Idle-wait rate* measures whether the timer interval is wasted without useful concurrent progress. *Device not closed* measures whether a timer window with a preceding heating-device `toggle_on_switch` lacks a corresponding `toggle_off_switch` within 30 steps after the timer ends.

The key pattern is that timer use alone is insufficient evidence of temporal-state maintenance. Several methods cover a non-trivial fraction of timed recipe steps, yet still exhibit high idle-wait rates and high device-not-closed rates. This indicates that current agents often treat timer calls as blocking pauses instead of coordinating suspended heating states with other preparation, transfer, and assembly subtasks. Overall, this phenomenon should be interpreted as part of the **embodied working memory** gap: agents can sometimes recognize that a timed operation is required, but they do not reliably maintain the active process in memory, exploit the waiting interval, and close the corresponding physical state when the timer finishes.

Figure 22 further isolates the physical switch lifecycle behind the device-not-closed metric. For

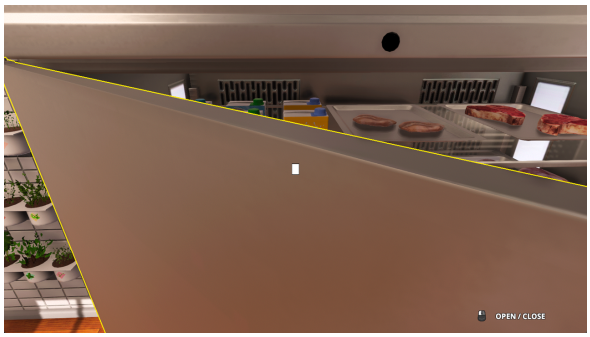

Figure 18: Embodied self-interference. The agent's own body position can block an appliance or door, so successful interaction may require repositioning before the intended open/close action becomes physically executable.

each timer window, we first search the preceding 20 steps for an inferred heating-device `toggle_on_switch`; only timer windows with such a prior device-on event enter this denominator. We then search up to 30 steps after the timer window ends for an inferred `toggle_off_switch`. The bars divide the shutdown delay into four buckets: within 3 steps, 4–10 steps, 11–30 steps, and beyond 30 steps or missing. The large "> 30 / missing" segments show that the problem is not merely delayed timer recognition: agents frequently open a heating device before a timer, but fail to close the device soon after the timed interval finishes.

Can agents repair execution after failure in a physically grounded way?

Detailed reading of the closed-loop repair signals. Figure 5 in the main text summarizes two negative signals extracted from execution traces. *Blocked retry* is computed over failed physical execution steps within the 200-step budget, excluding pure query, order, wait, and timer bookkeeping actions. For each failed step, the parser inspects the next five executable steps; if this window contains another failed action with the same action name, the event is counted as a blocked retry. This metric therefore asks whether the agent converts feedback into a different physical repair strategy, or merely repeats the same locally blocked operation.

Replan error is computed over planner calls that are triggered after at least one execution failure since the previous planner call. For each such failure-triggered replan, the parser inspects the next five executable steps; if any execution error appears in that short post-replan window, the event is counted as a replan error. This metric asks whether replanning stabilizes the immediate execution suffix, rather than whether the whole episode eventually succeeds.

The resulting pattern shows that closed-loop feedback is often consumed but not physically grounded. In the GPT-5.4 framework comparison, blocked retry remains substantial for all frameworks: 58% for ReAct, 39% for PE, 41% for Reflexion, and 48% for CaP. The corresponding post-replan error rates are even higher: 77%, 67%, 68%, and 78%, respectively. Under CaP with different multimodal models, the same pattern persists: blocked retry ranges from 31% for Gemini-3.1-Pro to 68% for Qwen3.6-35B, while post-replan error ranges from 58% to 86%. These values indicate that feedback and replanning frequently keep agents locally active but do not reliably transform the physical state in the direction required for recovery. This supports the main-text conclusion that current agents lack robust **physical feedback comprehension**: they often interpret embodied constraints as retryable parameter errors rather than as signals requiring state-changing actions such as releasing a held object, exiting an interaction mode, or using a container-mediated transfer.

Why Can CaP Score Higher Despite Frequent Physical Errors? Figure 23, together with Table 2, suggests that CaP's advantage is not that it simply executes more actions, but that it allocates a larger share of its action budget to score-relevant state-changing operations. Under the fixed GPT-5.4 backbone, CaP does *not* have the largest raw action count per dish (75.74, versus 82.44 for Reflexion), nor the largest action-type diversity (14.49, versus 15.44 for Reflexion). Its advantage lies in executing the largest number of high-value actions per dish (31.39) and the highest high-value action rate (0.4145), above PE (28.42, 0.3725), ReAct (28.34, 0.3686), and Reflexion (30.11, 0.3653). This matches the score ranking in Table 2: CaP@GPT-5.4 reaches 8.35 overall, clearly above Reflexion@GPT-5.4 (6.05), PE@GPT-5.4 (5.28), and ReAct@GPT-5.4 (4.12). In other words, within the same backbone, CaP converts a similar total action budget into more cutting, heating, transferring, plating, and other directly score-bearing operations, which helps it accumulate reward even when many individual physical executions still leak. The design implication is important: the score gap is *not* explained by running longer or exploring more blindly. ReAct and PE

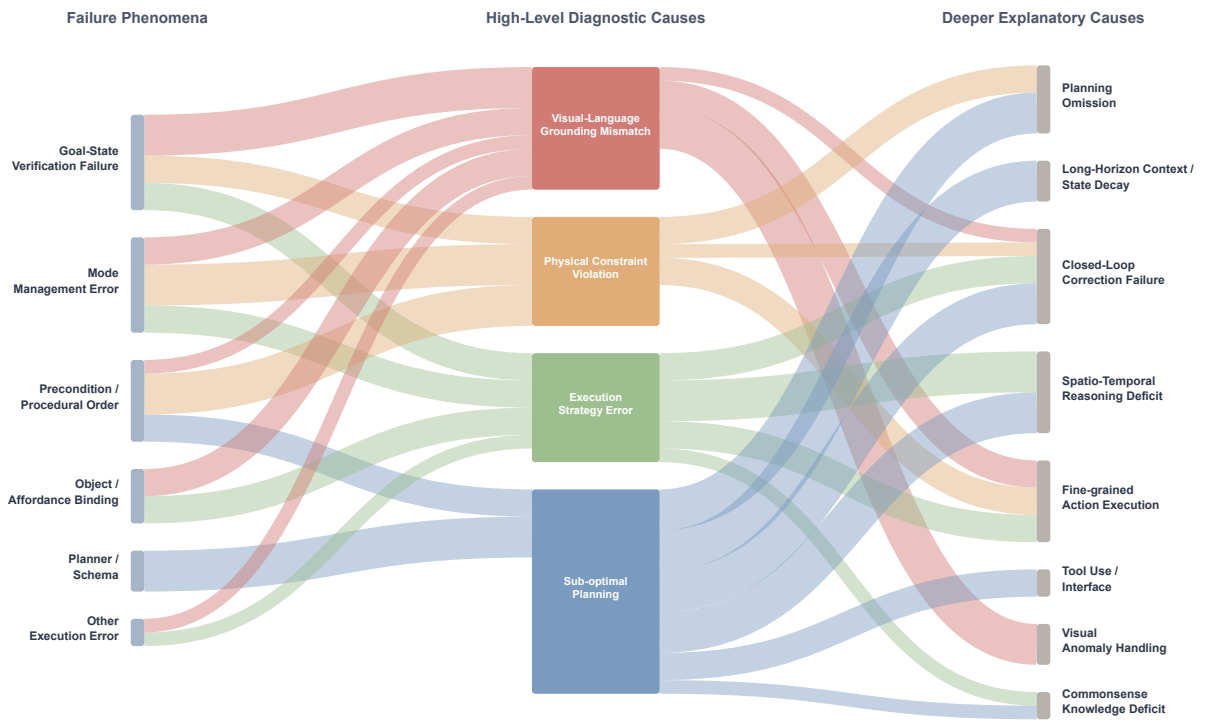

Figure 19: Mechanism-level interpretation of failure phenomena. The Sankey-style diagram separates observable failure phenomena from deeper diagnostic causes, illustrating that one surface error can arise from multiple mechanisms such as weak physical grounding, invalid mode or body-rule modeling, poor repair after feedback, and state-memory degradation.

spend a comparable number of core actions, but their short-horizon replanning or chunk-level replanning burns a larger fraction of the budget on setup, correction, and local recovery transitions. Reflexion increases action diversity, yet much of that extra diversity is spent on branching and textual repair attempts rather than stable culinary state advancement.

This explanation is also consistent with the category-level pattern in Table 2. CaP@GPT-5.4 is not carried by a single easy category; it is broadly strong on multi-step dishes such as Soup-Stew (9.02), Pasta-Assembly (13.72), and Salad-Cold (6.91), where progress depends on repeatedly completing ingredient transformations and container transfers. By contrast, the weaker GPT-5.4 frameworks spend a comparable or even larger number of total actions, but fewer of those actions become productive culinary state changes. The gap is especially revealing for Reflexion: it explores more broadly than CaP in terms of unique action types, yet its lower high-value action rate and much lower final score indicate that closed-loop activity alone does not guarantee useful task advancement.

However, the model comparison under fixed CaP shows that aggressive high-value exploration is only a *necessary* condition, not a sufficient one. Gemini-3.1-Pro and Qwen3.6-Plus are even more aggressive than GPT-5.4, reaching high-value action rates of 0.4412 and 0.4452, with more than 37 high-value actions per dish on average. Yet their final scores remain 6.77 and 4.20, well below CaP@GPT-5.4's 8.35. Qwen3.6-35B exhibits a similar mismatch: despite 37.09 high-value actions per dish and a 0.4249 high-value rate, its final score is only 3.84. This mismatch indicates that "doing more high-value-looking actions" is not enough if those actions are poorly grounded, mistimed, or fail to converge to the required final serving state. The category breakdown further supports this point. Gemini-3.1-Pro can be highly competitive on Grill-Single (11.73), but its Grill-Combo score drops to 1.96, showing that aggressive exploration does not automatically scale to dishes requiring tighter multi-component coordination. Qwen3.6-35B reaches 12.73 on Pasta-Assembly but nearly collapses on Salad-Cold (0.06), revealing that it can sometimes push one linear pipeline deeply while still failing on categories that require more stable long-horizon state maintenance.

Taken together, the violin-plot statistics explain CaP's advantage more precisely than a simple

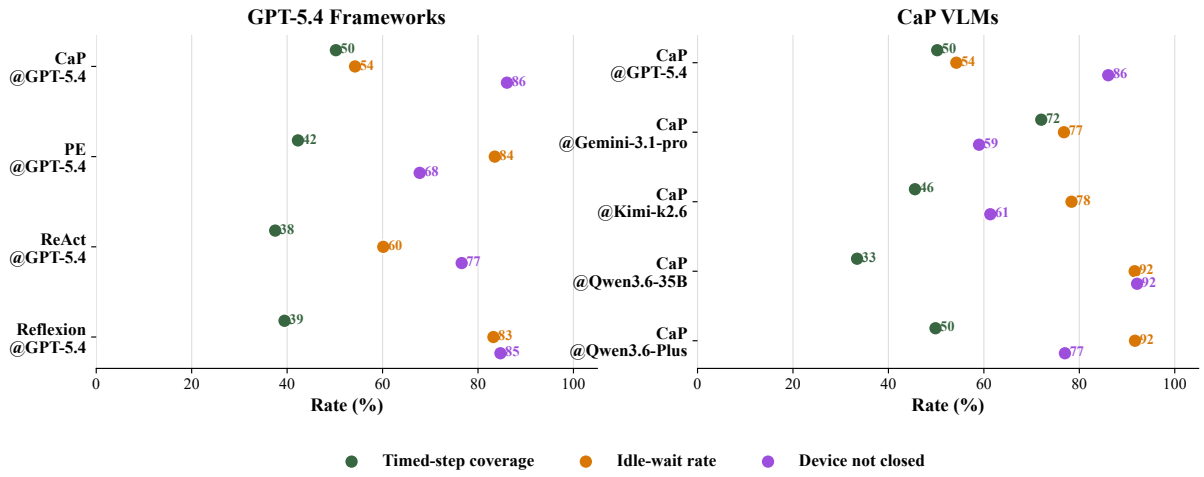

Figure 20: Timer-scheduling overview across the two experimental stages. The left panel compares GPT-5.4 under different agent frameworks, and the right panel compares VLM backbones under the fixed CaP framework. Timed-step coverage is not timer-action frequency; it estimates how many explicitly timed recipe operations are covered by timer calls. Idle-wait and device-not-closed rates expose whether agents waste timer windows or fail to close heating devices after timers finish.

"more exploration is better" story. The framework-level comparison shows that CaP wins under GPT-5.4 because it spends a larger fraction of its budget on score-bearing culinary operations rather than dissipating effort on low-yield interaction. The model-level comparison then shows the boundary of this mechanism: once exploration becomes poorly grounded or weakly recoverable, a high-value action profile can coexist with mediocre final scores. Therefore, CaP's higher score is best understood as a combination of *productive exploration* and *sufficient execution reliability*, rather than as evidence that physical errors no longer matter.

E.2 Micro-Mechanism Breakdown of Execution Failures

This section provides an extended analysis of the failure distributions discussed in Section 4.3 (RQ1), detailing the formal taxonomy of execution errors and providing a granular breakdown of how different control flows collapse under physical constraints.

E.2.1 Failure Taxonomy: The Semantic-to-Physical Spectrum

To accurately diagnose execution breakdowns, we classify failure logs along a spectrum ranging from abstract reasoning flaws to grounded physical interaction failures:

- **Schema and Binding Errors (Semantic Layer):** These reflect abstract planning flaws. *Schema* errors occur when the agent outputs invalid action formats, non-existent action names, or unsupported API calls. *Binding* errors happen when the agent issues a valid action type but references incorrect, misaligned, or hallucinated target objects.
- **Precondition and Mode Errors (Physical Layer):** These occur when agents violate spatial prerequisites or kinematic states. For example, a *Precondition* error is triggered if an agent attempts to cut a tomato without holding a knife or being close to the cutting board. A *Mode* error occurs if the agent attempts to perform a mode-specific interaction (e.g., pouring) without successfully entering the required physical state (e.g., `pouring_mode`).
- **Post-State Check Failures (Actuation Layer):** These indicate actions that are syntactically valid and pass initial physical precondition checks, but ultimately fail to actuate the intended environmental change (e.g., attempting to place an item, but the item drops outside the valid receptacle boundaries).

E.2.2 Automatic Log Classification Procedure

The phenomenon-level error statistics in Figure 4 and the appendix breakdowns are produced by deterministic Python log-parsing scripts, not by free-form model judgment. The scripts parse each `step_trace.log` or long-horizon history entry with a regular expression that extracts the step

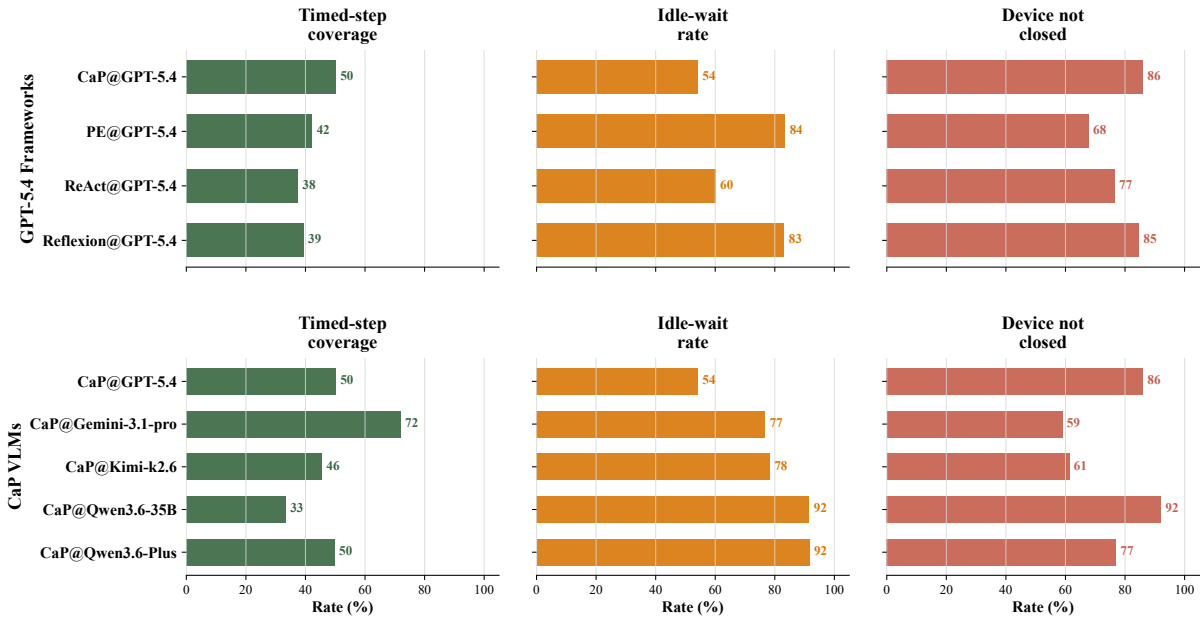

Figure 21: Simplified timer-scheduling failure view. The three columns separate timer-step coverage, passive waiting, and post-timer device shutdown failure. This representation emphasizes the main failure pattern: agents may create timers, but they often do not exploit waiting periods for parallel subtasks and frequently fail to close heating devices after the timer ends.

index, action type, action name, arguments, execution result, and simulator/EPM error string. Only failed steps with non-empty error messages are counted in the error taxonomy; successful steps are excluded from the numerator.

The first-round GPT-5.4 framework analysis uses a detailed Q1 trace table generated by the long-horizon analysis script. This script first assigns a fine-grained `Cause Coarse` label and then maps paper-facing labels to the six macro categories used in our figures: `precondition_or_action_order_failure` → *Precondition/Order*, `mode_management_failure` → *Mode*, `target_or_parameter_binding_failure` → *Binding*, `post_state_check_failure` → *Post-State*, and `planner_or_interface_schema_failure` → *Schema*. Runtime exceptions, API/provider failures, terminal anomalies, submission failures, navigation failures, specialized-skill failures, and unmatched cases are grouped into *Other Execution Failure* for the paper-level plots.

For the second-round CaP VLM comparison, where we classify logs directly from the action name plus error string, the script uses ordered substring matching. *Schema* is matched by tokens such as `invalid_plan`, `unknown_action`, `schema`, `json`, and `missing required`. *Mode* is matched by interaction-mode tokens such as `enter_pouring_mode_failed`, `enter_cutting_mode_failed`, `enter_sprinkle_mode_failed`, `exit_pouring_mode`, `exit_current_interaction_mode`, `blocked_by_mode`, `interaction_mode_still_active`, and messages such as "not in pouring/cutting/sprinkle mode". *Binding* is matched by target or instance resolution tokens such as `invalid_target_name`, `target_not_found`, `target_kind_mismatch`, `instance_disambiguation`, `invalid_instance`, and `object_not_found`. *Post-State* is matched by failed state-verification tokens such as `post_check_failed`, `post_state:`, `not_holding_after_click`, `still_holding_item`, `drawer_not_open`, and `door_not_open`. *Precondition/Order* is matched by unmet-prerequisite tokens such as `requires_`, `precheck_failed`, `hands_empty`, `not_holding`, `already_held`, `container is full`, and `source_empty`.

When a direct-parsed error string matches multiple categories, the script assigns the first matched category under the fixed priority order *Schema* → *Mode* → *Binding* → *Post-State* → *Precondi-*

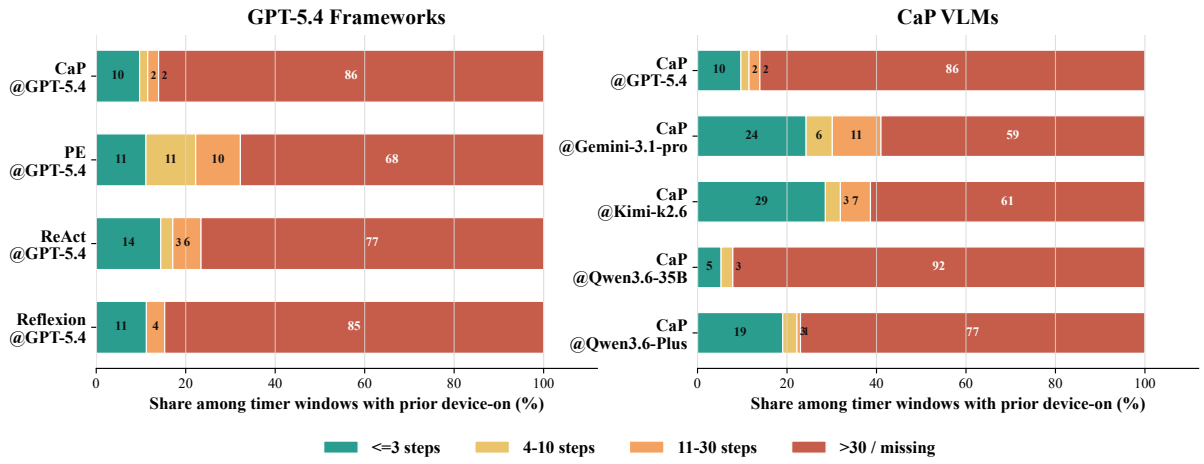

Figure 22: Heating-device switch lifecycle after timer completion. Each row includes only timer windows with an inferred heating-device `toggle_on_switch` in the preceding 20 steps. Segment colors show how quickly a later `toggle_off_switch` appears after the timer window ends. The dominant > 30 / missing segments indicate that agents often fail to close heating devices after timed cooking intervals.

tion/Order $\rightarrow$ *Other*, so each failed step contributes exactly one count. The more detailed Q1 script additionally performs earlier filters for external API errors, runtime bugs, environment anomalies, submission failures, and specialized-skill failures before applying the paper-facing physical-error labels; those non-core categories are collapsed into *Other* in the main taxonomy. We use this automatic classification for aggregate plots and then inspect representative traces and screenshots qualitatively when discussing fine-grained mechanisms; therefore, the taxonomy should be interpreted as a reproducible log-side diagnostic abstraction rather than a complete causal proof of every individual failure.

E.2.3 Framework-Specific Vulnerabilities

While pure logical errors represent a distinct minority across the benchmark, execution breakdowns are heavily dominated by *Post-State* and *Mode* failures. A detailed analysis of the specific failure profiles exposes the distinct vulnerabilities embedded in the control flows of the evaluated frameworks:

- **ReAct (Local Physical Deadlocks):** *ReAct* exhibits an overwhelming 69.3% post-state failure rate. Its short-horizon strategy traps it in local physical deadlocks where actions are repeatedly issued but yield no real-world changes. Without explicit long-term progress bookkeeping, it fails to escape persistent physical obstacles.
- **Plan-and-Execute (Brittle Contracts):** *PE* presents a uniquely high schema error rate (29.0%). This highlights the brittle contract between its explicit 10-step chunk planner and the executor: a single semantic or formatting hallucination collapses the entire predefined chunk before physical grounding is even attempted.
- **Reflexion (The Illusion of Textual Correction):** *Reflexion* successfully suppresses the schema errors seen in PE through textual correction. However, its post-state (37.4%) and mode errors (27.4%) sharply inflate as a result. This phenomenon indicates that while pure textual reflection can patch syntactic formatting to bypass the API interface, it cannot fundamentally resolve true 3D physical obstructions.
- **Code-as-Policy (State Drift):** *CaP* shows the most distributed error profile. Although its programmatic structure acts as persistent working memory to mitigate immediate deadlocks, it still suffers from substantial mode and precondition errors. This is primarily due to the state drift between internal code variables (the agent's belief state) and external physical reality over extended execution horizons.

E.3 More Quantitative Results

Figure 24 provides the primary-ingredient-level diagnosis used in the main analysis. Figures 25 and 26 further compare action-level bottlenecks for

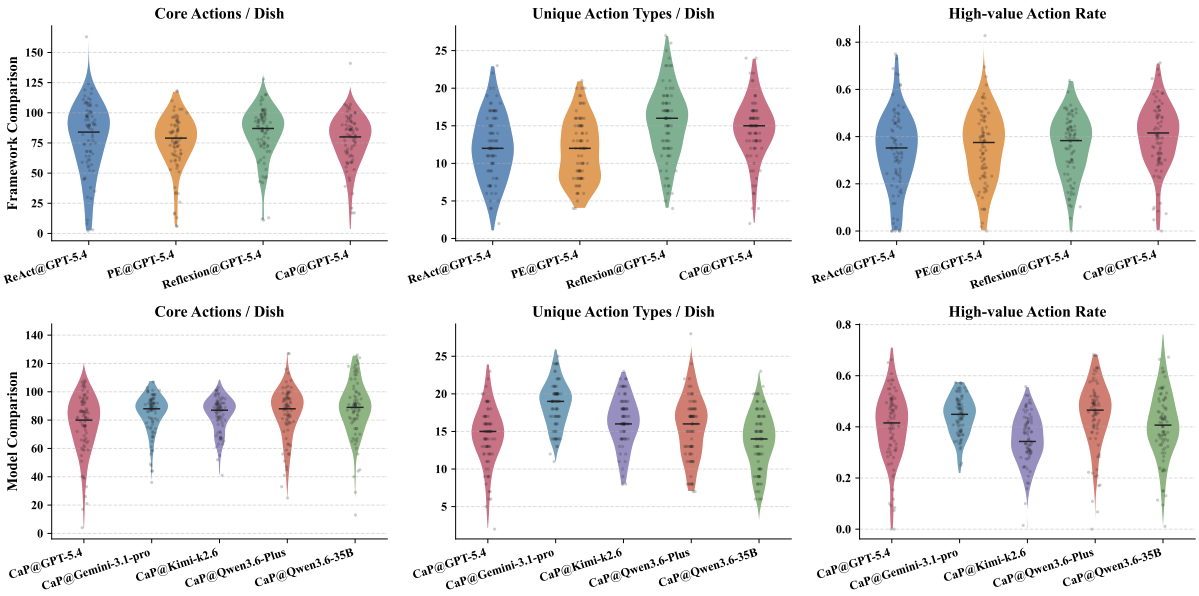

Figure 23: Action-exploration intensity across agent–model combinations. CaP executes more high-value cooking operations, helping explain why it can score higher despite frequent physical errors.

the GPT-5.4 framework study and the CaP VLM study.

Figure 24 sharpens the "last-mile" interpretation of the benchmark. Under the GPT-5.4 framework comparison, all closed-loop methods are dominated by *interacted but not finalized* failures rather than *not attempted* failures: ReAct 40.58%, PE 48.85%, Reflexion 50.90%, and CaP 44.26%. This means the main bottleneck is usually not recipe recall, but state convergence from intermediate manipulation to final dish assembly. The framework differences reveal where each design loses control. ReAct has the highest *not attempted or not found* share (26.81%), consistent with its short-horizon control flow losing track of pending ingredients before they are even engaged. PE lowers this early omission burden (19.47%) but still converts nearly half of classified failures into unfinished ingredient lifecycles, showing that chunk planning improves coverage more than finalization. Reflexion pushes the interaction rate even further: it reaches the highest interacted-primary rate among GPT-5.4 frameworks (77.45%), yet also the highest unfinished-finalization share (50.90%). This suggests that step-level reflection encourages renewed engagement with ingredients, but often without closing the full transfer, plating, or submission chain. CaP achieves the highest failure coverage over primary ingredients (73.49%) and the highest final-container evidence rate (18.67%), indicating that its programmatic control helps it track ingredient progress farther along the pipeline. However, even CaP still loses 44.26% of classified failures in the "interacted but not finalized" category, so its advantage is a partial mitigation rather than a resolution of the finalization problem.

The fixed-CaP model comparison reveals a second distinction: stronger backbones fail later in the ingredient lifecycle, while weaker backbones fail earlier. GPT-5.4 and Gemini-3.1-Pro keep *not attempted or not found* at 19.67% and 19.41%, whereas Qwen3.6-Plus and Qwen3.6-35B rise to 28.42% and 29.91%. This indicates that weaker VLMs more often break before stable interaction with the primary ingredient is established. By contrast, Kimi-k2.6 and Gemini-3.1-Pro frequently reach the ingredient but still stall at finalization, with unfinished shares of 56.35% and 46.47%. GPT-5.4 remains the strongest overall because it combines relatively low early omission with the highest final-container evidence, which aligns with its best overall score in Table 2. The broader lesson is that control-flow design mainly determines how far the agent can carry an ingredient through the execution chain, while backbone quality determines how often that chain breaks before the serving state is physically completed.

Figure 25 shows that, even when the language backbone is fixed to GPT-5.4, the dominant bottlenecks are not abstract planning verbs but embodied state-transfer primitives. The most important example is `put_down`: it is by far the most fre-

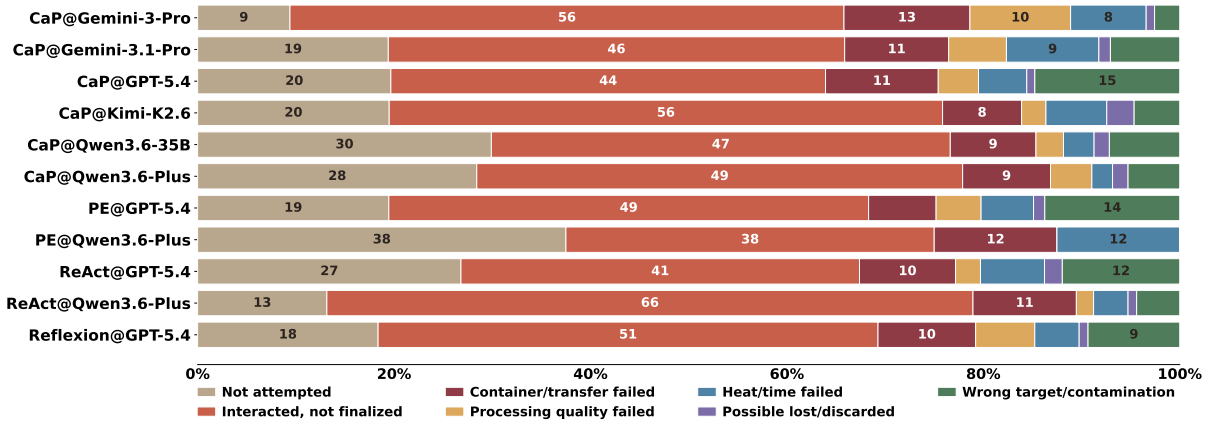

Figure 24: Primary-ingredient failure distribution across agent–model combinations. Numbers inside each segment denote the percentage among classified primary-ingredient failure events. The categories distinguish whether an ingredient was never attempted, interacted with but not finalized, lost during transfer, poorly processed, over-/under-heated, discarded, or contaminated by a wrong target. Most failures are not purely missing-plan cases: agents frequently interact with ingredients but fail to finalize them into the submitted dish.

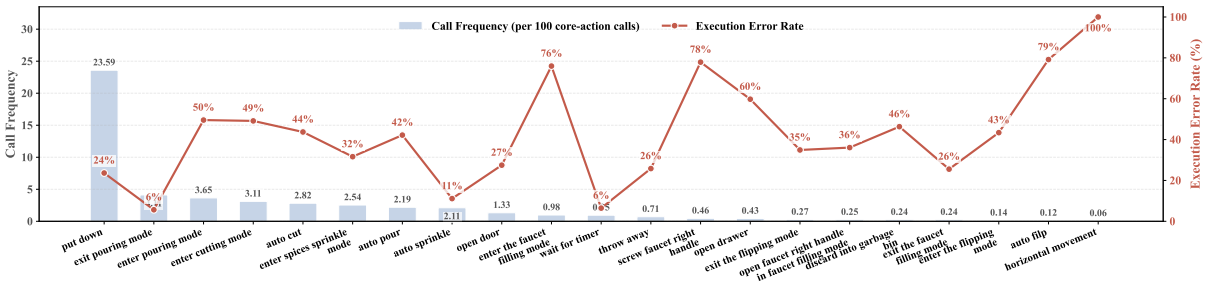

Figure 25: Action bottlenecks for GPT-5.4 frameworks. Bars show action-call frequency, and the line shows conditional error rate after excluding actions with zero observed errors. Actions are sorted by call frequency. Physical state-transfer actions have substantially higher error rates than simple navigation primitives.

quent action (23.59 calls per 100 steps) and still fails 23.63% of the time, making it both a common and costly source of score leakage. Mode-entry and manipulation-start actions are even more fragile: `enter_pouring_mode` fails 49.55% of the time, `enter_cutting_mode` 49.14%, `auto_cut` 43.72%, and `auto_pour` 42.19%. By contrast, `exit_pouring_mode` is frequent but has only a 5.64% error rate. This asymmetry is informative: the hard part is not issuing a cleanup command after the fact, but correctly binding the object, tool, spatial alignment, and interaction mode needed to enter a productive manipulation state in the first place.

These bottlenecks also explain why the four GPT-5.4 frameworks separate in score despite sharing the same underlying model. ReAct's short-horizon design is especially vulnerable once a high-frequency state-transfer primitive fails, because a failed `put_down`, `enter_pouring_mode`, or `auto_cut` can consume much of the local suffix and trigger another reactive detour before real progress resumes. PE improves global task coverage through chunk planning, but the chunk executor still pays the same embodied tax at these state-transfer junctions; thus its gains over ReAct are structural rather than action-level. Reflexion can reopen exploration after failures, which helps broaden action coverage, yet the figure shows that the core failure surface is still dominated by mode-entry and transfer actions that text-only reflection cannot physically resolve. CaP benefits because it invests more budget into these score-relevant actions than the other frameworks do, but Figure 25 clarifies why CaP still exhibits many physical errors: its preferred actions are exactly the ones with the highest embodied execution risk.

Figure 26 isolates what changes when the framework is fixed to CaP and only the multimodal backbone varies. The bottleneck profile

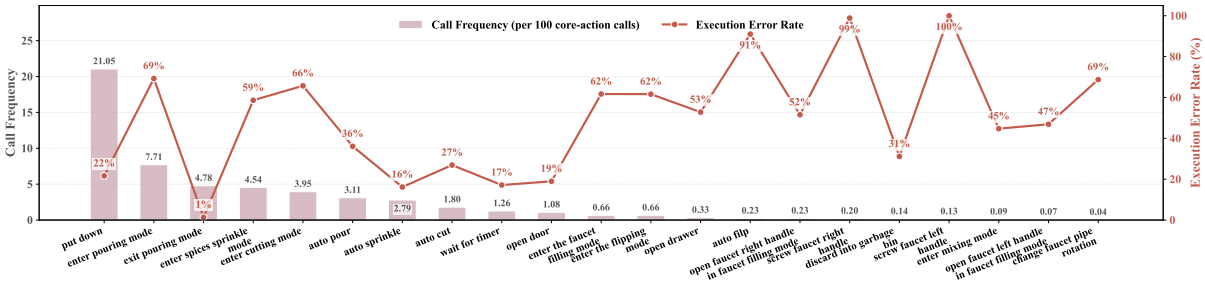

Figure 26: Action bottlenecks for CaP VLM settings. Bars show action-call frequency, and the line shows conditional error rate after excluding actions with zero observed errors. Actions are sorted by call frequency, exposing which frequently used embodied operations become high-risk under CaP-style execution with different multimodal models.

becomes even more concentrated around high-value mode-entry actions: `enter_pouring_mode` is called 7.71 times per 100 steps and fails 69.27% of the time, `enter_cutting_mode` fails 65.72%, and `enter_spices_sprinkle_mode` fails 58.64%. Faucet-control actions are even more brittle, with `screw_faucet_right_handle` at 98.85% conditional error and `screw_faucet_left_handle` at 100%. Meanwhile, `exit_pouring_mode` again remains almost error-free (1.36%). Relative to Figure 25, this shift indicates that once CaP encourages more aggressive manipulation attempts, the main cross-model discriminator is no longer whether the agent chooses a high-value action at all, but whether the backbone can ground the required preconditions before invoking it.

This is why the model comparison in Table 2 cannot be explained by exploration intensity alone. Gemini-3.1-Pro and Qwen3.6-Plus both exhibit higher high-value action rates than GPT-5.4 under CaP, yet only Gemini-3.1-Pro partially converts that aggressiveness into a competitive final score, while Qwen3.6-Plus and Qwen3.6-35B remain much weaker overall. Figure 26 provides the missing mechanism: under a fixed aggressive framework, poor mode grounding and poor object–state binding amplify the cost of every additional high-value manipulation attempt. In other words, CaP exposes backbone differences more sharply than the framework comparison does. Stronger models benefit from CaP because they can sustain aggressive state-changing action selection; weaker models suffer because the same aggressive control flow repeatedly drives them into the most brittle embodied primitives.

The per-method phenomenon-layer breakdowns complement the main-paper summary in Figure 4 by separating the profiles of *CaP*, *PE*, *Reflexion*, and *ReAct*. They make the contrast in failure profiles easier to inspect visually: *ReAct* is dominated by post-state check failures, *PE* shows a much larger planner–executor schema burden, and *CaP* and *Reflexion* exhibit more mixed profiles with substantial mode-management and local precondition failures.

F Prompt Templates and Examples

F.1 Prompt Structure and Layout

EPM does not use one monolithic handwritten prompt per method. Instead, each framework assembles a shared prompt envelope from modular assets under `epm/memory/`, then injects method-specific control lines that determine planning horizon, feedback usage, output contract, and recovery behavior. The full saved prompts are often several thousand lines long because they include the action catalog, supported-item snapshot, scene state, and runtime memory. For readability, Appendix F switches to a full-page-width layout and shows the framework-control fragments that most directly govern execution behavior.

F.2 Shared Prompt Envelope

All planning frameworks share the same observation shell. The common prompt body contains the current goal, raw recipe, oracle observation, agent state, countdown status, recent feedback, and the authoritative action/skill schemas. This shared structure keeps the method comparison mechanism-controlled.

```
SYSTEM:
- role / body rules / strategy notes
- action and skill schemas
- response contract

USER:
Goal: <current high-level goal>
Recipe (raw): <original recipe text>
Observation: <screenshot + oracle visible objects>
Agent state snapshot: <held item / mode / posture / timers / active container>
Countdown status: <episode step / forced-submit window / timer events>
Feedback from last execution: <success or structured error block>
```

F.3 Complete Saved Prompt Example

The framework fragments above show *how* control differs across methods. For completeness, we include one later-stage verbatim planning example from a PE@GPT-5.4 run in the archived experimental logs. The following prompt and its paired model output are copied directly from `experimental-result/pe/100-20260502_033929_857342-planner_executor-gpt-5.4/`. To keep the appendix readable while preserving a real saved prompt, we only shorten five oversized context dumps by keeping their first few and last few entries and omitting the middle portion: `Supported items (static snapshot)`, `Allowed actions/skills (authoritative schemas)`, `Placement occupancy (oracle snapshot)`, `Observation`, and `STM window snapshot`. The recipe, state snapshot, countdown status, execution feedback, and the model output remain unchanged.

Raw saved prompt (step_000194).

```
SYSTEM:
Role:
# System Role (Injected into SYSTEM prompt)
# update_policy: manual

You are an intelligent cooking planning robot (EPM) operating in CookingSimulator.

Your responsibilities:
- Based on the current task goal (Goal), visual observations (screenshot), and execution feedback
↪  (feedback/agent_state/stm_window), continuously plan and call allowed actions/skills to interact with the environment.
- Do not stop until the task is completed: place the final dish onto a plate (Plate/Deep Plate, etc.), hold the plate,
↪  navigate to the Checkout Stand, and submit the dish.

Working style:
- You can only use tools and parameter schemas listed under "Allowed actions/skills" in the prompt.
- When disambiguation/localization is needed, first use query/list tools to get candidates, then make an explicit selection
↪  (especially `instance_id`).
- Treat task-progress constraints, precondition feedback, and visual anomaly warnings as high-priority execution constraints.
- If they conflict with convenience staging, broad preparation, or speculative future-step setup, resolve the current
↪  blocker/subtask first.
- Prioritize making the final submitted result clearly match the target dish: after choosing the needed serving vessel or
↪  working container, quickly move on to recipe-defining ingredient processing, cooking, transfer, and assembly. Do not let
↪  "container preparation" expand into repeated staging or rearrangement loops.
```

```
- Empty-container handling by itself is not meaningful recipe progress. Picking up, moving, or staging an empty Plate / Deep
↪  Plate / Bowl / Pot / Pan only counts as setup when that exact container is needed for the current subtask or the immediate
↪  next interaction.
- Ingredients/components score mainly when they are correctly processed and assembled into the final submitted vessel or
↪  directly form the submitted food item itself. Therefore, after the required vessel is available, prioritize getting the
↪  main recipe-defining components into it instead of collecting or shuffling extra empty containers.
- Do not repeatedly acquire or stage duplicate serving vessels/containers of the same role unless the previously selected one
↪  is blocked, lost, wrong, full, or unusable.

Body rules (global constraints):
# Body Rules (Embodiment Rules)

Purpose: describe mandatory environment rules, interaction constraints, interface protocol, and execution hard constraints,
↪  as a static fragment in the Brain prompt.

[Rules]
1) Must arrive before placing: before executing the `put_down` placement action, you must first `auto_navigation` to the
↪  target placement point and face it. Ways to obtain placement points:
   - The system already provides `put_place_list.json` (platform categories and index ranges); you can directly reference
   ↪  platform categories in it (such as "Side Table" / "Countertop" / "Plate", etc.).
   - Use `auto_navigation(target="<platform_category>", target_instance_id=<index or instance_id>)` to navigate to a specific
   ↪  placement point (for platform categories, `target_instance_id` is treated as a platform index; or directly use the
   ↪  full name like "Side Table-3"). After navigating to the point and before placing, visual inspection is required to
   ↪  confirm alignment to a blank placement point; if entered into a wrong mode, exit that mode.
   - If `auto_navigation` feedback contains `put_place_platforms`, you may choose platform category and index from it.
2) Mandatory at the end: after all tasks are completed, the agent must hold a directly submittable serving carrier or food
↪  (for example a container, or `Tart`) and execute `gui_submit_dish_via_checkout_stand(dish_name: str)` to submit the dish
↪  and get a score.
2.1) One-dish submission constraint: unless the runtime `Task execution context` explicitly lists multiple active dishes, the
↪  current recipe must always be interpreted as one dish. Even if the recipe contains multiple components or multiple `serve
↪  hot` / `serve cold` lines, those are still components of the same dish rather than separate dishes to submit independently.
↪  A single dish may produce only one final submission result: consolidate it into one final submittable container or one
↪  directly submittable food item, and do not split one dish into multiple containers for separate submissions.
3) The agent can hold only one item at a time (held-item constraint); it must continue to pick up the next item only after
↪  `agent_state` shows empty hands (`is_held=false` and mode is `idle (hands empty)`). If the held item is a container,
↪  carrying multiple items in that container is allowed before continuing pickup.
4) The agent must execute corresponding actions in appropriate modes, such as pouring mode, sprinkling mode, cutting mode,
↪  mixing mode, flipping mode, `faucet_filling_mode`, etc.; for any complex interaction that enters a persistent interaction
↪  state, the agent must first check whether the current mode matches before deciding to execute, exit, or switch mode (mode
↪  constraint).
4.1) If after multiple mouse/keyboard perception actions, view-adjustment actions, or movement actions the camera pose still
↪  does not change, assume the agent is likely stuck in a mode or interaction state. In that case, exiting the current mode
↪  becomes the highest priority; use right mouse button by default, and any equivalent action that reliably exits the mode is
↪  acceptable.
5) The agent cannot directly pick up hot items with bare hands; it must first hold a container, then place the hot item into
↪  that container to effectively "pick up" the hot item (hot-item holding constraint). Do not plan bare-hand `pick_up` for
↪  hot items; prefer `hold container -> aim at hot item -> pick_up_into_the_container`.
6) Skills and atomic actions must be executed via supported actions and skills; bypassing the API to directly operate the UI
↪  is forbidden.
7) Action parameters must conform to the action schema (name, type, range, unit).
8) Same-name multi-instance must explicitly choose instance_id: when you need to execute an action/skill on a "normal object",
↪  and multiple instances with the same name exist in the scene (common for Cutting Board, Knife, Side Table, etc.), you must
↪  first call `query_scene_objects(query="...")` to obtain candidate list and each candidate's `instance_id`, `distance`,
↪  `container`, `is_on_screen`, then explicitly fill the corresponding `instance_id` in subsequent actions (for example
↪  `auto_navigation(target="Cutting Board", target_instance_id="...")` or `instance_id` parameters in other actions). It is
↪  forbidden to let the executor "automatically pick the nearest" for disambiguation.
8.1) If `blocking=instance_disambiguation` appears in previous-step feedback: this is a blocking failure, and your next plan
↪  must start with "select and retry" as the first step:
- select one `instance_id` from feedback `candidates` (can use strategies like nearer distance, container=Scene,
↪  is_on_screen=true)
- immediately retry the same skill/action, and fill the selected instance_id into the corresponding `*_instance_id` parameter
- do not insert any other step before "select and retry" (including another auto_navigation to elsewhere)
9) Navigate before interaction: before executing interactions like pick/place/cut/pour/sprinkle/toggle/open on an item, you
↪  must first use `auto_navigation(...)` to navigate to that item (or its Source/Stack) and face it, so it is within
↪  interactable distance, then execute the interaction action.
9.1) Occlusion constraint for interaction: interaction is affected by line-of-sight occlusion, and the environment tends to
↪  interact with the foremost visible object near the center of view. Therefore, navigation alone does not guarantee the
↪  target is interactable. If the target is still partially or fully blocked by another visible object after navigation, the
↪  blocking object may intercept the interaction. Before `pick_up` / `put_down` / cutting / pouring / sprinkling / toggling
↪  / opening, you must ensure the intended target is visually exposed and is the frontmost relevant object near the center of
↪  view; if not, first change viewpoint/aim, reposition, or move/remove the blocker, then retry the interaction.
9.2) Visual confirmation rule for interaction: treat the screen center as the effective interaction ray. Before committing an
↪  interaction, verify that the intended target itself is the object visually highlighted at the center (for example gold
↪  outline / glow / hover highlight). If the highlight appears on another object, or the intended target is visible but not
↪  highlighted, assume the target is occluded or not actually selected; do not force the interaction. Re-center the camera,
↪  reduce occlusion, or move interfering items first.
10) Refrigerator container rule (choose door/choose drawer + state checks): if `container` of the target item returned by
↪  `query_scene_objects` is Fridge/Refrigerator (or Chinese "\u51b0\u7bb1"), before pickup/interaction you must first open
↪  door and, if necessary, open drawer, then continue subsequent operations. More robust flow:
    - (a) Use `query_scene_objects(query="<item>")` to get the target item's
    ↪  `position`/`distance`/`is_on_screen`/`instance_id`.
    - (b) Use `query_scene_objects(query="Refrigerator Left Door")`, `query_scene_objects(query="Refrigerator Right Door")`,
    ↪  `query_scene_objects(query="Refrigerator Left Drawer")`, `query_scene_objects(query="Refrigerator Right Drawer")` to
    ↪  get door/drawer candidates' `position` and `is_open` (and `open_angle`).
    - (c) Choose the side (Left/Right) closer to the target item: compare target item position with Left/Right Door/Drawer
    ↪  positions (or directly use candidates' distance), and prioritize opening the nearer side.
    - (d) State check: if `is_open=false` for the selected Door/Drawer, you must first
    ↪  `auto_navigation(target="<Door/Drawer>", target_instance_id=...)`, then use `open_door()` for a Door and
    ↪  `open_drawer()` for a Drawer; if `is_open=true`, do not repeatedly open.
```

```
    - (e) Verify interactability: after opening door/drawer, immediately call `query_scene_objects(query="<item>")` again to
    ↪  confirm `is_on_screen=true` and `distance` has entered interactable range; otherwise, try opening the other side
    ↪  door/drawer and verify again.
    - (f) Drawer-priority items: the following items usually require "open door then open drawer" to obtain: Lemon, Lime,
    ↪  Radish, Mozzarella, Egg, Strawberry, Brussels sprouts (may also be written as Brusseis sprouts), Asparagus. For these
    ↪  items, execute Door + Drawer by default, and use checks in (d)(e) to verify door/drawer opened state.
11) When item does not exist, prioritize searching in acquisition list (Source/Stack whitelist, English names). Prioritize
↪  `auto_navigation` to corresponding Source/Stack and obtain a new item instance via interaction; only consider
↪  `gui_buy_new_item(...)` when no Source/Stack exists.
11.1) If `query_scene_objects(query="<item>")` returns empty, continue querying `<item> Source` / `<item> Stack`; if still
↪  empty (2 consecutive empty results), stop repeated querying and directly use `gui_buy_new_item(item="<item>")` to buy
↪  from the store. Here, "2 consecutive empty results" are merged by "same item base name": remove `Source` / `Stack` suffix
↪  (or equivalent spawn-point marker) in query to get base name; same base name counts as consecutive empty for the same
↪  item. This counter is not interrupted by `list_supported_items` / `auto_navigation` / failure retry; once 2 empty results
↪  are reached, the next step must immediately be `gui_buy_new_item(...)`.
11.2) Purchase flow must use `gui_buy_new_item(item="<item>")` (built-in navigation + ordering); in Carton Box/store UI,
↪  manually clicking/scrolling/searching to purchase is forbidden.
11.3) For any "shopping/restocking", do not open or operate Carton Box; directly call `gui_buy_new_item(...)`. Carton Box is
↪  only for pick/place of already-existing items, not for purchase flow.
11.4) Plate/tray/bowl categories (e.g., Bake Tray, Large/Small/Deep/Square Plate, Plate, Casserole, Bowl) are forbidden to be
↪  purchased via `gui_buy_new_item(...)`; they must be obtained from corresponding `X Stack` (first `query_scene_objects("<X
↪  Stack>")`, then `auto_navigation + pick_up`).
12) Entries with positive instance_id are not "unique instances": in realtime_products.json / query_scene_objects results, if
↪  an entry has positive `instance_id`, it usually indicates an "item acquisition/spawn point" (not a uniquely trackable
↪  single item instance). For such entries, prioritize treating them as Source/Stack for navigation and interaction to
↪  obtain a new instance; do not treat them as a unique item already in hand.
13) Do not rely only on the "visible items in view" list: percept/visible_items only reflect items currently visible on
↪  screen, not the full scene inventory. For items required by cooking goals but currently not visible, you must use
↪  `query_scene_objects(query="...")` for global scene retrieval to get `instance_id/position/container/distance`, then
↪  `auto_navigation(..., target_instance_id=...)` for interaction/acquisition/operation.
13.1) Repeatedly using `auto_perception` to "find items" is forbidden: if the current view does not include the target item,
↪  you may not call `auto_perception` again for repeated perception; you must switch to `query_scene_objects(query="<item>")`
↪  for global retrieval, and if needed search `<item> Source/Stack` or directly `gui_buy_new_item(item="<item>")`.
14.1) For final submission/evaluation, only a valid submission vessel or a directly submittable tart may be directly
↪  submitted: pots, plates, bowls, dishes, casseroles, `Food Processor Container`, and `Tart` count. For cookware with `pan`
↪  in the name, only `PAELLA pan` counts as a valid final submission vessel; other `*pan*` containers do not count. `Bake
↪  Tray` does not count as a valid final submission vessel.
15) Placement point selection: when executing put_down, you must choose an "empty placement point"; if an item was previously
↪  placed on a placement point, the next placement must switch to another empty point to avoid repeatedly occupying the same
↪  location.
16) Planning output constraints:
    - Do not invent non-existent actions or skills; only use names and parameters defined in Allowed actions/skills.
    - Output only one final plan; do not output multiple alternatives.
    - Do not use abstract verbs (such as move/move_to/continue); exact action/skill names must be used.
    - Parameters must strictly match the corresponding schema (no extra fields).
    - Each step expectation must include the target object/tool and a verifiable result.
17) Interaction loop constraints (applicable only to PE / planner_executor framework; not applicable to other frameworks):
    - The executor runs in `action_list` order.
    - If any step fails, the current round stops; next round starts replanning from the failure point, and do not repeat
    ↪  already successful steps.
    - If feedback contains `blocking=instance_disambiguation`, in the next step you must first choose a unique `instance_id`
    ↪  from `candidates` and retry the same skill (using specified `instance_arg`), then do other actions.
18) Stove placement point constraint: only pot-like containers (Pot/Small Pot/Big Pot/Deep Pot, etc.) can be placed on Stove
↪  platform points; other containers should be placed on platforms like Countertop/Side Table/Plate.
19) Stove switch mapping constraint: Stove Switch N only controls Stove Place Point N (same index). When turning fire on/off
↪  for a specific stove place point, you must navigate to and toggle the switch with the same index.
20) Liquid bottle constraint: a liquid bottle cannot be used as the pour target container (do not pour into a liquid bottle);
↪  however, you may hold a liquid bottle and pour into other containers (pot/bowl/plate, etc.).
20.1) When the held object can trigger a persistent interaction mode, and the current target is a valid trigger target for
↪  that mode, interacting toward that target (including trying to place the held object onto that target area, or explicitly
↪  calling the corresponding `enter_*_mode`) must be interpreted and planned as entering that mode rather than as an
↪  ordinary placement. This includes not only pouring mode, but also sprinkling mode, cutting mode, mixing mode, flipping
↪  mode, `faucet_filling_mode`, etc. If the real intent is to normally put the held item down, the target must be a platform
↪  placement point rather than a target that would trigger one of these modes.
20.2) Mode-trigger precedence and object interaction logic constraint: when holding an object that can trigger a special
↪  interaction mode, the effect of `put_down` depends on what the cursor is aimed at. If the aimed target is a valid
↪  interaction target for the held object, `put_down` can enter the corresponding mode instead of placing the object down.
↪  Planning must therefore avoid treating such targets as normal placement points unless entering that mode is intended. For
↪  example, aiming a pan or liquid container at another container can trigger pouring, and aiming a knife at a cutting board
↪  or ingredient can trigger cutting.
20.2.1) Cutting target constraint: a knife cannot reliably cut items while they are inside a `Bake Tray`; cutting should
↪  preferably be performed only on a `Cutting Board` or on an ordinary flat platform placement surface (such as `Countertop`
↪  / `Side Table`). Do not treat `Bake Tray` as a cutting base, and do not plan to cut ingredients through or inside a `Bake
↪  Tray`.
20.3) Repair constraint: if a cookware/tool is found to be damaged and unusable, you must hold the Repair Phone, navigate/aim
↪  at the damaged target, and execute `repair` to restore it before continuing to use that object.
20.4) Repair Phone placement constraint: the Repair Phone may only be released in two ways: place it onto `Phone Holder`, or
↪  remove it from hand via `throw_away`. Do not place the Repair Phone on ordinary placement points.
21) Query-to-action constraint: `query_scene_objects(...)` is only an information-gathering step and does not count as
↪  meaningful physical progress or scoring progress by itself. After one or more successful queries have already provided
↪  the candidate instance_ids / availability information needed for the current subtask, the next planning round should
↪  prioritize a concrete physical operation (for example `auto_navigation`, `pick_up`, `put_down`, `toggle_switch`, `cut`,
↪  `pour`, `sprinkle`) or a concrete instance-binding retry. Repeated querying without transitioning to physical interaction
↪  is forbidden unless the previous query result was empty, insufficient, stale, or still missing a required object/category.
21.1) Score-first assembly constraint: when constructing a dish inside a container or serving vessel, first add the
↪  recipe-defining core components into the chosen final vessel, and only then spend steps on secondary components,
↪  seasoning, and small liquid adjustments according to recipe order. This does not justify repeatedly fetching, staging, or
↪  rearranging extra empty containers.
```

21.2) If the current final vessel still lacks a core dish-defining component, prioritize acquiring and adding that component before low-value polish actions such as repeated tiny pours, extra sprinkles, garnish-like adjustments, speculative micro-optimization, or more empty-container shuffling.
21.3) Empty-container movement constraint: picking up, moving, and placing an empty Plate / Deep Plate / Bowl / Pot / Pan is setup only. It is allowed when the current subtask immediately needs that vessel, but it should not be treated as recipe completion progress or as a substitute for ingredient processing, cooking, transfer, or assembly.
21.4) Duplicate-vessel constraint: once a suitable serving vessel or working container of the needed role has been selected and remains available, do not fetch or stage another same-role empty vessel unless the current one is lost, blocked, contaminated, wrong for the recipe, or otherwise unusable.

Step-budget and scoring rule:
You are operating under a hard execution deadline. The environment will force submission starting at step 200, and may take over submission behavior within the next 30 steps.
Treat the countdown / remaining-step status as a first-class decision signal. Do not assume unlimited time or future recovery opportunities.

Budget-aware policy:
- Early budget phase: gather key recipe items, set up cookware/serving vessels, and make progress on core recipe components.
- Mid budget phase: reduce redundant querying, long detours, and low-value local repairs; prioritize the highest-value missing recipe components.
- Late budget phase: prioritize final assembly, plating, and getting into a state that is likely scorable if forced submission happens soon.
- Final budget phase: maximize expected score under the remaining steps; avoid long recovery loops, repeated object shuffling, and speculative retries that are unlikely to finish before forced submission.

Submission discipline under deadline:
- Do NOT behave as if there is no deadline.
- Do NOT spend many steps on unrelated exploration when the countdown is low.
- Do NOT enter repeated repair/retry loops near the deadline unless they directly improve the likely submitted result.
- Before any major detour, ask whether it is worth the remaining step budget.
- Reaching the force-submit threshold does NOT by itself mean immediate submit is optimal; use the remaining window first for small, high-value improvements when they clearly increase likely score.
- Only submit early inside the forced window when the dish is already in a plausible final serving state, or when the remaining steps are too few to materially improve score.
- Minimal submit gate: before submit, quickly check final object, serving vessel, and core recipe components. If any of these is still doubtful, use the remaining window for the highest-value fix instead of submitting immediately.

Scoring reminder:
- Dish score is determined mainly by the final submitted container contents and their state: correct ingredients/components, correct seasoning, correct processing/cooking operations, and appropriate doneness/temperature help; wrong or unrelated contents can reduce score.
- Under limited steps, prioritize recipe-required ingredients and the highest-value processing/assembly steps that most improve the final submitted result.
- Before spending many steps on seasoning or small liquid adjustments, first ensure the final container already contains the main recipe-defining components; building that core dish skeleton usually improves score more than early polishing.
- If key dish-defining components are still missing, avoid low-value garnish/polish loops and use the remaining budget on the highest-value missing core component first.

Strategy notes (planning guidance and heuristics):
Strategy Notes

Purpose: describe planning guidance, common-sense knowledge, experience notes, recovery suggestions, and score-oriented strategy, as a static fragment in the Brain prompt.

[Notes]
1) If `is_held=true` in agent_state, it means an item is currently held. You must first navigate to a placement platform point (such as Plate / Side Table / Countertop and other placeable points), then execute put-down/placement, and then re-navigate to pick up other items.
2) When you need to execute an item pick-up action, if an item is currently held, you should first put down the item in hand, then proceed. Or place it properly after use.
2.1) Handle "held-item occlusion": when `agent_state.is_held=true` and the held object (especially bottles/pots/large containers) blocks the crosshair/front view, prefer putting it down or switching to an unoccluded viewpoint before the next pick-up. Do not repeatedly retry `pick_up` under obvious occlusion.
2.2) If pick-up fails (for example `pick_up_post_check_failed:not_holding_after_click` or `post_check_failed:pick_up_not_holding_item`), it is recommended to use visual signals for recovery: re-check `is_on_screen`, centering (`screen_x/screen_y`), and distance, then prefer "re-navigate + micro-view alignment + retry" instead of blind repeated clicks.
2.3) Avoid global staging. Do not gather or spread out many future-use tools/containers just to keep hands free. Only stage items that are required for the current recipe subtask or the next immediate interaction. Empty container staging is setup only, not scorable progress by itself.
2.4) If `task_progress`, precondition feedback, or visual anomaly feedback indicates a current subtask, blocker, or warning, prioritize resolving that concrete issue before convenience actions such as broad staging or speculative preparation for later recipe steps.
2.5) Once a required bowl/plate/pot/pan for the current subtask has already been acquired or staged, the next planning rounds should normally switch to ingredient acquisition, cutting, pouring, cooking, transfer, or submission-related progress. Do not keep acquiring or rearranging same-role empty containers unless the current one is unusable, missing, wrong, or blocked.
3) Do not only pick items: after obtaining ingredients, they must be processed according to the recipe (cut/fry/bake/mix, etc.), and after operations, outputs should be consolidated into the same chosen serving vessel. Keep container choice stable; avoid turning one chosen vessel into many staged duplicates.
4) Check `agent_state` before actions: if currently in pouring/sprinkling/cutting or other modes, either execute the corresponding skill action, or exit that mode before other operations; determine whether crouching/standing is needed based on posture. Note that `auto_navigation` already automatically includes crouch/stand decisions according to item height position.
5) Container selection must remain consistent: once a certain container/plate is determined as the carrying container for the current process, subsequent related operations should keep using this container; do not switch with view changes. Only reselect and update `active_containers` (recently used list) and `active_container` (current container) when the container is unavailable/lost or task requirements need extra tools.
6) The agent can carry multiple items at once by holding a container, but too many items can easily drop, so be careful.

```
7) Failure recovery priority: retry (small parameter perturbation) -> switch skill/switch path -> fallback to safe state ->
↪  optional reset.
8) If the selected instance_id causes navigation/interaction failure for 2 consecutive attempts: do not loop retrying the
↪  same instance_id; choose another instance_id in `candidates` and try again.
9) If feedback shows `error=invalid_target_name`: do not repeat the same target string; prioritize `mapping_suggestions` in
↪  feedback, or call `list_supported_items()` first to find available canonical target names and then call
↪  `auto_navigation(target=...)`.
10) Acquisition list: Chili Pepper Source; Cucumber Source; Tomato Source; Jalapeno Source; Lemon Source; Shelf; Burger Bun
↪  Source; Bun Bottom Source; Bun Top Source; Apple Source; Red Potato Source; Onion Source; Garlic Source; Cob of Corn
↪  Source; Broccoli Source; Bread Source; Parsley Root Source; Button Mushroom Source; Red Bell Pepper Source; Beetroot
↪  Source; Eggplant Source; Green Bell Pepper Source; Yellow Bell Pepper Source; Potato Source; Whole Ginger Source; Red
↪  Cabbage Source; Zucchini Source; Scallion Source; Horseradish Source; Parsnip Source; Orange Source; Carrot Source;
↪  Pumpkin Source; Small Plate Stack; Plate Stack; Square Plate Stack; Deep Plate Stack; Plate Stack; Casserole Stack; Bowl
↪  Stack; Large Plate Stack.
11) If 5 consecutive steps are `auto_perception` / `query_scene_objects` / `list_supported_items` (any combination), the next
↪  step must choose a "real physical action" to execute (such as `auto_navigation` / `pick_up` / `open_door` / `place` /
↪  `pour` / `sprinkle` / `cut` / `toggle`, etc.); continuing perception/retrieval loops is forbidden. If necessary, directly
↪  use `gui_buy_new_item(...)`.
12) Score priority: final score mainly depends on the final submitted container content and its state, including seasoning
↪  ratio, heat-treatment method and duration, food-processing procedure (e.g., cutting/mixing/cooking flow), and final
↪  temperature. If the final submitted container has no scorable content, it usually gets no score; if it contains relevant
↪  and partially correct content for the target dish, it may still receive partial score items even when not fully matched.
12.1) Build the scorable dish skeleton early: before spending many steps on seasoning tweaks or small liquid adjustments,
↪  prioritize adding the primary recipe-defining components into the final container/serving vessel (for example the main
↪  protein, key produce, or required major subcomponent). These core components usually increase score more than
↪  fine-grained seasoning/liquid optimization when they are still missing.
12.2) If the current final container still lacks one or more core dish-defining components, do not over-invest in low-value
↪  polish (extra sprinkles, tiny pours, repeated micro-adjustments). First make the dish clearly identifiable as the target
↪  recipe, then refine seasoning, temperature, and presentation.

Supported items (static snapshot):
Supported items (static snapshot from data mapping):
{
  "query": "",
  "mapping_error": null,
  "items": [
    {
      "name_en": "Barrel of dark beer",
      "name_cn": "\u6876\u9ed1\u5564\u9152",
      "category": "background"
    },
    {
[... omitted middle entries from static item catalog ...]
      "category": "Top Oven Door",
      "name_template": "Top Oven Door",
      "index_ranges": null
    },
    {
      "category": "Top Oven Switch",
      "name_template": "Top Oven Switch",
      "index_ranges": null
    }
  ]
}

Allowed actions/skills (authoritative schemas):
[
  {
    "type": "function",
    "function": {
      "name": "action__Horizontal_movement",
      "description": "Rotation and spatial-control action. Move the object horizontally in space.",
      "parameters": {
        "type": "object",
        "properties": {
          "x": {
            "type": "string"
          },
          "y": {
            "type": "string"
          }
        },
        "required": [
[... omitted middle action and skill schemas ...]
- `type` must be exactly `action` or `skill`.
- `name` must be the exact action or skill name from the schemas above.
- `args` must be a JSON object using the actual parameter names for that action or skill; use `{}` when no args are needed.
{count_constraint}

Response Format (JSON only):
{
  "high_level_id": "Hk",
  "goal": "one sentence goal",
  "explanation": null | "why last attempt failed",
  "thoughts": "brief planning rationale",
  "action_list": [
{response_format_action_example_1},
{response_format_action_example_2},
{response_format_action_example_3}{response_format_action_examples_extra}
```

```
  ]
}

USER:
Goal: Follow recipe

Full recipe:
Dish: Barbecue T-bone Steak with Potatoes and a Hot Corn on the Cob
Ingredients:
- Onion
- Garlic
- Cob of Corn
- Chili Pepper
- T-bone Steak
- Red Potato
- Water
- Barbecue Sauce
- Clarified Butter
- Salt
- Black Pepper
- Sweet Bell Pepper, powder

Recipe sections:
[Barbecue T-bone Steak]
Take T-bone Steak [680g]. Season with: Salt [7g], Black Pepper [7g], Sweet Bell Pepper, powder [10g]. Drizzle in Barbecue
↪  Sauce [30ml]. Grill for 70s each side. Transfer onto a Plate. Serve hot.

[Baked Red Potatoes and Garlic]
Add to a Pot: Water [1000ml], Salt [20g], Red Potato [300g]. Boil for 20s. Cut Onion [120g] into Pieces [20g]. Add to a Baking
↪  Tray: Onion. Bake for 20s. Add: Garlic [50g], Red Potato. Bake for 40s. Transfer onto a Plate. Serve hot.

[Corn on the Cob with Chilli Butter]
Add to a Pot: Water [1000ml], Salt [20g], Cob of Corn [180g]. Boil for 100s. Cut Chili Pepper [30g] into Halves [15g]. Add to
↪  a Pan: Clarified Butter [30ml], Chili Pepper. Fry for 15s. Take Cob of Corn. Drizzle in Mixture [30ml]. Transfer
↪  everything onto a Plate. Serve cold.

Task execution context:
episode_mode=single_active_dish
active_dish_ids_for_this_prompt=[100]
active_dish_names_for_this_prompt=["Barbecue T-bone Steak with Potatoes and a Hot Corn on the Cob"]
final_submission_count_for_this_prompt=1
submission_scope=current_active_dish_only
interpretation_note=The current recipe belongs to the active dish set listed above. Multiple components or multiple serve
↪  hot/cold lines still belong to one dish and one final submission unless this execution context explicitly lists multiple
↪  active dishes.

Placement occupancy (oracle snapshot):
{
  "updated_at": "2026-05-01T21:05:36.838571+00:00",
  "kind": "platform_point",
  "note": "Each platform category is derived by removing the trailing '-<index>' from name_en. IMPORTANT: the trailing number
  ↪  in name_en is an index, NOT the instance_id. Use the `instance_id` field in realtime_products.json for disambiguation
  ↪  (same as normal items). This file only summarizes platform categories and valid index ranges.",
  "free-place-point": [
    {
      "category": "Large Table Center",
      "name_template": "Large Table Center-{index}",
      "index_ranges": [
        [
          1,
          18
        ],
        [
          21,
[... omitted middle occupancy entries ...]
      "name": "Ladle 50 ml",
      "instance_id": -1046350,
      "places": [
        {
          "place_point": "Top Shelf 2 (2-Shelf)-9",
          "reason": "plane_distance_match"
        },
        {
          "place_point": "Top Shelf 2 (2-Shelf)-12",
          "reason": "plane_distance_match"
        }
      ]
    }
  }
}

Observation:
- A screenshot image is provided as the input.
Oracle visible objects (count=68, sorted by distance):
Oracle screen coordinates use the original realtime_products screen size; the VLM image is compressed to approx (768, 432).
- name='Side Table-26' instance_id=-1327511271 kind='platform_point' distance=1.00 on_screen=True is_held=False
↪  screen=(807.8, 596.9) vlm_screen=(387.7, 286.5) screen_uv=(0.505, 0.663) pos=(1.570, 0.917, 0.495)
- name='Side Table-25' instance_id=-1327511272 kind='platform_point' distance=1.03 on_screen=True is_held=False
↪  screen=(634.3, 595.1) vlm_screen=(304.5, 285.6) screen_uv=(0.396, 0.661) pos=(1.570, 0.917, 0.275)
```

```
- name='Side Table-27' instance_id=-1327511270 kind='platform_point' distance=1.03 on_screen=True is_held=False
↪  screen=(981.7, 598.6) vlm_screen=(471.2, 287.3) screen_uv=(0.614, 0.665) pos=(1.570, 0.917, 0.715)
- name='garlic' instance_id=-1119268 kind='product' distance=1.06 on_screen=True is_held=False screen=(782.2, 455.0)
↪  vlm_screen=(375.5, 218.4) screen_uv=(0.489, 0.506) pos=(1.382, 0.958, 0.459)
- name='Onion' instance_id=-1118924 kind='product' distance=1.07 on_screen=True is_held=False screen=(770.6, 422.7)
↪  vlm_screen=(369.9, 202.9) screen_uv=(0.482, 0.470) pos=(1.341, 0.976, 0.442)
- name='Red Potato' instance_id=-1118450 kind='product' distance=1.07 on_screen=True is_held=False screen=(813.5, 417.0)
↪  vlm_screen=(390.5, 200.2) screen_uv=(0.508, 0.463) pos=(1.335, 0.982, 0.501)
- name='Side Table-24' instance_id=-1327511273 kind='platform_point' distance=1.09 on_screen=True is_held=False
↪  screen=(461.3, 593.3) vlm_screen=(221.4, 284.8) screen_uv=(0.288, 0.659) pos=(1.570, 0.917, 0.055)
- name='Small Pot' instance_id=-1038340 kind='container' distance=1.09 on_screen=True is_held=False screen=(799.9, 447.2)
↪  vlm_screen=(384.0, 214.7) screen_uv=(0.500, 0.497) pos=(1.356, 0.940, 0.483)
- name='Side Table-28' instance_id=-1327511269 kind='platform_point' distance=1.10 on_screen=True is_held=False
↪  screen=(1156.2, 600.4) vlm_screen=(555.0, 288.2) screen_uv=(0.723, 0.667) pos=(1.570, 0.917, 0.935)
- name='Side Table-19' instance_id=-1327511278 kind='platform_point' distance=1.11 on_screen=True is_held=False
↪  screen=(808.6, 452.6) vlm_screen=(388.1, 217.2) screen_uv=(0.505, 0.503) pos=(1.350, 0.917, 0.495)
[... omitted middle visible-object observations ...]
- name='Pan' instance_id=-1045660 kind='container' distance=2.11 on_screen=True is_held=False screen=(140.7, 372.1)
↪  vlm_screen=(67.5, 178.6) screen_uv=(0.088, 0.413) pos=(0.950, 0.597, -0.882)
- name='Large Table Center-38' instance_id=-1324339259 kind='platform_point' distance=2.15 on_screen=True is_held=False
↪  screen=(60.4, 151.7) vlm_screen=(29.0, 72.8) screen_uv=(0.038, 0.169) pos=(0.667, 0.916, -0.950)
- name='Large Table Center-35' instance_id=-1324339262 kind='platform_point' distance=2.16 on_screen=True is_held=False
↪  screen=(222.6, 87.6) vlm_screen=(106.8, 42.0) screen_uv=(0.139, 0.097) pos=(0.447, 0.916, -0.730)
- name='Large Table Center-32' instance_id=-1324339265 kind='platform_point' distance=2.21 on_screen=True is_held=False
↪  screen=(362.2, 32.4) vlm_screen=(173.9, 15.6) screen_uv=(0.226, 0.036) pos=(0.227, 0.916, -0.510)
- name='Large Table Center-34' instance_id=-1324339263 kind='platform_point' distance=2.29 on_screen=True is_held=False
↪  screen=(117.6, 87.0) vlm_screen=(56.4, 41.8) screen_uv=(0.073, 0.097) pos=(0.447, 0.916, -0.950)
- name='Large Table Center-31' instance_id=-1324339266 kind='platform_point' distance=2.32 on_screen=True is_held=False
↪  screen=(264.4, 31.9) vlm_screen=(126.9, 15.3) screen_uv=(0.165, 0.035) pos=(0.227, 0.916, -0.730)
- name='Large Table Center-33' instance_id=-1324339264 kind='platform_point' distance=2.43 on_screen=True is_held=False
↪  screen=(12.7, 86.3) vlm_screen=(6.1, 41.4) screen_uv=(0.008, 0.096) pos=(0.447, 0.916, -1.170)
- name='Large Table Center-30' instance_id=-1324339267 kind='platform_point' distance=2.44 on_screen=True is_held=False
↪  screen=(166.7, 31.3) vlm_screen=(80.0, 15.0) screen_uv=(0.104, 0.035) pos=(0.227, 0.916, -0.950)
- name='Large Table Center-29' instance_id=-1324339268 kind='platform_point' distance=2.58 on_screen=True is_held=False
↪  screen=(69.2, 30.7) vlm_screen=(33.2, 14.7) screen_uv=(0.043, 0.034) pos=(0.227, 0.916, -1.170)
- name='Grill Pan' instance_id=-1044574 kind='container' distance=2.73 on_screen=True is_held=False screen=(305.5, 301.6)
↪  vlm_screen=(146.6, 144.8) screen_uv=(0.191, 0.335) pos=(0.352, 0.197, -0.971)
- name='Bucket' instance_id=-1039076 kind='container' distance=3.03 on_screen=True is_held=False screen=(1091.0, 165.8)
↪  vlm_screen=(523.7, 79.6) screen_uv=(0.682, 0.184) pos=(-0.359, 0.161, 1.468)
- name='Door Right' instance_id=2971742 kind='tool_point' distance=3.13 on_screen=True is_held=False screen=(256.3, 67.8)
↪  vlm_screen=(123.0, 32.5) screen_uv=(0.160, 0.075) pos=(-0.317, 0.518, -1.199)
- name='Small Pot' instance_id=-1039970 kind='container' distance=3.70 on_screen=True is_held=False screen=(653.2, 26.1)
↪  vlm_screen=(313.5, 12.5) screen_uv=(0.408, 0.029) pos=(-1.262, 0.161, -0.149)

Agent state snapshot:
{
  "is_held": null,
  "held_item": null,
  "item_kind": "unknown",
  "mode": "pour_mode (global_realtime_fallback)",
  "posture": "standing",
  "held_item_weight": null,
  "active_container": "Bake Tray",
  "active_container_instance_id": -1037380,
  "active_containers": [
    {
      "name": "Deep Plate",
      "instance_id": -1081682
    },
    {
      "name": "Deep Plate",
      "instance_id": -1085148
    },
    {
      "name": "Plate",
      "instance_id": -1088388
    },
    {
      "name": "Plate",
      "instance_id": -1090360
    },
    {
      "name": "Red Potato",
      "instance_id": -1118450
    },
    {
      "name": "Small Pot",
      "instance_id": -1038340
    },
    {
      "name": "Bake Tray",
      "instance_id": -1037380
    }
  ],
  "timers": [],
  "timer_events": [
    {
      "id": "timer_1777668773232116700",
      "status": "done",
```

```
      "at_ts": 1777668793.2927475,
      "step_id": 170
    }
  ],
  "last_updated": "2026-05-01T21:05:39Z",
  "held_item_name": null,
  "held_item_instance_id": null,
  "active_container_evidence": [
    {
      "name": "Deep Plate",
      "instance_id": -1081682,
      "score": 4.5,
      "add_count": 0,
      "remove_count": 0,
      "touch_count": 4,
      "last_touch_step": 41
    },
    {
      "name": "Deep Plate",
      "instance_id": -1085148,
      "score": 1.5,
      "add_count": 0,
      "remove_count": 0,
      "touch_count": 1,
      "last_touch_step": 43
    },
    {
      "name": "Plate",
      "instance_id": -1088388,
      "score": 3.0,
      "add_count": 0,
      "remove_count": 0,
      "touch_count": 2,
      "last_touch_step": 52
    },
    {
      "name": "Plate",
      "instance_id": -1090360,
      "score": 15.75,
      "add_count": 0,
      "remove_count": 0,
      "touch_count": 17,
      "last_touch_step": 78
    },
    {
      "name": "Red Potato",
      "instance_id": -1118450,
      "score": 51.0,
      "add_count": 12,
      "remove_count": 0,
      "touch_count": 15,
      "last_touch_step": 143,
      "last_add_step": 143
    },
    {
      "name": "Small Pot",
      "instance_id": -1038340,
      "score": 36.0,
      "add_count": 0,
      "remove_count": 0,
      "touch_count": 34,
      "last_touch_step": 188
    },
    {
      "name": "Bake Tray",
      "instance_id": -1037380,
      "score": 2.25,
      "add_count": 0,
      "remove_count": 0,
      "touch_count": 2,
      "last_touch_step": 192
    }
  ],
  "is_held_note": "unknown_in_mode_from_game_feed"
}

Resume state snapshot:
{
  "status": "running",
  "reason": "",
  "dish_id": 100,
  "last_step_id": 193,
  "last_completed_step_id": 193,
  "last_physical_step_id": 193,
  "updated_at": "2026-05-02T05:05:38"
}

Countdown status:
- episode_step=194
- logical_committed_step=193
```

```
- timer_running: none
- latest_timer_event: id='timer_1777668773232116700' status='done' at_ts=1777668793.2927475

Feedback from last execution:
last_step_id=H1.A193
type=action name=put_down
result=failure
error=put_down_post_check_failed:interaction_mode_still_active; agent_mode='pour_mode (global_realtime_fallback)'
blocking=put_down_failed_place_invalid
instruction=Put-down did not release the held item. This place is likely occupied or invalid for current item. Navigate to another empty place point and retry put_down.
hint=Use put_place_occupancy snapshot and choose a different free point than the last attempt.
return: {"success":false,"error":"put_down_post_check_failed:interaction_mode_still_active","hint":"choose_another_empty_place_point_and_retry_put_down","precheck":{"held_name_en":"Bake Tray","held_name_cn":"\u70d8\u70e4\u6258\u76d8","held_kind":"container","is_held":true,"is_pouring_mode":false,"is_sprinkle_mode":false,"is_cutting_mode":false,"is_mixing_mode":false,"is_flip_mode":false,"pouring_mode_item_name":"","cutting_mode_item_name":"","sprinkle_mode_item_name":"","mixing_mode_item_name":"","flip_mode_item_name":"","dominant_mode":"","dominant_mode_item_name":""},"postcheck":{"held_name_en":"","held_name_cn":"","held_kind":"","is_held":false,"is_pouring_mode":true,"is_sprinkle_mode":false,"is_cutting_mode":false,"is_mixing_mode":false,"is_flip_mode":false,"pouring_mode_item_name":"Bake Tray","cutting_mode_item_name":"","sprinkle_mode_item_name":"","mixing_mode_item_name":"","flip_mode_item_name":"","dominant_mode":"pour","dominant_mode_item_name":"Bake Tray"},"retry_attempts":[]}

Cross-step query memory:
Recent query cache (cross-step):
Note: each entry includes `source_step` and `query_time`; older entries may be stale after scene changes.
- source_step=10 query_time='2026-05-02T03:43:25' query='Salt' results_len=1 top=[{"name_en":"Salt","name_cn":"\u76d0","kind":"spices","instance_id":-1025762,"distance":10.11,"is_on_screen":false,"container":"Scene","position":{"x":7.403,"y":1.796,"z":-3.833}}]
- source_step=9 query_time='2026-05-02T03:43:24' query='Cutting Board' results_len=2 top=[{"name_en":"Cutting Board","name_cn":"\u5207\u83dc\u677f","kind":"tools","instance_id":-1041298,"distance":2.79,"is_on_screen":false,"container":"Scene","position":{"x":-1.145,"y":0.94,"z":0.838}},{"name_en":"Cutting Board","name_cn":"\u5207\u83dc\u677f","kind":"tools","instance_id":-1039628,"distance":4.28,"is_on_screen":false,"container":"Scene","position":{"x":0.867,"y":0.94,"z":-0.95}}]
- source_step=8 query_time='2026-05-02T03:43:23' query='Knife' results_len=10 top=[{"name_en":"Knife Block-6","name_cn":"\u5200\u67b6-6","kind":"tool_point","instance_id":2965176,"distance":3.98,"is_on_screen":false,"container":"Scene","position":{"x":1.288,"y":1.013,"z":-0.543}},{"name_en":"Knife Block-5","name_cn":"\u5200\u67b6-5","kind":"tool_point","instance_id":2966562,"distance":4.01,"is_on_screen":false,"container":"Scene","position":{"x":1.305,"y":1.033,"z":-0.574}},{"name_en":"Knife Block-4","name_cn":"\u5200\u67b6-4","kind":"tool_point","instance_id":2968380,"distance":4.04,"is_on_screen":false,"container":"Scene","position":{"x":1.322,"y":1.045,"z":-0.603}},{"name_en":"Knife","name_cn":"\u5200","kind":"tool","instance_id":-1031604,"distance":4.05,"is_on_screen":false,"container":"Scene","position":{"x":1.307,"y":1.138,"z":-0.632}},{"name_en":"Knife holder","name_cn":"\u5200\u67b6\u672c\u4f53","kind":"tool","instance_id":2967158,"distance":4.05,"is_on_screen":false,"container":"Scene","position":{"x":1.329,"y":1.06,"z":-0.619}},{"name_en":"Knife Block-3","name_cn":"\u5200\u67b6-3","kind":"tool_point","instance_id":2967732,"distance":4.07,"is_on_screen":false,"container":"Scene","position":{"x":1.339,"y":1.055,"z":-0.632}},{"name_en":"Knife Block","name_cn":"\u5200\u67b6","kind":"tool_point","instance_id":2969606,"distance":4.07,"is_on_screen":false,"container":"Scene","position":{"x":1.329,"y":0.954,"z":-0.619}},{"name_en":"Knife Block-2","name_cn":"\u5200\u67b6-2","kind":"tool_point","instance_id":2965876,"distance":4.1,"is_on_screen":false,"container":"Scene","position":{"x":1.366,"y":1.067,"z":-0.66}},{"name_en":"Knife","name_cn":"\u5200","kind":"tool","instance_id":-1031474,"distance":4.12,"is_on_screen":false,"container":"Scene","position":{"x":1.394,"y":1.161,"z":-0.69}},{"name_en":"Knife Block-1","name_cn":"\u5200\u67b6-1","kind":"tool_point","instance_id":2967990,"distance":4.14,"is_on_screen":false,"container":"Scene","position":{"x":1.396,"y":1.073,"z":-0.692}}]
- source_step=7 query_time='2026-05-02T03:43:23' query='Corn' results_len=1 top=[{"name_en":"Cob of Corn Source","name_cn":"\u7389\u7c73\u68d2\u83b7\u53d6\u5904","kind":"spawn_point","instance_id":2965742,"distance":6.15,"is_on_screen":false,"container":"Scene","position":{"x":5.238,"y":0.949,"z":-0.142}}]
- source_step=6 query_time='2026-05-02T03:43:22' query='Potato' results_len=3 top=[{"name_en":"Red Potato Source","name_cn":"\u7ea2\u571f\u8c46\u83b7\u53d6\u5904","kind":"spawn_point","instance_id":2966572,"distance":7.06,"is_on_screen":false,"container":"Scene","position":{"x":6.205,"y":1.415,"z":-0.395}},{"name_en":"Potato Source","name_cn":"\u571f\u8c46\u83b7\u53d6\u5904","kind":"spawn_point","instance_id":2965778,"distance":7.46,"is_on_screen":false,"container":"Scene","position":{"x":6.611,"y":0.867,"z":-0.4}},{"name_en":"Potato Dumplings Box","name_cn":"\u571f\u8c46\u9762\u7403\u76d2","kind":"liquids","instance_id":-1031258,"distance":10.57,"is_on_screen":false,"container":"Scene","position":{"x":7.433,"y":0.224,"z":-4.3}}]
- source_step=5 query_time='2026-05-02T03:43:21' query='T-bone Steak' results_len=2 top=[{"name_en":"T-bone Steak","name_cn":"T\u9aa8\u725b\u6392","kind":"product","instance_id":2965402,"distance":8.33,"is_on_screen":false,"container":"Refrigerator","position":{"x":3.466,"y":1.491,"z":-4.436}},{"name_en":"T-bone Steak","name_cn":"T\u9aa8\u725b\u6392","kind":"product","instance_id":2970752,"distance":8.52,"is_on_screen":false,"container":"Refrigerator","position":{"x":3.579,"y":1.492,"z":-4.6}}]
- source_step=4 query_time='2026-05-02T03:43:21' query='Grill' results_len=8 top=[{"name_en":"Grill Pan","name_cn":"\u70e4\u76d8","kind":"container","instance_id":-1044574,"distance":4.45,"is_on_screen":false,"container":"Scene","position":{"x":0.352,"y":0.197,"z":-0.971}},{"name_en":"Grill Switch","name_cn":"\u6761\u7eb9\u70e4\u7089\u5f00\u5173","kind":"tool_point","instance_id":2971334,"distance":5.14,"is_on_screen":false,"container":"Scene","position":{"x":-4.197,"y":0.853,"z":0.563}},{"name_en":"Grill Surface-4","name_cn":"\u6761\u7eb9\u70e4\u7089\u9762\u677f-4","kind":"platform_point","instance_id":-1319173293,"distance":5.27,"is_on_screen":false,"container":"platform","position":{"x":-4.507,"y":0.903,"z":0.814}},{"name_en":"Grill Surface-5","name_cn":"\u6761\u7eb9\u70e4\u7089\u9762\u677f-5","kind":"platform_point","instance_id":-1319173292,"distance":5.38,"is_on_screen":false,"container":"platform","position":{"x":-4.507,"y":0.903,"z":0.594}},{"name_en":"Grill Surface-1","name_cn":"\u6761\u7eb9\u70e4\u7089\u9762\u677f-1","kind":"platform_point","instance_id":-1319173296,"distance":5.47,"is_on_screen":false,"container":"platform","position":{"x":-4.727,"y":0.903,"z":0.814}},{"name_en":"Grill Surface-6","name_cn":"\u6761\u7eb9\u70e4\u7089\u9762\u677f-6","kind":"platform_point","instance_id":-1319173291,"distance":5.49,"is_on_screen":false,"container":"platform","position":{"x":-4.507,"y":0.903,"z":0.374}},{"name_en":"Grill Surface-2","name_cn":"\u6761\u7eb9\u70e4\u7089\u9762\u677f-2","kind":"platform_point","instance_id":-1319173295,"distance":5.57,"is_on_screen":false,"container":"platform","position":{"x":-4.727,"y":0.903,"z":0.594}},{"name_en":"Grill Surface-3","name_cn":"\u6761\u7eb9\u70e4\u7089\u9762\u677f-3","kind":"platform_point","instance_id":-1319173294,"distance":5.67,"is_on_screen":false,"container":"platform","position":{"x":-4.727,"y":0.903,"z":0.374}}]
```

- source_step=3 query_time='2026-05-02T03:43:20' query='Pot' results_len=7 top=[{"name_en":"Small Pot","name_cn":"\u5c0f\u9505","kind":"container","instance_id":-1038340,"distance":3.63,"is_on_screen":false,"container":"Scene","position":{"x":-1.111,"y":0.161,"z":0.163}},{"name_en":"Small Pot","name_cn":"\u5c0f\u9505","kind":"container","instance_id":-1039970,"distance":3.95,"is_on_screen":false,"container":"Scene","position":{"x":-1.262,"y":0.161,"z":-0.149}},{"name_en":"Red Potato Source","name_cn":"\u7ea2\u571f\u8c46\u83b7\u53d6\u5904","kind":"spawn_point","instance_id":2966572,"distance":7.06,"is_on_screen":false,"container":"Scene","position":{"x":6.205,"y":1.415,"z":-0.395}},{"name_en":"Potato Source","name_cn":"\u571f\u8c46\u83b7\u53d6\u5904","kind":"spawn_point","instance_id":2965778,"distance":7.46,"is_on_screen":false,"container":"Scene","position":{"x":6.611,"y":0.867,"z":-0.4}},{"name_en":"Big Pot","name_cn":"\u5927\u9505","kind":"container","instance_id":-1043206,"distance":7.88,"is_on_screen":false,"container":"Scene","position":{"x":0.448,"y":0.164,"z":-4.525}},{"name_en":"Big Pot","name_cn":"\u5927\u9505","kind":"container","instance_id":-1043890,"distance":7.99,"is_on_screen":false,"container":"Scene","position":{"x":-0.135,"y":0.164,"z":-4.637}},{"name_en":"Potato Dumplings Box","name_cn":"\u571f\u8c46\u9762\u7403\u76d2","kind":"liquids","instance_id":-1031258,"distance":10.57,"is_on_screen":false,"container":"Scene","position":{"x":7.433,"y":0.224,"z":-4.3}}]
- source_step=2 query_time='2026-05-02T03:43:19' query='Plate' results_len=18 top=[{"name_en":"Square Plate Source","name_cn":"\u65b9\u76d8\u83b7\u53d6\u5904","kind":"container","instance_id":2964652,"distance":3.13,"is_on_screen":false,"container":"Scene","position":{"x":1.372,"y":0.169,"z":0.792}},{"name_en":"Large Plate Source","name_cn":"\u5927\u76d8\u83b7\u53d6\u5904","kind":"container","instance_id":2964546,"distance":3.34,"is_on_screen":false,"container":"Scene","position":{"x":1.365,"y":0.169,"z":0.517}},{"name_en":"Deep Plate Stack","name_cn":"\u6df1\u76d8\u5806","kind":"container","instance_id":2964380,"distance":3.49,"is_on_screen":false,"container":"Scene","position":{"x":1.118,"y":0.168,"z":0.234}},{"name_en":"Plate Source","name_cn":"\u76d8\u5b50\u83b7\u53d6\u5904","kind":"container","instance_id":2964374,"distance":3.56,"is_on_screen":false,"container":"Scene","position":{"x":1.366,"y":0.169,"z":0.248}},{"name_en":"Plate Source","name_cn":"\u76d8\u5b50\u83b7\u53d6\u5904","kind":"container","instance_id":2964692,"distance":3.79,"is_on_screen":false,"container":"Scene","position":{"x":1.397,"y":0.169,"z":-0.016}},{"name_en":"Large Plate Source","name_cn":"\u5927\u76d8\u83b7\u53d6\u5904","kind":"container","instance_id":2964762,"distance":3.84,"is_on_screen":false,"container":"Scene","position":{"x":1.109,"y":0.168,"z":-0.164}},{"name_en":"Small Plate Source","name_cn":"\u5c0f\u76d8\u83b7\u53d6\u5904","kind":"container","instance_id":2964510,"distance":4.01,"is_on_screen":false,"container":"Scene","position":{"x":1.431,"y":0.169,"z":-0.252}},{"name_en":"Deep Plate Stack","name_cn":"\u6df1\u76d8\u5806","kind":"container","instance_id":2964628,"distance":5.82,"is_on_screen":false,"container":"Scene","position":{"x":-4.83,"y":1.531,"z":0.15}},{"name_en":"Deep Plate Stack","name_cn":"\u6df1\u76d8\u5806","kind":"container","instance_id":2964732,"distance":6.54,"is_on_screen":false,"container":"Scene","position":{"x":-4.865,"y":1.529,"z":-1.033}},{"name_en":"Plate Stack","name_cn":"\u76d8\u5b50\u5806","kind":"container","instance_id":2964500,"distance":6.75,"is_on_screen":false,"container":"Scene","position":{"x":-4.855,"y":1.528,"z":-1.366}},{"name_en":"Small Plate Stack","name_cn":"\u5c0f\u76d8\u5806","kind":"container","instance_id":2964596,"distance":6.95,"is_on_screen":false,"container":"Scene","position":{"x":-4.847,"y":1.529,"z":-1.658}},{"name_en":"Square Plate Stack","name_cn":"\u65b9\u76d8\u5806","kind":"container","instance_id":2964606,"distance":9.38,"is_on_screen":false,"container":"Scene","position":{"x":7.576,"y":1.225,"z":-2.484}},{"name_en":"Large Plate Stack","name_cn":"\u5927\u76d8\u5806","kind":"container","instance_id":2964426,"distance":9.55,"is_on_screen":false,"container":"Scene","position":{"x":7.569,"y":1.225,"z":-2.771}},{"name_en":"Plate Stack","name_cn":"\u76d8\u5b50\u5806","kind":"container","instance_id":2964416,"distance":9.72,"is_on_screen":false,"container":"Scene","position":{"x":7.57,"y":1.224,"z":-3.04}},{"name_en":"Deep Plate Source","name_cn":"\u6df1\u76d8\u83b7\u53d6\u5904","kind":"container","instance_id":2964386,"distance":9.78,"is_on_screen":false,"container":"Scene","position":{"x":7.616,"y":0.717,"z":-3.009}},{"name_en":"Plate Stack","name_cn":"\u76d8\u5b50\u5806","kind":"container","instance_id":2964776,"distance":9.91,"is_on_screen":false,"container":"Scene","position":{"x":7.601,"y":1.225,"z":-3.304}},{"name_en":"Large Plate Stack","name_cn":"\u5927\u76d8\u5806","kind":"container","instance_id":2964794,"distance":10.03,"is_on_screen":false,"container":"Scene","position":{"x":7.607,"y":0.716,"z":-3.407}},{"name_en":"Small Plate Stack","name_cn":"\u5c0f\u76d8\u5806","kind":"container","instance_id":2964364,"distance":10.11,"is_on_screen":false,"container":"Scene","position":{"x":7.635,"y":1.226,"z":-3.567}}]

STM window snapshot:
STM Window (Short-Term Memory)
update_policy: overwrite_each_step
window_size: 200
last_updated: "2026-05-01T21:05:37Z"

steps:
  - step_id: 1
    action_or_skill: "gui_order_dish_via_computer"
    params: {"dish_name": "Barbecue T-bone Steak with Potatoes and a Hot Corn on the Cob", "computer_target": "Computer"}
    result_summary: "success"
    errors: ""
  - step_id: 2
    action_or_skill: "query_scene_objects"
    params: {"query": "Plate", "only_on_screen": "false", "max_items": "20", "max_distance": "-1"}
    result_summary: "success"
    errors: ""
  - step_id: 3
[... omitted middle STM entries ...]
  - step_id: 191
    action_or_skill: "pick_up"
    params: {}
    result_summary: "success"
    errors: ""
  - step_id: 192
    action_or_skill: "auto_navigation"
    params: {"target": "Side Table", "target_instance_id": "19"}
    result_summary: "success"
    errors: ""
  - step_id: 193
    action_or_skill: "put_down"
    params: {}
    result_summary: "failure"
    errors: "put_down_post_check_failed:interaction_mode_still_active; agent_mode='pour_mode (global_realtime_fallback)'"

```
Planner-facing visual summary:
Visible items (oracle): Side Table-26, Side Table-25, Side Table-27, garlic, Onion, Red Potato, Side Table-24, Small Pot,
↪  Side Table-28, Side Table-19, Side Table-18, Side Table-20, Side Table-17, Side Table-21, Side Table-23, Side Table-12,
↪  Paella Pan, Side Table-11, Side Table-13, Side Table-16, Side Table-10, Side Table-14, Side Table-22, Side Table-5, Side
↪  Table-9, Side Table-15, Side Table-4, Side Table-6, Large Table Center-56, Side Table-3, Side Table-7, Knife Block-6,
↪  Knife Block-5, Large Table Center-52, Side Table-8, Side Table-2, Knife Block, Large Table Center-48, Side Table-1, Large
↪  Table Center-44, Large Table Center-47, Large Plate Source, Plate Source, Square Plate Source, Plate Source, Large Table
↪  Center-43, Large Table Center-40, Deep Plate Stack, Small Plate Source, Casserole Stack, Bowl Source, Large Plate Source,
↪  Plate, Large Table Center-39, Large Table Center-36, Pan, Large Table Center-38, Large Table Center-35, Large Table
↪  Center-32, Large Table Center-34, Large Table Center-31, Large Table Center-33, Large Table Center-30, Large Table
↪  Center-29, Grill Pan, Bucket, Door Right, Small Pot
```

Paired saved model output (step_000194).


```
{"high_level_id":"H1","goal":"Exit the current object manipulation mode and place the held Bake Tray onto a different empty
↪  side table spot.","explanation":"The last attempt failed because the tray was still in an active interaction/control mode
↪  and the prior placement target was invalid.","thoughts":"Clear the active manipulation modes, move to a different empty
↪  side table target, and retry placement.","action_list":[{"step_id":"H1.A1","type":"action","name":"action__exit_pouring
↪  _mode","args":{},"expectation":"The held object is no longer in pouring
↪  mode."},{"step_id":"H1.A2","type":"action","name":"action__switch_to_normal_control_mode","args":{},"expectation":"The
↪  object exits advanced control and returns to normal hold
↪  state."},{"step_id":"H1.A3","type":"skill","name":"skill__auto_navigation","args":{"target":"Side
↪  Table","target_instance_id":"21"},"expectation":"The agent reaches Side Table-21 and faces a valid empty placement
↪  area."},{"step_id":"H1.A4","type":"action","name":"action__put_down","args":{},"expectation":"The Bake Tray is placed on
↪  Side Table-21."},{"step_id":"H1.A5","type":"action","name":"action__switch_to_normal_control_mode","args":{},"expectati
↪  on":"The control state remains in normal mode after
↪  placement."},{"step_id":"H1.A6","type":"skill","name":"skill__auto_navigation","args":{"target":"Side
↪  Table","target_instance_id":"21"},"expectation":"The agent stays aligned with the placed tray
↪  location."},{"step_id":"H1.A7","type":"action","name":"action__put_down","args":{},"expectation":"If still held, the tray
↪  is released
↪  successfully."},{"step_id":"H1.A8","type":"action","name":"action__exit_pouring_mode","args":{},"expectation":"Any
↪  residual pouring interaction is cleared."},{"step_id":"H1.A9","type":"action","name":"action__switch_to_normal_control_
↪  mode","args":{},"expectation":"The agent remains ready for standard interactions with free
↪  hands."},{"step_id":"H1.A10","type":"skill","name":"skill__auto_navigation","args":{"target":"Side
↪  Table","target_instance_id":"21"},"expectation":"The final position is stable near the successful placement spot."}]}
```


F.4 Framework-Specific Control Fragments

Open Loop. The strict open-loop baseline is controlled by prompt text that explicitly forbids reliance on future feedback. It asks for one one-shot whole-task plan and then executes that plan sequentially without replanning.

```
SYSTEM:
You must output one complete JSON action plan for the current high-level goal,
using ONLY the allowed actions/skills.
This is a strict open-loop setting: plan from the current observation only, and
do not assume you will receive later execution feedback or replanning chances.
Each step object MUST include `step_id`, `type`, `name`, `args`, and
`expectation`.
```

ReAct. ReAct uses the same JSON action schema, but the control prompt converts the planner into a short-horizon closed-loop policy by forcing an exact three-step suffix and by prioritizing immediate retry after disambiguation failures.

```
SYSTEM:
You must decide exactly 3 next steps based on the latest observation and
feedback.
You must output STRICT JSON only.

Hard constraints:
- Output exactly 3 steps: action_list length MUST be exactly 3 steps.
- Do NOT invent actions. Use exact action or skill names from the schemas above.
- If feedback includes `blocking=instance_disambiguation`, you MUST choose
  exactly one `instance_id` from `candidates` and retry the same skill with the
  specified `instance_arg`.
```

Plan-and-Execute (PE). PE changes the control flow by making each prompt a partial rolling-horizon plan, rather than a whole-episode script. The planner is reminded that each round is a 10-step chunk and that failures will stop the current chunk and trigger replanning.

```
SYSTEM:
You must output a JSON plan for the current high-level goal, using ONLY the
allowed actions/skills.
PE reminder: this is rolling-horizon planning; each round is partial.
Do NOT force task completion in one round.
Do NOT include `gui_submit_dish_via_checkout_stand` unless there is clear
evidence the dish is ready to serve (or force-submit mode is explicitly active).
Plan length constraint: output exactly 10 steps in this planning round.
```

Reflexion. Reflexion keeps the same 10-step PE planner, but adds a second prompt stream for short-horizon reflection. The reflection prompt is triggered by recent failure density and asks the model to summarize local progress, the blocker, and what the next chunk should focus on.

```
REFLECTOR:
You are Reflexion, a short-horizon self-reflective agent for embodied cooking.
Your job is to briefly reflect on the current planning chunk (about the recent
10 steps): what real progress was made toward the local subgoal, why that
subgoal was not fully completed, what part is still unfinished, and what the
next 10 steps should focus on.

Rules:
- Keep the reflection brief: prefer 3-5 short lines or one short paragraph.
- Focus only on the current planning chunk, not on the whole episode.
- If the recent steps show a loop or oscillation, explicitly explain why the
  loop persisted and what rule would have broken it.
- Prefer planner-level lessons over brittle low-level speculation.
```

Code-as-Policy (CaP). CaP changes both the output format and the planning style. Instead of emitting a JSON action list directly, the prompt asks for a short straight-line `policy(ctx)` function whose sequential `yield` statements are later converted into actions. The prompt also explicitly forbids Python control flow so that the model uses code syntax as structured chain-of-thought rather than as an executable program.

```
SYSTEM:
You are CaP (Code-as-Policies). Write Python planning code that describes a
short action sequence.
This code is for planning only. It is not executed as Python at runtime.
Every regeneration starts from a fresh Python module scope.
Do not use `if`, `for`, `while`, `try`, or helper functions inside `policy(ctx)`.
Use comments to express phase structure. Do not use loops, branches, or
explicit `stage` variables.
Every yield argument must be a plain Python literal only.
Return ONLY a Python code block and no extra text.
```

The prompt package also provides a canonical CaP example so that the model learns to express planning as a literal-only multi-step policy rather than as a runtime state machine:

```
def policy(ctx):
    # Acquire the target object.
    yield skill("query_scene_objects", query="Chicken Breast", only_on_screen=False)
    yield skill("auto_navigation", target="Chicken Breast")
    yield action("pick_up")

    # Move to a placement point and release it.
    yield skill("auto_navigation", target="Side Table")
    yield action("put_down")
```

F.5 Specialized Repair and Disambiguation Prompts

Beyond the main planning prompts, EPM uses short repair prompts for local binding failures. These prompt modes are important because they show how instance resolution is removed from free-form reasoning and turned into an auditable constrained subproblem.

Dedicated instance-ID disambiguation prompt. When a step fails because the target name maps to multiple live scene instances, the next planner call is forced into a narrow selection problem instead of a new free-form multi-step plan. The executor then injects the chosen `instance_id` back into the blocked retry.

```
SYSTEM:
You are in instance-ID disambiguation mode.
The previous step is blocked only because multiple live scene instances match
the same target name.

Return STRICT JSON only:
{"selected_instance_id": 123}

Hard rules:
- Choose from `candidates` only.
- Retry the same blocked action/skill first; do not add unrelated steps.
- Prefer visible, nearer, scene-level, and task-plausible candidates.
- Do not output markdown, code, explanations, or a new action list.
```

CaP query-result binding mode. CaP uses an additional short binding prompt after a successful `query_scene_objects` call. This avoids regenerating a full code policy when the only missing operation is to bind one returned candidate.

```
You are in CaP query-result binding mode.
The previous step was `query_scene_objects` and it already returned non-empty
results.
Choose exactly one result that best matches the current target and return JSON
only:
{"name":"candidate name","instance_id":123}

Selection rules:
- Choose from `results` only.
- Prefer the exact target name from the previous query when present.
- Prefer visible / scene-level / more plausible task objects when needed.
- Do not output code, markdown, or explanations.
```

These prompt fragments illustrate the main control differences across methods. Open Loop suppresses future feedback, ReAct constrains the next suffix, PE enforces chunked partial planning, Reflexion adds a separate failure-summary prompt, and CaP rewrites the planning interface itself into a constrained code-shaped policy language. Because these differences are encoded directly in the prompt contract rather than only in the executor, Appendix F provides the auditable textual evidence for how each agent framework is operationalized in CookBench.