COOKBENCH: CAN DIGITAL AGENTS ACTUALLY COOK? A MULTIMODAL BENCHMARK FOR LONG-HORIZON EMBODIED PLANNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent multimodal agents have demonstrated substantial progress in digital environments, yet whether these capabilities transfer to long-horizon, physically grounded tasks remains unclear. To bridge this evaluation gap, we introduce CookBench, a high-fidelity interactive benchmark for long-horizon embodied planning in complex cooking scenarios, in which agents act primarily through structured text observations, with RGB screenshots as an optional input channel. Featuring 133 single-dish tasks with an average length of over 130 steps, CookBench goes beyond existing benchmarks by incorporating a large action space, 176 interactive objects with 50 functional items, and high-fidelity physical dynamics. Through systematic evaluation of representative agent frameworks and state-of-the-art vision-language models via a standardized evaluation harness (the Embodied Planning Machine, EPM), we reveal a substantial reality gap: even GPT-5.4-driven agents reach a dish-quality score of only 8.35 out of 100, while human operators reach about 93 under the same interface, showing that near-perfect scores are attainable through it. 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. Code and benchmark resources are available at `https://anonymous.4open.science/r/cookbench-anon-C13C`.

1 INTRODUCTION

Recent multimodal agents excel in digital problem-solving. To evaluate their transition to physically grounded environments, existing research relies on digital-agent benchmarks (Koh et al., 2024; Xie et al., 2024; Rawles et al., 2024) and embodied planning benchmarks (Shridhar et al., 2020a; Puig et al., 2018; Padmakumar et al., 2022). 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 planning benchmarks decouple long-horizon planning from persistent physical interaction (see Table 1 for a systematic comparison): their tasks are short-horizon and their physical interactions simple and discrete, so a competent agent can succeed by issuing a fixed sequence of symbolic actions whose outcomes are effectively guaranteed—planning never has to monitor, track, or recover from an ongoing physical process. Continuous state evolution is what changes this: once temperature, liquid volume, and heating progress evolve on their own between actions, irreversible failures such as overcooking or overflow can occur at any moment, so the planner must continuously watch the world, detect anomalies, and replan around degraded states. Consequently, a fundamental question remains unanswered: *how do modern agents perform when long-horizon planning is tightly coupled with persistent physical interactions?*

We anchor our study in the cooking domain, which concentrates persistent physical grounding, the manipulation of functional items, and complex continuous state changes into a single everyday setting; cooking serves as the stress test, while persistent physical state management is the capability under evaluation. To bridge this gap, we introduce CookBench (Figure 1), an interactive benchmark

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Comparison of representative embodied AI and agentic reasoning benchmarks (column definitions: Appendix D.1). '-' = not applicable or not reported; '*' = estimated from the oracle step-length analysis (Appendix D.3); other Seq. Len. entries are measured from demonstrations or reported by the original papers. †Excludes 27 interaction anchors and 171 placement points for comparability with prior benchmarks. ‡BEHAVIOR-1K simulates continuous temperature and liquid states but scores task goals via discrete predicates.

	Benchmark	Task & Eval.		— Environment & Interaction —					— Core Capabilities —						
		Seq. Len.	Auto. Eval.	Simulator	3D Env.	Objs	Func. Items	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+Tool
	ReALFRED (Kim et al., 2024)	~14	✓	AI2-THOR	✓	119	✗	13	✗	✗	✗	✓	Discrete	Coarse	Common+Tool
	EB-ALFRED (Yang et al., 2025)	~13	✓	AI2-THOR	✓	84	✗	13	✗	✗	✗	✓	Discrete	Coarse	Common+Tool
	ALFWorld (Shridhar et al., 2020b)	~9	✓	Text-only	✗	84	✗	13	✗	✗	✗	✓	Discrete	Coarse	Common+Tool
	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+Tool
	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
Manip.	BEHAVIOR-1K (Li et al., 2022)	–	✓	OmniGibson	✓	>10k	✗	–	✓	✗	✓	✗	Continuous‡	Fine	Common
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**	✓	**176**	**50†**	**98**	✓	✓	✓	✓	**Continuous**	**Coarse + Fine**	**Common + Domain + Tool**



Figure 1: The CookBench environment combines diverse ingredients and functional items, an action library, continuous physical state changes, irreversible execution risks, and built-in automatic evaluation. All main experiments standardize on the Classic kitchen scene (Section 2).

built on a high-fidelity Unity simulator, in which agents act primarily on structured text observations, with RGB screenshots as an optional input channel. Its design follows from the capabilities this question demands: long task horizons force sustained goal tracking; a large vocabulary of interactive objects, functional items (e.g., ovens and deep fryers), and actions forces agents to select and bind the right item among many alternatives; and high-fidelity physical dynamics let states evolve between actions and fail irreversibly, so planning must keep pace with a world that does not wait for the agent (scale in Table 1). We release the full stack so that other researchers can pursue their own questions on this platform.

Experiments reveal a substantial reality gap: while human operators reach a quality score of $\sim$93 (out of 100) through the same interface as the agents, even the strongest baseline (Code-as-Policy powered by GPT-5.4) peaks at just 8.35. Mechanism-level diagnosis of this gap is possible because CookBench makes the execution process observable: during interaction, each action returns pre-

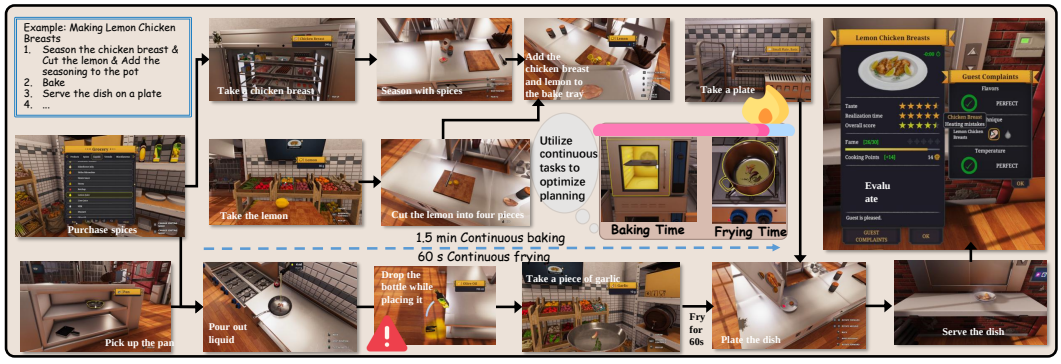

Figure 2: A worked CookBench task instance, *Making Lemon Chicken Breasts*. The workflow interleaves sequential physical operations (seasoning, cutting, pan-frying, plating) with a 1.5-minute continuous baking process, during which the agent can interleave other subtasks to save time, and includes in-run anomaly handling (e.g., dropping the knife while plating). CookBench tasks thus directly pose the challenges of asynchronous temporal optimization—overlapping continuous processes with other work instead of idling—and of online recovery from unexpected physical events.

condition checks, post-state verification, and quantitative progress signals together with live object states; after interaction, each dish is graded along multiple attribute dimensions with textual diagnostics, so partial progress and failure mechanisms remain visible instead of collapsing into a binary flag. Viewed through this lens, the collapse traces to three capability deficits in the physical interaction loop: (1) agents cannot map semantic intents into precise physical actions; (2) when such actions fail, they struggle to translate environmental feedback into corrections that actually change the physical state; and (3) over extended horizons, they lose track of dynamic states they themselves created. The main contributions of this paper are as follows:

1. An interactive benchmark for long-horizon cooking. CookBench provides 133 recipe-grounded tasks combining functional-item use, persistent physical state changes, and final-dish quality evaluation.

2. A shared evaluation substrate for agents. The Embodied Planning Machine (EPM) connects every compared configuration—and the human reference group—to the same action API, structured observations, execution feedback, and human-curated cooking knowledge, under a specified access protocol.

3. A trace-based analysis of execution failures. We combine milestone annotations and execution logs to characterize gaps between planned operations, attempted actions, and final outcomes, identifying recurring failures in precondition management, recovery, and state tracking.

2 CookBench: Tasks, Environment, Harness, and Evaluation

CookBench is a benchmark for long-horizon embodied planning in cooking, organized into two layers. The *benchmark content* provides an interactive cooking **environment** with continuously evolving physical states and a suite of 133 recipe-grounded tasks; the **evaluation harness** (the Embodied Planning Machine, EPM; Figure 3) then connects these tasks to generic digital agents, standardizing how agents perceive, act, and receive feedback. This section details the components in turn: the tasks, the environment, the harness, and the evaluation metrics.

2.1 Task Definition

CookBench formulates tasks as embodied interactions (Figure 2): agents iteratively invoke actions to execute recipe-grounded workflows over a large set of interactive objects and functional items, and receive an objective score from the final physical outcome. As Figure 2 illustrates, these workflows couple sequential operations with continuous timed processes (e.g., baking), so efficient execution

requires asynchronous temporal optimization—interleaving other subtasks while a process runs—as well as online recovery from unexpected physical events.

CookBench comprises 133 single-dish tasks, each averaging over 130 estimated oracle steps (oracle step-length analysis in Appendix D.3), further grouped into six cooking-style categories based on dominant operations, component organization, and dish styles (single-dish cooking-style taxonomy in Appendix D.4).

2.2 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). To isolate embodied planning capabilities from fundamental perception noise, we develop a dedicated state-extraction interface via custom engine-level scripting, called the **World Adapter**. It provides agents with structured semantic observations—names, locations, and physical attributes of all entities, alongside the agent's real-time state—covering **176** interactive objects (**97** products, **37** spices, and **42** liquids) and **50** functional items, together with **27** interaction anchors (appliance switches, place points, and hooks) and **171** placement points (the full released vocabulary is listed in Table 11). The World Adapter, the evaluation harness, and all evaluation artifacts are open-sourced; the complete reproducibility stack (release and setup details) is in Appendix B.

Transforming this human-centric environment into an evaluation testbed for autonomous agents requires bridging the gap between raw keyboard/mouse inputs and semantic reasoning. We manually map the raw motor controls into a callable semantic action space of **25** action categories and **85** atomic actions, and further encapsulate **13** composite actions (e.g., `auto_cut`, which aims a knife at a target item and cuts it into n equal pieces)—predefined combinations of atomic actions that execute recurring physical routines—collectively the **Cerebellum**. A diagnostic system translates raw physical collisions into semantic, actionable textual feedback, giving the environment closed-loop execution (action and state design in Appendix D.5).

2.3 THE COOKBENCH HARNESS

We introduce the *Embodied Planning Machine* (EPM, Figure 3), a standardized evaluation harness that equips generic digital agents with the interaction, feedback, and memory interfaces required for long-horizon, physically grounded execution. EPM provides semantic actions, structured observations, execution feedback, and persistent runtime memory for all compared agents; Appendix F.8 evaluates the contribution of these components through paired single-component ablations. All agent frameworks are therefore uniformly adapted through the harness.

As one component of the harness, we manually curate a structured knowledge base with six parts: *action knowledge* (interface protocols for callable actions), *cookware knowledge* (functions and affordances of containers and functional items), *product knowledge* (physical and semantic attributes of ingredients), *common sense* (interaction constraints and physical rules), *experience knowledge* (reusable heuristics for scoring and efficiency), and *domain knowledge* (recipes for the 133 dishes), persisted as plain-text/JSON artifacts injected into the planner's prompt at runtime (artifact samples in Appendix C). Each task supplies its target recipe as part of the agent's input; CookBench evaluates how agents execute this knowledge under evolving physical states and interaction constraints, and the knowledge-removal ablations in Appendix F.8 quantify what these priors contribute.

2.4 EVALUATION METRICS

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

Dish quality is scored automatically against the recipe-defined ingredient composition and processing requirements. CookBench uses *Cooking Simulator*'s native recipe-scoring function and retains its original weights and thresholds (Appendix F.6). At the component level, the simulator aggregates four dimensions—taste and proportion α_k, thermal processing β_k, technique γ_k, and final serving temperature δ_k, with weights λ_α, λ_β, λ_γ, λ_δ—into the quality score ψ_k =

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

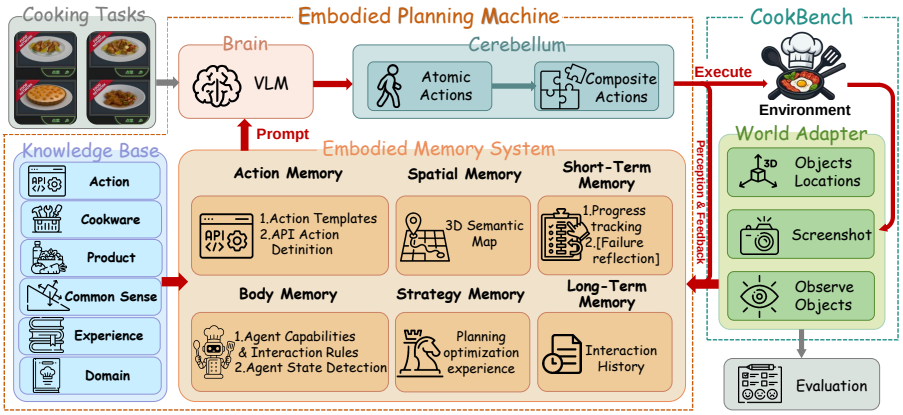

Figure 3: Overview of the Embodied Planning Machine. As a cognitive evaluation substrate, EPM coordinates the information flow across three dimensions: it aligns agent reasoning with a structured **knowledge base**, tracks long-horizon changes via an **embodied memory system**, and processes continuous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

$(\lambda_\alpha\alpha_k + \lambda_\beta\beta_k + \lambda_\gamma\gamma_k + \lambda_\delta\delta_k)/(\lambda_\alpha + \lambda_\beta + \lambda_\gamma + \lambda_\delta)$. The complete hierarchical scoring pipeline—part-level weighting, component-level aggregation, and dish-level weighting and penalties—is described in Appendix F.6, with worked case studies. The score is calibrated at the top of its range—human operators average 96.8 under native controls and 93.34 through the harness—and it tracks verified progress rather than exploitation of the scoring function: over the 8×133 annotated rollouts of the main experiment (eight agent–model settings, one run per dish; Table 2), the final score correlates with LLM-verified gold-milestone completion (Spearman $\rho = 0.42$, $p < 10^{-41}$, significant within every agent–model setting; score–coverage validation in Appendix F.6.1).

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 are detailed in Appendix Table 17.

3 Experiments and Analysis: Agent Performance in Physical Interaction

3.1 Experimental Settings

In all main experiments we adopt the oracle-goal protocol: agents receive structured cooking goals rather than raw natural-language instructions, separating instruction-interpretation noise from embodied planning and execution errors; natural-language intent recognition is evaluated separately (Appendix F.3.1).

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., 2023b)), **Reflexion** (reflection-augmented correction (Shinn et al., 2023)), and **Code-as-Policy (CaP)** (following the programmatic formulation of Liang et al. (2022): the model writes a Python-style plan over the action API, from which EPM statically extracts the action queue and executes it stepwise—the emitted code itself never runs as runtime control flow; implementation details in Appendix F.1). For this framework comparison, we fix the backbone model to GPT-5.4 (OpenAI, 2026).

Multi-modal Models: We deploy five models using *CaP*, the strongest-performing framework from our initial evaluation: 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).

Table 2: Main benchmark results across six CookBench dish categories. The human reference row reports volunteers operating through the same harness semantic interface as the agents (protocol in Appendix F.4); human time is wall-clock including deliberation and is not comparable to agent execution time. Among the nine non-human agent/model settings, bold values denote the best and underlined values the second-best performance. Execution time is wall-clock episode time up to termination—voluntary submission, forced submission, or the 250-step hard cap, which consists of the 200-step interaction limit plus a 50-step force-submission buffer (Appendix F.1.1); for the four CaP multimodal rows it is averaged over each setting's completed runs. Each agent–model cell is a single rollout per dish over the full 133-dish set; run-level variance is estimated separately from three repetitions on a stratified 20-dish subset (run-level repetition analysis in Appendix F.9).

Setting	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	
	Score % ↑	Time min ↓	Score % ↑	Time min ↓	Score % ↑	Time min ↓	Score % ↑	Time min ↓	Score % ↑	Time min ↓	Score % ↑	Time min ↓	Score % ↑	Time min ↓
Human Reference (Same Harness Interface as Agents)														
Human (Harness)	94.85	24.84	93.12	46.06	92.40	34.40	94.60	40.33	91.75	23.34	93.30	19.41	93.34	32.23
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: 20 volunteers with no prior *Cooking Simulator* experience covered the 133 dishes (up to three attempts each, mean recorded) under native keyboard-and-mouse controls; a second, disjoint group of 20 operated through the same harness semantic API as the agents (protocol in Appendix F.4).

To keep episodes comparable and prevent unbounded retries, all settings share one step budget (oracle step-length analysis in Appendix D.3) and one force-submission policy (Appendix F.1.1); agent-framework and configuration details are in Appendices F.1 and F.2.

3.2 Main Results: The Reality Drop in Physical Interaction

Table 2 reports the main results, from which we draw four observations.

(1) There is a large performance separation between agents and humans. The best evaluated agent configuration scores 8.35/100, while human operators using the same harness semantic API average ∼93 (Table 2, first row; protocol in Appendix F.4, where human scores average up to three attempts per dish versus one run per dish for agents). Section 3.3 examines the action and state-transition patterns associated with agent failures.

(2) Under a fixed backbone, control flow matters. Fixing the model to GPT-5.4, CaP obtains the highest observed mean (8.35%), followed by Reflexion (6.05%), PE (5.28%), and ReAct (4.12%). Paired Wilcoxon signed-rank tests with Benjamini–Hochberg correction over the ranking family (ranking tests and repetition analysis in Appendix F.9) confirm that CaP is significantly better than ReAct ($q = 0.006$); its advantage over PE is marginal after correction (raw $p = 0.022$, $q = 0.060$) and its advantage over Reflexion is not significant ($q = 0.116$). The remaining mid-tier gaps (1–2 points) are within the run-level variance from three independent repetitions on a 20-dish stratified subset (Appendix F.9), so we discuss these frameworks mechanistically rather than as a strict ranking. PE's 10-step chunking descriptively outperforms ReAct's 3-step short horizon (5.28% vs. 4.12%) with the shortest execution time: high-frequency reactive planning tends to lose track of global goals. Reflexion adds failure-triggered textual reflection to the PE base, but the difference (6.05% vs. 5.28%) lies within run-level noise: logs show agents proposing parameter tweaks rather than state-altering physical repairs. CaP's code format forces an explicit, sequential enumeration of the action plan, and persistent state is maintained externally by EPM rather than inside the model's context—consistent with the no-history ablation (Appendix F.8) and qualitative logs. The zero score

of *Open Loop* further shows that one-shot semantic planning without closed-loop physical feedback is not viable in this setting.

(3) Under a fixed framework, backbones differ, and task structure mediates difficulty. Under the *CaP* framework, *GPT-5.4* achieves the highest average score (8.35%), while *Gemini-3.1-Pro* is the strongest alternative (6.77%). *GPT-5.4*'s lead over *Gemini-3.1-Pro* does not reach significance ($q = 0.164$), while *Gemini-3.1-Pro* significantly exceeds *Kimi-k2.6* after Benjamini–Hochberg correction ($q = 0.033$; Appendix F.9). The gaps among the other non-GPT models (*Kimi-k2.6* 4.13%, *Qwen3.6-Plus* 4.20%, *Qwen3.6-35B* 3.84%) are below 0.4 points and lie within run-level noise. Tasks with relatively linear, pipeline-style workflows (e.g., Pasta-Assembly, Soup-Stew) allow most models to make non-zero progress; by contrast, tasks demanding intensive multi-component synchronization, frequent mode switching, and delayed state convergence (e.g., Grill-Combo, Salad-Cold) yield the lowest scores, with several models collapsing to near zero. Category-level orderings are descriptive; the dish-level bootstrap 95% confidence intervals of Table 2 (Appendix F.10) confirm that the overall agent–human gap is an order of magnitude larger than any interval width. This indicates that CookBench's difficulty lies not merely in long step horizons but in the structural complexity of embodied physical interactions. Dish-level features confirm this: 14 dishes (10.5%) score zero under every agent–model configuration while human testers average 96.9 on exactly these dishes, and difficulty is predicted by procedural length (recipe steps, $\rho = -0.23$, $p < 0.01$) and by multi-container transfer (the universally unsolved dishes average more containers than dishes solved by at least one setting: 1.79 vs. 1.24 containers; point-biserial $r = -0.19$, $p < 0.05$; recipe-level difficulty analysis in Appendix F.9).

(4) Interface components contribute measurably, and the headline gaps are robust to run-level variance. Our ablation configuration removes one EPM component family at a time, pairing each of the eight single-component conditions against the full configuration on the same 20-dish subset under three agent–model settings (condition definitions in Appendix F.8); with rollout variability from three repetitions on that subset (Appendix F.9). Removing the semantic action layer collapses scores to exactly zero in all settings, removing execution feedback costs 89–100% of the score for GPT-based pipelines, and removing body rules, strategy knowledge, or cross-step history consistently degrades performance (-20.8% to -78.0%). The contribution of RGB input is model-dependent: removing it lowers the mean score by 40–42% in the two GPT-5.4 settings (removing pre-injected perception support lowers them by 32–34%) but raises Gemini's score by 52%. Run-level variance is small relative to the headline gaps: the reality gap to human performance (human harness $\sim$93% vs. 8.35% for the best agent setting, both over the full 133 dishes) and the zero score of *Open Loop* are fully robust, while the three-run subset standard deviations are at most 1.02 points, so only mid-tier framework differences (1–2 points) fall within run-level noise.

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

To diagnose where agents break the physical interaction loop, we analyze the natural cascade of failures: from translating semantic intents into physical actions (3.3.1), to processing environmental friction during failures (3.3.2), to maintaining dynamic states over extended horizons (3.3.3).

3.3.1 The Intent–Execution Ladder: Where Does the Cooking Chain Break?

To localize where the cooking chain breaks, we annotate every run at the granularity of *gold milestones*, classifying each by the highest attained level of a five-level *intent–execution ladder*: **L0** no recipe-grounded intent; **L1** component-level intent only; **L2** an explicit culinary subgoal (e.g., *cut Onion*); **L3** a concrete action call attempted (atomic or composite); **L4** the intended outcome confirmed by the final scoring feedback. A two-stage LLM pipeline (DeepSeek-v4-Pro) labels every gold milestone of every run log, yielding 23,743 milestone-level judgments over the 8×133 annotated rollouts of the main experiment; the score-validity analysis relates these labels to the final scoring feedback (19,728 milestone judgments; annotation pipeline in Appendix G.2, score–coverage validation in Appendix F.6.1). We also report error-format statistics and a micro-mechanism breakdown of execution failures in Appendix G.3.

Findings. Figure 4 yields three observations. **(1) Agents rarely fail because they do not know what to cook.** Component-level intent coverage (L1) exceeds 73% in all eight settings, and concrete milestone-level intent (L2) reaches 67.4% for *CaP@GPT-5.4* and 72.6% for *CaP@Gemini-3.1-Pro*.

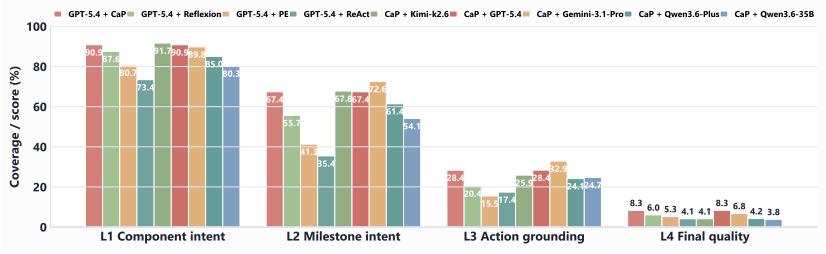

Figure 4: Milestone-level intent–execution analysis across the eight agent–model settings: per-run coverage of gold milestones attaining L1 (component intent), L2 (culinary-milestone intent), and L3 (action-level grounding); L4 is the final quality score. L1–L3 percentages are LLM-annotation estimates (L2 human–model agreement $\kappa = 0.41$; Appendix G.2); L4 is computed from the benchmark score and serves only as a consistency check, with the independent diagnostic information carried by L1–L3. Intent coverage stays high at L1/L2 but collapses toward L3/L4, localizing the dominant loss in semantic-to-physical conversion.

(2) The chain collapses at grounding. L3 action-grounding coverage drops to 15.5%–32.9%, and final score-recognized completion (L4) is lower still (3.8%–8.4%). *CaP@Gemini-3.1-Pro* achieves the highest L2/L3 coverage yet loses 65.8 points between L2 and L4: the deficit lies almost entirely in conversion rather than comprehension. **(3) A substantial share of required milestones is never even attempted.** 6.3–15.1 required milestones per dish remain at L0. The dominant bottleneck is thus **the semantic-to-physical conversion of recipe-grounded intentions into executable, score-recognized behavior, not API-format validity or wholesale planning failure**. The downstream signals in Sections 3.3.2 and 3.3.3 are computed deterministically from execution logs.

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

We analyze two auditable failure signals following execution feedback and replanning. Figure 5 yields two observations. **(1) Both failure signals remain consistently high across all settings**: blocked retry ranges from 31% to 68% and post-replan error from 58% to 86%. **(2) Qualitative logs expose the root cause**: agents misread physical constraints (e.g., occupied hands) as digital parameter errors and reissue near-identical intents instead of state-altering interventions—a clear deficit in **physical feedback comprehension**.

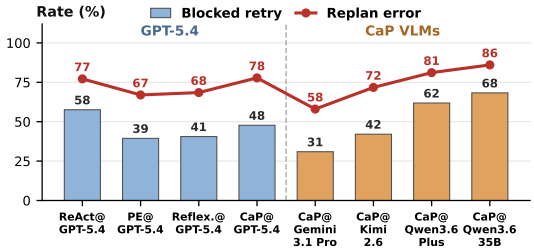

Figure 5: Failure signals for closed-loop repair. Bars: failed steps repeating the same failed action within five steps. Line: replans failing to prevent another error within five steps.

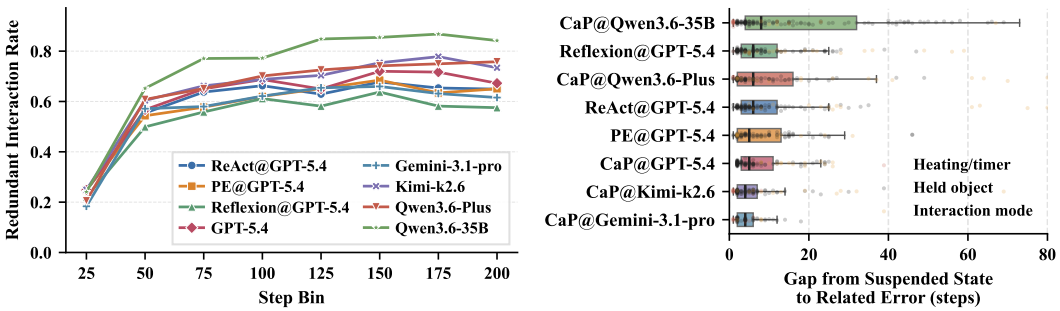

Figure 6: Symptoms of embodied working-memory decay. **(a)** Redundant interaction rate rises over the horizon. **(b)** Steps from opening a continuous physical state to the later state-conflict error it causes; a state spans from initiating a process to closing it (e.g., switching heating on and, later, off).

3.3.3 THE HORIZON EFFECT: DOES EMBODIED WORKING MEMORY DECAY OVER TIME?

Cooking requires persistent state maintenance across physical transformations; we probe two symptoms of its decay: suspended unresolved states that later trigger conflicts, and rising repetition of attempted interactions late in the horizon (Figure 6).

Figure 6 yields three observations. **(1) Redundant interactions rise steadily over the horizon** (Figure 6(a)): agents increasingly spend steps on repeated successful operations, repeated failed attempts, and local repair loops. **(2) Opened continuous states stay unresolved for dozens of steps** (Figure 6(b)) until a state-conflict error surfaces the unmaintained variable. **(3) This is not a context-window problem**: the textual history records earlier actions, yet agents still fail to maintain the current embodied state as trajectories lengthen—**embodied state amnesia**. Interface and recovery policies are shared across frameworks, so agents need a structurally updated working memory of the evolving 3D state, not just a textual log.

4 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; Yao et al., 2025; Wang et al., 2026). While these benchmarks establish robust multi-step reasoning capabilities, they remain decoupled from the physical world, bypassing 3D spatial interaction, continuous state evolution, 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; Puig et al., 2018; Padmakumar et al., 2022; Deitke et al., 2022). However, tasks in these environments remain relatively short-horizon (avg. $\sim$10 steps) with coarse-grained action primitives and discrete state changes. BEHAVIOR-1K (Li et al., 2022) is larger (1,000 household activities with extended object-state tracking) but targets low-level mobile manipulation rather than long-horizon task planning. Kitchen settings are a canonical domain for coordination and state interpretation (Bollini et al., 2013; Mavrogiannis et al., 2024), but existing cooking benchmarks study abstract multi-agent cooperation (Carroll et al., 2019) or asynchronous scheduling (Gonzalez-Pumariega et al., 2025; Zhang et al., 2025b) in simplified or specialized settings. Most directly related, VRKitchen (Gao et al., 2019) builds a UE4 cooking simulator with tool use and object state changes, but targets short-horizon single tasks without continuous quantity control (temperature, volume) or automatic full-workflow scoring; CuisineWorld (Gong et al., 2023) coordinates multi-agent cooking through a text-only interface.

Games as Long-Horizon Agent Environments. Complementary work uses games as long-horizon testbeds. Minecraft agents such as MineDojo (Fan et al., 2022), Voyager (Wang et al., 2023a), GITM (Zhu et al., 2023), and Optimus-3 (Li et al., 2025) pursue goals via LLM planning and memory in a symbolic world model (item graphs, discrete crafting rules). Other game agents instead imitate human keyboard-and-mouse play (Tan et al., 2025a; SIMA Team, 2025; Magne et al., 2026), while a related line targets the general computer interface (Tan et al., 2025b). Such play stays at screen level, demanding rich perception and long-horizon strategy, yet only marginally involves complex 3D physical interaction. CookBench keeps the paradigm but targets persistent physical-state tracking, irreversible execution risks, and fine-grained continuous control, asking whether digital capabilities transfer to physically grounded execution.

5 CONCLUSION

CookBench exposes the gap between digital semantic reasoning and physically grounded execution. The strongest current agents reach a dish-quality score of 8.35 where human operators reach 93, and the failure traces converge on three missing capabilities: **physical grounding**, **closed-loop physical feedback understanding**, and **embodied working memory**. By measuring this loop directly under one harness shared by humans and agents, CookBench turns an often-anecdotal gap into a quantifiable target for tracking progress on the digital-to-physical transition.

Ethics statement

The human baseline study involved 40 volunteers in two disjoint groups of 20: one group operated the game under native keyboard-and-mouse controls, and the other operated through the same harness semantic API as the agents (Appendix F.4). All participants took part with informed consent; no personal identifying information was collected, and participants were compensated for their time. The benchmark is built on a commercially licensed game used within the terms of its license, and all released artifacts contain no proprietary game assets. We do not foresee harmful applications of this work beyond general concerns applicable to embodied-agent research.

Reproducibility statement

We provide a complete reproducibility stack in Appendix B, and we distinguish three levels of reproducibility explicitly. **(i) Open code**: the World Adapter mod (compiled binary), the full EPM codebase with agent configurations and prompts, all task definitions, and evaluation scripts are released under the MIT license (anonymized repository linked in the abstract). **(ii) Runnable reproduction**: the repository ships an exact game-version lock with an automated reconstruction script and a one-command episode runner, so a clean machine with a standard Steam installation can rerun a task end-to-end. **(iii) Verifiable numbers**: the scoring formulas, metric constructions, and statistical-test procedures needed to verify all quantitative claims are documented in Appendices F.6 and G.1. The only closed-source dependency is the commercial *Cooking Simulator* engine, accessed through a standard Steam installation.

References

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

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

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

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

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In *NeurIPS Datasets and Benchmarks*, 2022.

Xiaofeng Gao, Ran Gong, Tianmin Shu, Xu Xie, Shu Wang, and Song-Chun Zhu. Vrkitchen: An interactive 3d virtual environment for task-oriented learning. In *ICCV*, 2019.

Ran Gong, Qiuyuan Huang, Xiaojian Ma, Hoi Vo, Zane Durante, Yusuke Noda, Zilong Zheng, Song-Chun Zhu, Demetri Terzopoulos, Li Fei-Fei, and Jianfeng Gao. Mindagent: Emergent gaming interaction. *arXiv preprint arXiv:2309.09971*, 2023.

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

Google. 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/`, 2026. Accessed: 2026-05-23.

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

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

Emily Jin, Jiaheng Hu, Zhuoyi Huang, Ruohan Zhang, Jiajun Wu, Li Fei-Fei, and Roberto Martín-Martín. Mini-behavior: A procedurally generated benchmark for long-horizon decision-making in embodied ai. *arXiv preprint arXiv:2310.01824*, 2023.

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

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. 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)*, pp. 881–905, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.50. URL https://aclanthology.org/2024.acl-long.50/.

Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Claire Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, Mona Anvari, Minjune Hwang, Manasi Sharma, Arman Aydin, Dhruva Bansal, Samuel Hunter, Kyu-Young Kim, Alan Lou, Caleb R. Matthews, Ivan Villa-Renteria, Jerry Hu Tang, Claire Tang, Fei Xia, Silvio Savarese, Hyowon Gweon, Karen Liu, Jiajun Wu, and Li Fei-Fei. Behavior-1k: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation. In *CoRL*, 2022.

Zaijing Li, Yuquan Xie, Rui Shao, Gongwei Chen, Weili Guan, Dongmei Jiang, Yaowei Wang, and Liqiang Nie. Optimus-3: Towards generalist multimodal minecraft agents with scalable task experts. *arXiv preprint arXiv:2506.10357*, 2025.

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

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

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, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Loïc Magne, Anas Awadalla, Guanzhi Wang, Yinzhen Xu, Joshua Belofsky, Fengyuan Hu, Joohwan Kim, Ludwig Schmidt, Georgia Gkioxari, Jan Kautz, Yisong Yue, Yejin Choi, Yuke Zhu, and Linxi Fan. Nitrogen: An open foundation model for generalist gaming agents. In *CVPR*, 2026.

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

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023. URL https://arxiv.org/abs/2311.12983.

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

OpenAI. Introducing gpt-5.4. https://openai.com/index/introducing-gpt-5-4/, 2026. 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. Teach: Task-driven embodied agents that chat. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 2017–2025, 2022.

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

Qwen Team. Visual understanding models. https://docs.qwencloud.com/developer-guides/getting-started/vision-models, 2026. 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, et al. Androidworld: A dynamic benchmarking environment for autonomous agents, 2024. URL https://arxiv.org/abs/2405.14573.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

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

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

SIMA Team. Sima 2: A generalist embodied agent for virtual worlds. *arXiv preprint arXiv:2512.04797*, 2025.

Weihao Tan, Xiangyang Li, Yunhao Fang, Heyuan Yao, Shi Yan, Hao Luo, Tenglong Ao, Huihui Li, Hongbin Ren, Bairen Yi, Yujia Qin, Bo An, Libin Liu, and Guang Shi. Lumine: An open recipe for building generalist agents in 3d open worlds. *arXiv preprint arXiv:2511.08892*, 2025a.

Weihao Tan, Wentao Zhang, Xinrun Xu, Haochong Xia, Ziluo Ding, Boyu Li, Bohan Zhou, Junpeng Yue, Jiechuan Jiang, Yewen Li, et al. Cradle: Empowering foundation agents towards general computer control. In *ICML*, 2025b.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2023a.

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

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

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

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

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. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

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

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

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

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

Lingfeng Zhang, Yuening Wang, Hongjian Gu, Atia Hamidizadeh, Zhanguang Zhang, Yuecheng Liu, Yutong Wang, David Gamaliel Arcos Bravo, Junyi Dong, Shunbo Zhou, et al. 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)*, pp. 21566–21573. IEEE, 2025a.

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

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, Yu Qiao, Zhaoxiang Zhang, and Jifeng Dai. Ghost in the minecraft: Generally capable agents for open-world environments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*, 2023.

APPENDIX ROADMAP

- **Appendix A:** Detailed architecture of the Embodied Planning Machine (EPM), including environment encapsulation and composite action abstraction.
- **Appendix B:** Reproducibility stack, open-source release, and version compatibility policy.
- **Appendix C:** Representative data samples of the knowledge-base and memory artifacts.
- **Appendix D:** Additional benchmark details, including task construction, oracle step-length analysis, and the cooking-style taxonomy.
- **Appendix E:** Detailed challenge analysis of CookBench (action space, affordance constraints, continuous states, physical anomalies, knowledge requirements).
- **Appendix F:** Additional experimental details: agent framework implementations, configuration, evaluation protocol, human baseline protocol, planning cost, scoring details, case studies, EPM ablation, and run-level repetition analysis.
- **Appendix G:** Additional quantitative results: extended analysis questions, the intent–execution annotation pipeline, and the micro-mechanism breakdown of execution failures.
- **Appendix H:** Domain scope and generalization of the diagnosed failure modes.
- **Appendix I:** Limitations and future work.
- **Appendix J:** Prompt templates and complete prompt examples.

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 Composite Action 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 composite actions (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 containers.
- **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 Composite Action 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. Figures 7–11 illustrate this interface from complementary views. Figure 7 shows the instance-level semantic overlay of the ingredient storage area: every interactable entity carries a live label with its object name and distance, color-coded by category, which together form the color-coded 3D spatial map that exposes the kitchen layout to the agent. Figures 8–10 zoom into the appliance stations, where the interface exposes fine-grained interaction anchors rather than monolithic appliances: the sink and oven area (Figure 8) exposes faucet switches, sink placement points, oven doors, oven place points, and baking-tray rack anchors; the stovetop station (Figure 9) exposes per-burner stove switches, stove place points, griddle surfaces, and functional-item hook points; and the fryer–grill station (Figure 10) exposes fryer slots, basket anchors, oil filling/discharge switches, and individual grill surfaces. Finally, Figure 11 illustrates containment relations and articulated states: the refrigerator exposes its open/closed door and drawer states together with the per-shelf and per-drawer contents stored inside, so the agent can query not only where an ingredient is but also which container or compartment currently holds it and whether that compartment is accessible. Figures 12–17 further illustrate the live state tracking of this layer: scene-wide instance labeling that remains consistent as the agent moves across stations (Figure 12); recognition of loose items poured out of their packaging, with their real-time 3D positions (Figure 13); the item sources, placement anchors, and the agent's currently held item along the preparation counters (Figure 14); newly spawned items recognized as soon as they appear at item sources (Figure 15); new pieces created by cutting, tracked as separate instances (Figure 16); and per-component open/closed states of doors and drawers (Figure 17). Tables 3 and 4 summarize the low-level action space and the reusable high-level composite actions 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 composite actions. 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 composite-action 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. All composite actions that require physical engagement (e.g., cutting, pouring) encapsulate geometric alignment as part of their implementation: the composite action first navigates the agent to the target and orients its view, and only then requests the engine's mode-entry check. The agent itself never outputs low-level poses or coordinates. Consequently, failures of composite actions such as `enter_pouring_mode` and `enter_cutting_mode` reflect semantic binding or precondition errors on the agent side, or the simulator's own interaction validation, rather than perception or alignment noise from the World Adapter.



Figure 7: Instance-level semantic overlay of the ingredient storage area. Every interactable entity is annotated with its object name and live distance, color-coded by category, forming the color-coded 3D spatial map through which EPM exposes the kitchen layout.

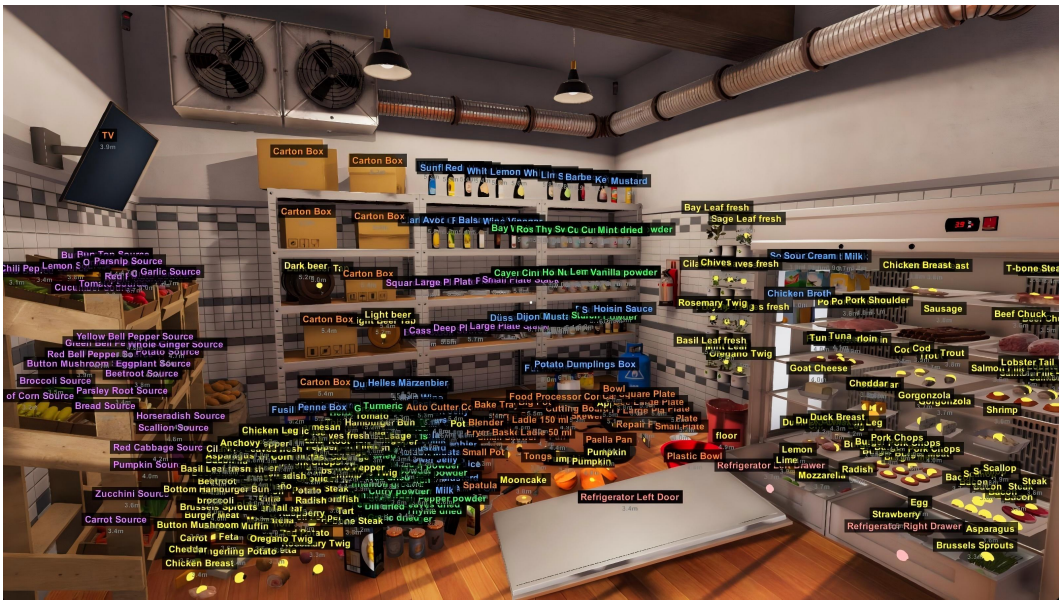

Figure 8: Interaction anchors of the sink and oven area. The interface exposes faucet switches, sink placement points, oven doors, per-compartment oven place points, and baking-tray rack anchors instead of a monolithic appliance object.

B Reproducibility and Open-Source Release

This section specifies exactly how CookBench can be reproduced end-to-end, which components are open-sourced, and how the single closed-source dependency (the commercial game) is isolated from the benchmark logic. Figure 18 summarizes the three-layer reproducibility stack.



Figure 9: Interaction anchors of the stovetop station. Per-burner stove switches, stove place points, griddle surfaces, and functional-item hook points are individually addressable, allowing fine-grained control and state verification during cooking.



Figure 10: Interaction anchors of the fryer–grill station. Fryer slots, basket anchors, oil filling/discharge switches, and individual grill surfaces are exposed separately, so composite actions such as deep-frying must reason about the correct functional-item chain rather than a single appliance toggle.

B.1 A Three-Layer Reproducibility Stack

Layer 1: Simulation engine base (Cooking Simulator). CookBench uses *Cooking Simulator* (Unity-based) purely as a physics and rendering engine: it provides thermodynamics, fluid and collision simulation, the Classic sandbox kitchen scene, and the built-in dish-scoring system. The game itself is the only closed-source component. Two properties make this dependency manageable.



Figure 11: Containment relations and articulated states exposed by the interface. The refrigerator exposes its open/closed door and drawer states together with per-shelf and per-drawer contents, so agents can query which compartment holds an ingredient and whether that compartment is currently accessible.



Figure 12: Scene-wide instance labeling across the storage and cooking stations. The semantic overlay remains consistent as the agent moves, covering appliances, shelves, spawn points, and functional items with live names and distances.

First, *Cooking Simulator* is an offline single-player title released on Steam in 2019 that has remained stable for years, and its scoring system has been continuously validated by a large player community. Second, because it is an offline single-player title, Steam updates it only when the user launches it online, so an installed copy can be kept at a fixed version indefinitely. CookBench therefore records the exact Steam build used for all experiments in this paper, and all reported scores are conditioned on that build. In practice, a standard Steam installation is expected to work out of the box: the



Figure 13: Recognition of loose items poured out of their packaging. Dry goods emptied from boxes (e.g., fusilli, penne, potato dumplings) and individual spices are detected as separate instances with real-time 3D positions.



Figure 14: Item sources, placement anchors, and held-item tracking along the preparation counters. Plate, bowl, and casserole sources, per-table placement points, and the item currently held by the agent are all exposed with live positions.

game's core physics and scoring logic has remained stable across updates since its 2019 release, and the World Adapter accesses only these stable engine APIs. The game is obtained through the standard Steam store; no modification of game binaries or assets is required.

Layer 2: World Adapter (engine-level mod). The bridge between the closed engine and the benchmark is the **World Adapter**, an engine-level extension layer that we built on top of the simulator's physics engine: at runtime it hooks into the engine's object, event, and scoring systems



Figure 15: Recognition of newly spawned items. Items generated at item sources appear in the structured observation immediately, together with their category, attributes (e.g., fresh or dried herbs), and positions.



Figure 16: Recognition of items created by cutting. A single whole tomato becomes two separately tracked instances after cutting, each with its own live position, so the agent can address the pieces individually.

through reflection and augments them with benchmark-facing capabilities. Technically, the extension is delivered as a mod script injected via the standard *MelonLoader* mod loader—the same legal, officially tolerated modding mechanism used by the game's modding community; it neither modifies nor redistributes any proprietary game content. Once loaded, the extension continuously exposes everything the benchmark needs:



Figure 17: Per-component open/closed state detection. Doors and drawers (e.g., of the refrigerator) expose their individual articulated states together with their contents, so the agent can tell which compartments are currently accessible.

- **Structured world state**: live instance IDs, object names, 3D positions and bounding boxes, physical attributes (temperature, quantity, form, doneness), containment relations, and door/drawer open states for all 97 products, 42 liquids, 37 spices, and 50 functional items;
- **Diagnostic feedback signals**: collision rejections, overflow events, state conflicts, and per-dimension score traces (taste/proportion, thermal processing, technique, serving temperature) with textual error descriptions;
- **Semantic action information**: the interaction points and supported operations of every appliance, container, and functional item, which ground the semantic action API described in Appendix A.1.

All of this information is written to a real-time file bridge (e.g., `realtime_products.json`, radar scan, camera info, and interaction status streams), which fully decouples the benchmark from the engine internals: the Python side only reads structured files and never touches game code. The compiled module will be released under an open-source license.

Layer 3: EPM evaluation harness (open-source Python project). The full closed-loop interaction is driven by EPM, a standalone Python project (Figure 3). At each step, EPM's World Adapter client reads the structured state files and captures an RGB screenshot of the game window; the Brain (an arbitrary LLM/VLM planner—ReAct, Reflexion, Planner-Executor, etc.) produces the next action calls; the Cerebellum (`cookbench_api`) executes them through keyboard-and-mouse APIs (85 atomic actions and 13 composite actions); and the environment writes updated state and feedback files, closing the loop. Every run automatically archives its full memory, prompts, per-step screenshots, and execution logs, which constitute the reproducible run record.

B.2 Setup and Released Artifacts

Reproducing CookBench requires only the following steps, all of which are documented in a step-by-step configuration guide that we release together with the code (anonymized repository linked in the abstract):

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

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

1. Purchase and install the pinned *Cooking Simulator* base game[2] and the *Cooking with Food Network* DLC[3] from Steam;
2. Install *MelonLoader*[4] (v0.5.7) into the game directory and deploy our World Adapter module into the game's `Mods` folder—a runtime engine-level extension that patches the simulator's physics and object layer in memory (legal mod injection; no game files are modified). Two small object-name mapping files are placed under the game's `UserData` directory;
3. Clone the EPM repository, create a Python 3.10 environment, and install the package (`pip install -e ./epm`); then set one path in the configuration file (`game_userdata_root`) pointing to the game's `UserData` directory. Private credentials such as LLM API keys are supplied through local, untracked config files or environment variables, for which annotated templates are provided;
4. Launch the game into the Classic sandbox kitchen (New Game → Sandbox Mode → Classic), verify that the adapter's in-game overlay (toggled with `F12`/`F10`) reports live object data, and run the provided episode/evaluation scripts. A dashboard script supports batch dish ranges, automatic chaining, and resuming interrupted runs from checkpoints.

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

[3] https://store.steampowered.com/app/1168680/Cooking_Simulator__Cooking_with_Food_Network/

[4] https://melonloader.co/download.html

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

Composite Action	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 functional item.
`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 usage metadata from the cookware manual.

Version compatibility policy. Since the World Adapter reads engine internals, it is the only component sensitive to the game version. Steam updates to the game have historically touched content and peripheral features rather than the core simulation and scoring logic the adapter depends on, so a current Steam download is in most cases directly compatible with our released adapter. We nevertheless record the exact build used for all reported experiments in the released configuration guide, and we ship versioned adapter builds keyed to the game build they were validated against (our internal archive already maintains a dated history of adapter builds as the game evolved). In the event that a future game update does change the underlying engine internals, we commit to shipping an updated adapter build and a migration note, so that the benchmark remains usable on current versions while previously released adapter builds remain archived for reference.

Upon publication, we will publicly release, under the MIT license: (i) the World Adapter compiled module and its versioned binary archive; (ii) the complete EPM Python project, including all agent framework implementations, prompts, and configuration files; (iii) all evaluation scripts and the automated scoring pipeline; and (iv) all task definitions (133 recipes, single- and multi-dish task suites, and the CookBench-IR instruction sets).

On the closed-source engine dependency. Absolute scores are conditioned on the pinned game build, which provides the high-fidelity physics the benchmark relies on. Reproduction therefore uses the game build and matching World Adapter version specified in the configuration guide. Three design choices bound this dependency. (i) **Architectural isolation**: all benchmark logic—task definitions, the action/composite-action API, state extraction, feedback, and scoring aggregation—lives in the open-source Layers 2–3, while Layer 1 is accessed only through stable, versioned engine APIs. (ii) **Version pinning**: as an offline single-player title, the pinned build can be kept installed and run indefinitely without accepting updates, and the exact build ID is documented, so a future game update does not invalidate the reference environment (see the version compatibility policy in Appendix B.2). (iii) **Thin adaptation layer**: even if the game version changes, re-validating the benchmark requires updating only the World Adapter hook layer (a few hundred lines of reflection code, which we commit to maintaining), not the benchmark itself.

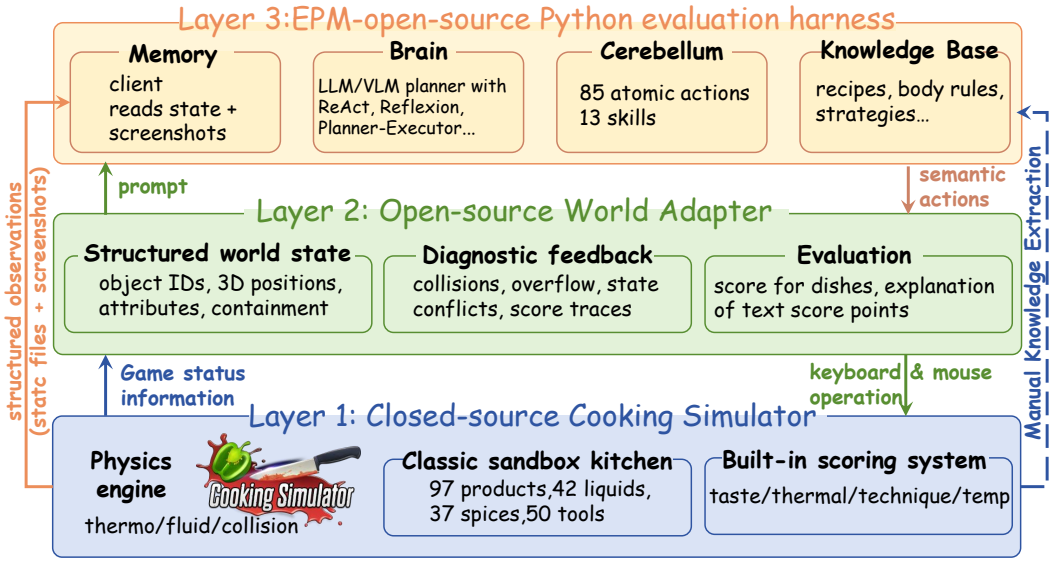

Figure 18: The CookBench reproducibility stack. Layer 1 is the closed-source *Cooking Simulator* engine base (pinned Steam build). Layer 2 is the World Adapter mod, injected via MelonLoader, which exposes structured world state, diagnostic feedback, and semantic action information through a real-time file bridge. Layer 3 is the open-source EPM Python harness that closes the loop: observe $\rightarrow$ plan $\rightarrow$ act $\rightarrow$ feedback, while archiving all logs. All benchmark-facing components (Layers 2–3 and the evaluation artifacts) are released under the MIT license.

Task knowledge and evaluation target. Target recipes and interaction rules are supplied to the agents, and the structured knowledge base (recipes, cookware manuals, embodiment rules) was manually collected and organized by the authors from in-game interaction. The dishes in *Cooking Simulator* also do not fully correspond to real-world recipes: ingredient quantities, procedures, and interaction constraints follow the game's own mechanics. Evaluation measures execution through the game's action interface under evolving object states: an agent that has memorized a recipe must still ground every step into instance-level physical actions under closed-loop feedback. The task set therefore evaluates execution with explicit domain knowledge rather than recipe recall. Our intent–execution analysis (Section 3.3) supports this empirically: agents attain high recipe-intent coverage (L1/L2, above 67%) yet fail sharply at action grounding and score-recognized completion (L3/L4), showing that possessing the textual knowledge does not translate into physical success.

C Knowledge-Base and Memory Artifacts: Representative Data Samples

Table 5 maps each released knowledge-base/memory artifact to the knowledge components defined in Section 2.3 and to the EPM memory dimensions (Appendix A). All files are plain text/JSON and are released with the benchmark. Below we show representative excerpts of the most important ones (lightly trimmed for space).

Real-time world state (`realtime_products.json`). This is the primary observation file continuously written by the World Adapter (Layer 2 in Figure 18) and read by the EPM client each step. The file contains a header with the timestamp, player position, and screen resolution, followed by one entry per live scene instance (745 instances in the Classic sandbox kitchen). Each entry exposes the instance's identity, 3D pose and bounding box, screen-space projection, visibility/interaction flags, containment, and distance to the agent:

```
{"timestamp": ..., "player_position": {...}, "total_products": 745,
 "products": [
  {"id": 146, "name_en": "Wine Vinegar", "kind": "liquids",
   "position": {"x": 6.173, "y": 0.138, "z": -2.267},
```

Table 5: Knowledge-base and memory artifacts released with CookBench, and their roles.

Artifact (file)	Knowledge component	Content
`action_catalog.txt`	Action knowledge	85 atomic actions in 25 categories + 13 composite actions, with signatures and parameter schemas.
`tools_manifest_openai.json`	Action knowledge (interface)	OpenAI-style tool-call schemas for every action and composite action.
`supported_items.txt`	Product knowledge	Static snapshot of all supported items (EN/CN names, categories).
`body_rules.txt`	Common sense (embodiment rules)	17 groups of hard interaction constraints injected into the planner's system prompt.
`strategy_notes.txt`	Experience knowledge	Score-oriented heuristics, failure-recovery priorities, and acquisition lists.
`action_feedback_catalog.txt`	Action knowledge (feedback)	Pre-/post-checks and error codes per action (e.g., `already_holding_item`).
`realtime_products.json`	World Adapter observation	Live snapshot of all scene instances: 3D pose/bounds, screen projection, visibility, containment, distance.
`task_progress.txt`	Long-term memory	Recipe-grounded milestone list with per-step status and optional atomic plans.
`stm_window.txt`	Short-term memory	Rolling window of the last N steps (action, result, errors, loop detection).
`long_horizon_history.txt`	Long-term memory (log)	Append-only JSONL record of every executed step.

```
   "bounds_min": {"x": 6.132, "y": 0.0, "z": -2.309},
   "bounds_max": {"x": 6.215, "y": 0.276, "z": -2.225},
   "screen_x": 1154.9, "screen_y": 548.6, "is_on_screen": true,
   "can_be_clicked": true, "container": "Scene", "distance": 5.66,
   "instance_id": -56820, "is_held": false, "rotation": {"y": 20.4, ...}},
  ... ]}
```

Companion streams in the same bridge provide the radar scan of nearby interactables (`realtime_radar_scan.txt`), camera parameters (`realtime_camera_info.txt`), and the current interaction mode/status (`realtime_interaction_status.txt`).

Action knowledge (`action_catalog.txt`).

```
[PickupObject] (4 actions)
- pick_up()  # pick up the aimed item when hands are empty
- pick_up_into_the_container()  # pick the aimed item into the held container
- pick_more(steps: int = 1)  # container multi-pick: increase pickup quantity
```

Action interface schema (`tools_manifest_openai.json`). Every action and composite action is also exposed as an OpenAI-compatible tool-call schema, so LLM agents invoke them as structured function calls:

```
{"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": ["x", "y"], "additionalProperties": false}}}
```

Common sense / embodiment rules (`body_rules.txt`).

```
3) The agent can hold only one item at a time; it may pick up the next item only after
   agent_state shows empty hands. If the held item is a container, carrying multiple
   items inside it is allowed.
5) The agent cannot pick up hot items with bare hands; use
   "hold container -> aim at hot item -> pick_up_into_the_container".
9) Navigate before interaction: before pick/place/cut/pour on an item, first use
   auto_navigation(...) to face it within interactable distance.
```

Experience knowledge (`strategy_notes.txt`).

```
7) Failure recovery priority: retry (small parameter perturbation) -> switch skill/path
```

```
   -> fallback to safe state -> optional reset.
12) Score priority: the final score mainly depends on the submitted container's content
    and state (seasoning ratio, heat treatment, processing procedure, serving
    temperature); a container with partially correct content still receives partial score.
```

Product knowledge (`supported_items.txt`). A static snapshot of every supported item with its English/Chinese name pair (Chinese glosses omitted below) and category:

```
{"name_en": "Avocado Oil",      "name_cn": "...", "category": "liquids"}
{"name_en": "Balsamic Vinegar", "name_cn": "...", "category": "liquids"}
```

Long-term task memory (`task_progress.txt`). The planner's progress is persisted as a recipe-grounded milestone list; each milestone carries a status and, optionally, a decomposed atomic plan:

```
task:  {recipe_id: "1", recipe_name: "Lemon Chicken Breasts", goal_description: "Add to a
  Pot: Olive Oil [50ml], ... Serve hot."}
high_level_todos:
  - {id: "H1", title: "Add to a Pot: Olive Oil [50ml], White Wine [10ml], ...",
     status: doing, recipe_step_ref: "recipe.text.sentence.1"}
  - {id: "H2", title: "Fry for 60s", status: todo, recipe_step_ref:
  ↪  "recipe.text.sentence.2"}
```

Short-term memory (`stm_window.txt`). A rolling window records each step's action, outcome, and error code, plus loop-detection state:

```
- step_id: 2
  action_or_skill: "move_forward"   params: {"duration": 0.3}
  result_summary: "success"         errors: ""
loop_detection: {loop_detected: false, consecutive_failures: 0,
                 last_error_types: ["invalid_plan:unknown_action:move_to"]}
```

Per-step execution log (`long_horizon_history.txt`). An append-only JSONL record per step constitutes the full audit trail of a run, so that every reported result can be traced back to the exact interaction sequence that produced it.

D Additional Benchmark Details

D.1 Column Definitions for the Benchmark Comparison Table

Table 1 uses the following column definitions. **Seq. Len.**: average number of high-level action steps per task. **Auto. Eval.**: whether the benchmark provides built-in automatic evaluation. **Simulator**: the primary operating environment or interaction interface exposed to the agent. **3D Env.**: whether the benchmark provides an interactive embodied 3D environment. **Objs / Func. Items**: the count of interactive objects and functional items; we define a functional item by its role—an item is counted only if it can execute a specific function on another object (informally, cookware and utensils; the interface groups these entries under *utensils*), which avoids confusion with the tool-calling APIs of digital agents. For CookBench, both counts are taken from the released agent-interface vocabulary (Table 11): 176 interactive objects (97 products, 37 spices, and 42 liquids) and 50 functional items (9 labeled *tools* and 41 *utensils* in the interface); it additionally exposes 27 interaction anchors (appliance switches, place points, and hooks) and 171 placement points across 8 platform categories (table centers, side tables, shelves, and grill/griddle surfaces), which are not counted as objects. These anchors and placement points make complex appliances and workspaces addressable at the component level (e.g., individual stove burners, per-compartment oven place points, and fryer slots rather than a monolithic appliance object); VRKitchen likewise composes objects from parts that carry their own affordances, and CookBench's distinction is the exposure of per-component place points and interaction anchors within continuous long-horizon cooking processes, so we report them separately rather than inflating the object or functional-item counts. **Actions**: the size of the benchmark-exposed action interface; for CookBench, 98 = 85 atomic actions + 13

Table 6: 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.

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

composite actions. **Temp / Time / Volume / Shape**: whether the benchmark explicitly simulates these physical state dimensions with observable, task-relevant consequences (e.g., continuous heating progress, overcooking, or irreversible shape changes); a purely categorical object flag without physical process simulation (e.g., a binary "hot" attribute) does not count. **Physical State Change**: whether state evolution is primarily discrete or continuous. **Manip. Gran.** (manipulation granularity): *Coarse* = only high-level semantic task primitives such as "pick up X"; *Fine* = only low-level continuous control such as trajectory- or parameter-level adjustment; *Coarse+Fine* = both layers coexist. **Knowledge**: the dominant knowledge type required by the task setting—*Common* = everyday commonsense; *Domain* = task-domain procedural knowledge such as recipes; *Tool* = tool-affordance and tool–operation pairing knowledge.

CookBench uniquely combines both manipulation granularities: encapsulated semantic actions and composite actions (coarse) coexist with continuous, parameter-level control over physical quantities such as poured volume, heating duration, and target cut size (fine). Since CookBench does not provide ground-truth execution trajectories, its Seq. Len. is an oracle-style estimate derived from a recipe-to-action analysis; the estimation procedure and per-category step-length distributions are detailed in Appendix D.3.

D.2 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 6 shows one representative entry from the single-dish recipe pool.

D.3 Oracle Step-Length Analysis

CookBench does not ship ground-truth execution trajectories, so we derive oracle-style step-length estimates from structured gold annotations instead of replaying real runs. The pipeline has two stages. First, each recipe is decomposed into *gold culinary subtasks*—semantically meaningful cooking milestones such as adding a required amount of liquid, seasoning a specific ingredient, cutting an item, heating an ingredient to a target state, or transferring a prepared component to the final serving plate. The decomposition is performed by `deepseek-v4-pro` under a fixed annotation protocol (prompt template and request builder released with the benchmark: `build_recipe_gold_reference_requests.py` and `run_semantic_llm_annotations.py`), and each subtask is annotated with its operation type, targets, contextual objects, and explicit success criteria. Second, each subtask is aligned with the EPM interface by expanding it into three action segments: *setup* actions (e.g., scene query, navigation, pickup, entering the correct interaction mode), the *core* action or composite action implementing the culinary operation, and *completion/verification* actions (exiting the mode, container handling, state verification). The expanded step length of a dish is the total number of EPM actions and composite actions across all segments, representing the expected end-to-end effort of a compe-

tent agent rather than the trace of any particular run. Table 7 reports the per-category distributions over all 133 tasks.

Table 7: Oracle step-length statistics per cooking-style category over the 133 single-dish tasks. *Subtasks*: number of gold culinary milestones per dish. *Steps*: expanded EPM action/composite-action length. Across all tasks, dishes contain 25.2 subtasks on average (range 9–49) and expand to a mean of 132.4 steps (median 132, range 50–249), which is the basis for the $>$130-step long-horizon claim in Section 2.

Category	n	**Subtasks**	**Expanded EPM steps**		
		Mean (range)	Mean	Median	Range
Grill-Single	22	22.7 (12–48)	120.1	110.5	50–233
Grill-Combo	24	25.9 (13–48)	136.5	129.0	60–249
Pan-Stir	23	27.1 (12–41)	140.2	135.0	56–211
Soup-Stew	25	23.9 (9–41)	126.2	143.0	53–197
Pasta-Assembly	23	25.8 (16–49)	135.7	138.0	75–233
Salad-Cold	16	26.1 (13–38)	136.9	124.0	66–191
All	133	25.2 (9–49)	132.4	132.0	50–249

Two observations follow. First, every category averages around or above 120 expanded steps, and the overall mean (132.4) confirms that long horizons are a property of the task design rather than of a few outliers; the longest dishes reach 249 steps (e.g., Barbecue T-bone Steak with Potatoes and a Hot Corn on the Cob, 48 subtasks). Second, the estimates align with the empirically observed difficulty pattern: the two categories with the highest expanded-step means (Pan-Stir, Grill-Combo) are also among the hardest empirically (Appendix D.4). These estimates further guided the runtime budgets used in our experiments (primary cutoff at 200 steps, hard cap at 250), which cover the estimated length of every task. The per-dish gold annotations and step-length tables are released with the benchmark data.

D.4 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

Table 8: Task-style characterization of the six cooking-style categories. *Steps*: average number of recipe sentences. *Mean*: average score over the eight non-human agent–model settings (percent). *Zero*: fraction of zero-score dish–setting cells. *Univ.-0*: dishes scored zero by all eight settings. Statistical patterns refer to the tests in Appendix F.9; category-winner patterns beyond the validated pairs are descriptive only.

Category	Dominant task-style demands	n	Steps	Mean	Zero	Univ.-0	Distinctive empirical pattern
Grill-Single	Roasting timing of a single hot main; precise heat-duration control	22	7.7	5.6	73.3%	2	Strongest model effect (Cochran's $Q = 22.4$, $p = 0.002$ over the eight settings; $Q = 21.9$, $p < 0.001$ over the five CaP backbones); GPT-5.4 and Gemini-3.1-Pro significantly exceed Kimi-k2.6
Grill-Combo	Multi-component coordination and final plate assembly	24	7.7	3.3	74.0%	7	Hardest category; contains half (7/14) of all universally-failed dishes
Pan-Stir	Local skillet sequencing; short-window heat control in a single pan (fewest containers, 1.0)	23	7.1	4.6	72.8%	1	No detectable within-category setting differences
Soup-Stew	Liquid management, boil timing, staged ingredient addition, multi-phase convergence	25	10.2	5.1	66.0%	0	Most procedurally extended; significant model effect (Cochran's $Q = 17.9$, $p = 0.013$ over the eight settings; $Q = 10.7$, $p = 0.03$ over the five CaP backbones); Kimi-k2.6 > Qwen3.6-35B
Pasta-Assembly	Linear pipeline of staple-base preparation plus topping assembly	23	6.5	9.5	57.1%	2	Most tractable category; fewest steps
Salad-Cold	Cutting, ingredient selection, and cold assembly without thermal control	16	8.3	3.5	78.9%	2	Highest zero-score rate despite the absence of heating (descriptive)

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.

Table 8 summarizes the six categories at the task-style level, pairing each category's dominant demands with its empirical profile. Two contrasts stand out. First, difficulty does not follow the presence of heating alone: Salad-Cold involves no thermal control yet exhibits the highest zero-score rate, while the most thermally demanding category (Soup-Stew) is the only one with no universally-failed dish; instead, the procedural and coordination structure of a category—linear pipelines (Pasta-Assembly) versus multi-component synchronization (Grill-Combo)—better predicts agent success. Second, the two categories with significant within-category model effects (Grill-Single and Soup-Stew) are precisely those dominated by a single core competence—heat-duration control and staged liquid convergence, respectively—suggesting that category-level model differences surface where one capability dominates the task.

The complete ID-to-name mapping for all single-dish tasks is listed in Table 9. Table 11 further provides representative examples from the interactive object vocabulary exposed by the environment.

D.5 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`.

Table 9: Complete list of the 133 single-dish tasks used in CookBench. IDs correspond to the cooking-style taxonomy in Appendix D.4.

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: Composition of the CookBench interactive object vocabulary. *Objs* and *Func. Items* correspond to the two columns of Table 1; interaction anchors and placement points are exposed by the interface but not counted as objects.

Category	Counted as	#	Examples
Products	Objs	97	Steak, Salmon Fillet, Asparagus, Cheddar
Spices	Objs	37	Salt, Black Pepper, Cumin powder, Turmeric
Liquids	Objs	42	Olive Oil, Soy Sauce, Water, Chicken Broth
Tools & Utensils	Func. Items	50	Knife, Pan, Big Pot, Oven, Refrigerator
Interaction Anchors	—	27	Stove Switch, Sink Place Point, Hook Point
Placement Points	—	171	Large Table Center, Side Table, Top Shelf, Grill Surface
Total interface entries		**424**	

- **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.
- **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_id`s, requiring explicit target disambiguation.

E 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 difficulty, CookBench removes the assumption of perfect execution prevalent in prior benchmarks, introducing 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 large state space and extended horizons, requires agents to rely heavily on closed-loop feedback, which substantially raises the overall task difficulty.

E.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 composite actions, while interacting with hundreds of ingredients, liquids, seasonings, functional items, 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.

E.2 Operation Logic and Affordance Constraints

Cooking requires domain-specific affordance reasoning. Agents must know which functional item 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 19, 20, and 21 show representative cases.

Table 11: Interactive object vocabulary supported by the CookBench environment (253 interface entries; *Food Processor Container* appears in two interface roles). Entries are grouped by category: *Products* (97), *Spices* (37), *Liquids* (42), *Tools and Utensils* (50), and *Interaction Anchors* (27, not counted as objects). The first four groups correspond to the Objs (176) and Func. Items (50) columns of Table 1. The additional 171 placement points are generated from 8 platform templates (e.g., *Large Table Center-1* to *-56*) and are summarized in Table 10 rather than listed here.

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

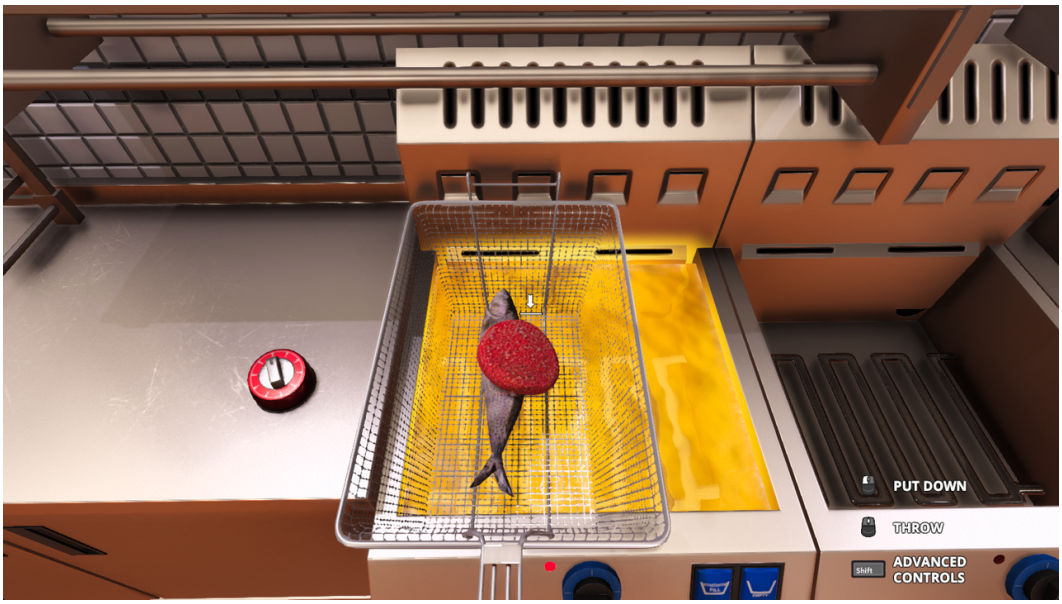

Figure 19: Functional-item–appliance affordance. Operating the deep fryer requires the fryer basket rather than direct placement, so a semantically correct "deep-fry" intention still fails without the correct functional-item chain.



Figure 20: Container-material constraint. A solid-only container cannot hold a liquid, which forces agents to distinguish generic "container" semantics from physically valid containment affordances.

E.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 22, 23, and 24 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.



Figure 21: Body-rule constraint. Hot ingredients cannot be directly picked up by hand; successful recovery requires using an appropriate functional item or container rather than retrying the same pickup action.



Figure 22: 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.

E.4 Physical Simulation Anomalies and Irreversible Failures

The simulator also introduces physical anomalies that are hard to capture with pure recipe logic. Figures 25, 26, 27, and 28 show representative cases in which object fragility, functional-item 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.



Figure 23: Temporal state-memory failure. Compared with Figure 21, 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.



Figure 24: Appliance degradation and recovery. Some appliances can become unusable after extended operation and require repair, introducing an exception-handling branch that must be integrated into the remaining cooking plan.

E.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; cookware knowledge for capacities and functional-item–operation pairings; embodiment-rule knowledge for hand limits, modes, placement rules, and navigation-before-

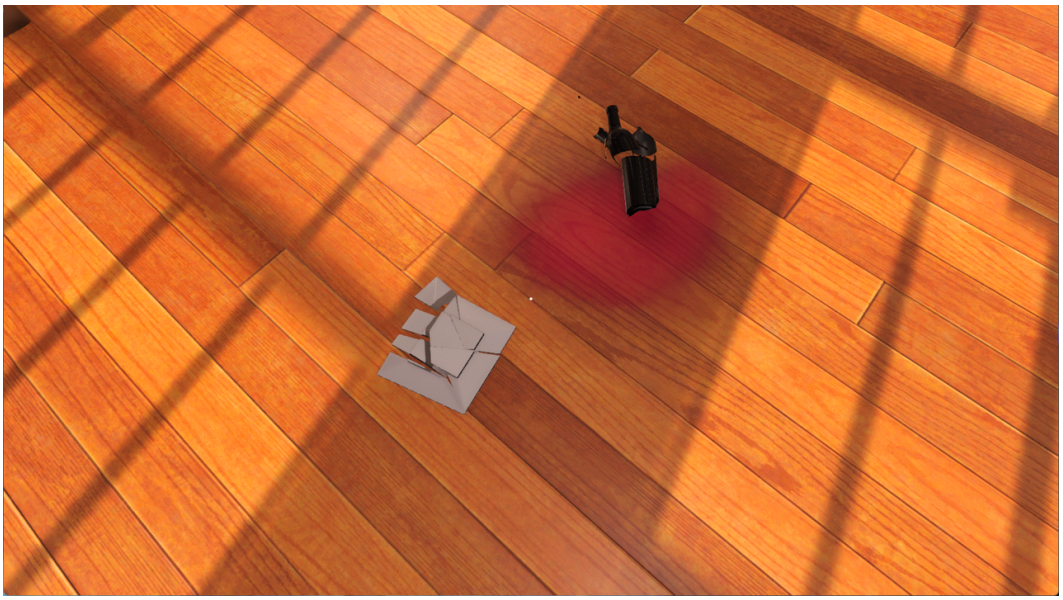

Figure 25: Fragile-object failure. Bottles and plates may break after collision or dropping, turning a local placement or navigation mistake into an irreversible loss of functional items, containers, or ingredients.



Figure 26: Geometry-sensitive functional-item use. Object shape affects whether a functional item can successfully manipulate the target, so agents must reason about physical geometry rather than only object categories.

interaction; and strategy knowledge for timing, recovery, and score-oriented prioritization. The same surface failure may arise from different mechanisms, as summarized in Figure 29.

F Additional Experimental Details

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



Figure 27: 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.

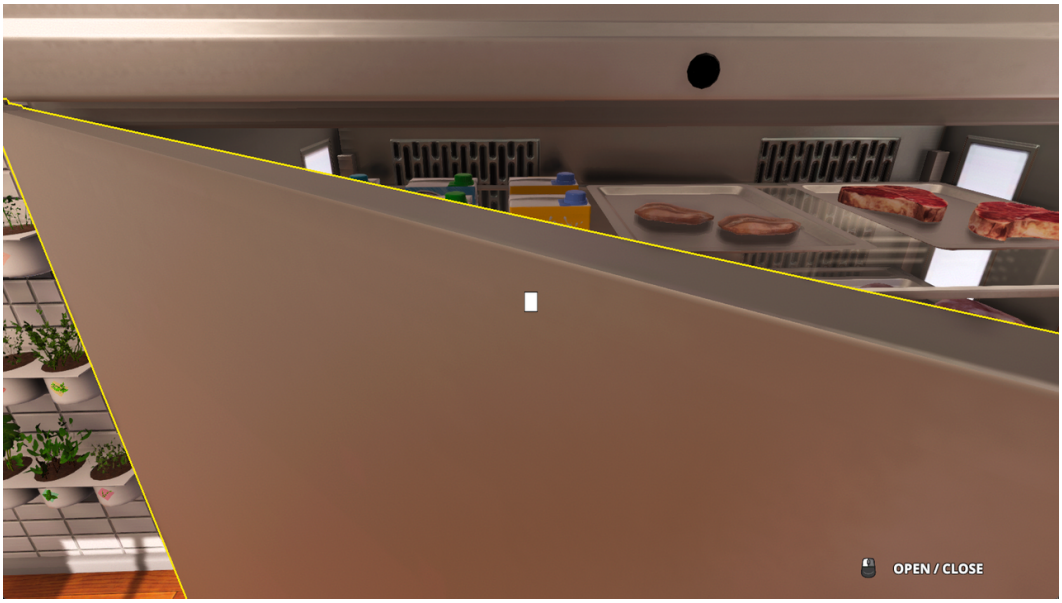

Figure 28: Embodied self-interference. The agent's own body position can block an appliance or door, so successful interaction may require repositioning before the intended open/close action becomes physically executable.

F.1 Agent Framework Implementation Details

To support controlled comparison across heterogeneous agent designs in Section 3, 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/composite-action API, execution feedback channel, and memory-store backbone, differing primarily in their control flow. In

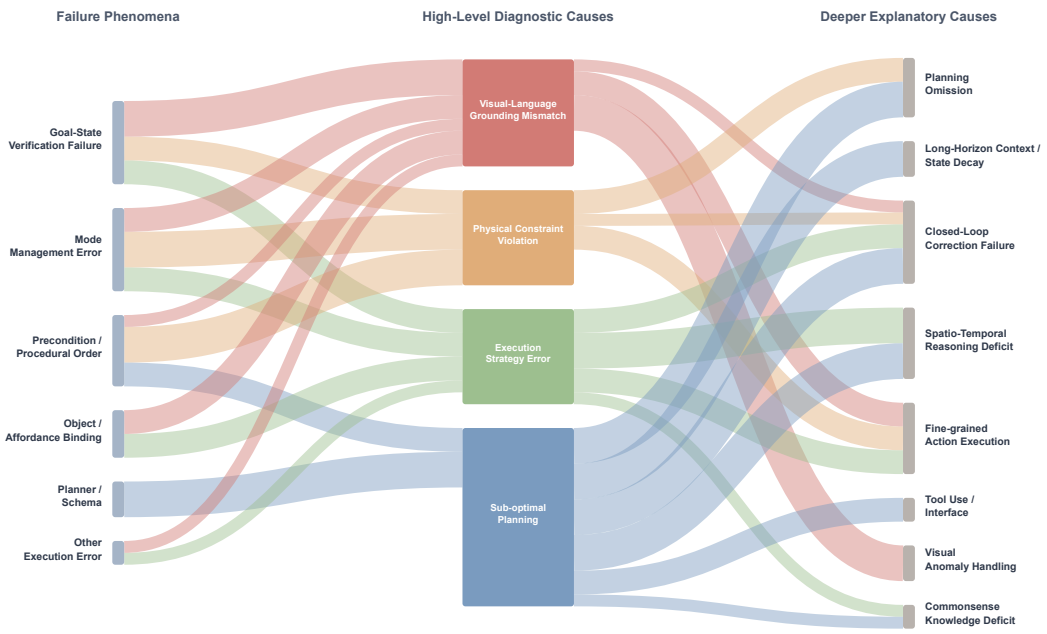

Figure 29: Mechanism-level interpretation of failure phenomena. The Sankey-style diagram separates observable failure phenomena from deeper diagnostic causes, illustrating that one surface error can arise from multiple mechanisms such as weak physical grounding, invalid mode or body-rule modeling, poor repair after feedback, and state-memory degradation.

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 12.

- **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-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, composite actions, 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 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/composite-action 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. We retain the CaP name because the prompting style follows the Code-as-Policies formulation; the generated program is never executed as runtime control flow, and its program variables do not persist across regenerations—all persistent state lives in EPM's external runtime record.

F.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 or meaningless repeated retries. The same supervisor is applied uniformly across all methods. In the reported configuration, normal execution can use up to 200 steps before force submission is armed; this limit is intentionally set above the oracle step-length estimate of the tasks (Appendix D.3), so the cap itself does not preclude successful completion—in practice, no agent run completed its dish within the 200-step budget. Once the executed step index reaches this threshold, EPM records a deadline 50 steps later, yielding the 250-step

Table 12: Capability comparison of the five adapted agent frameworks. Check marks indicate that the capability is explicitly enabled in our EPM-based implementation.

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

hard cap; during this finalization buffer the interface is restricted to finalization-only primitives (navigation, pick-up/put-down, pouring, and submission), so an agent cannot continue cooking covertly after the limit. 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`, `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 containers, records whether each probe is nonempty, counts observed contents, checks name matches and pickup signals, and selects the best confirmed nonempty submission container. If a nonempty transfer-source container is found while the preferred final container 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. Force submission operates only through the same embodied action/composite-action interface and submits containers that the environment makes reachable and observable.

Termination breakdown. We analyze the termination reasons of the strongest setting, *CaP@GPT-5.4*. Its run archive contains 138 logged episodes; after excluding 30 infrastructure aborts (network failures, manual stops, and crashed runs), 108 completed episodes remain. Of these, 77.8% terminated at the step limit: 25.0% (27) ended with a successful submission during the 50-step finalization buffer, 32.4% (35) exhausted the 250-step hard cap without a submission, and 20.4% (22) entered forced submission but failed—of the 22 failed forced submissions, 14 found no confirmed nonempty submittable container (the agent had lost or never plated its food) and 8 tripped the consecutive-failure guard, most often on navigation failures while carrying the dish to the checkout stand. Only 22.2% (24) of episodes ended with a voluntary submission before the 200-step limit. The step cap therefore determines *when* episodes end, but it is not the binding constraint on scores: the cap is set above the oracle step-length estimate (Appendix D.3), and the dominant failure mode is that agents cannot convert the available steps into completed cooking progress.

Table 13: Summary of experimental runtime and pipeline settings.

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 with no prior game experience under native keyboard-and-mouse control, plus a disjoint 20-volunteer human-through-harness control group; dishes partitioned; up to 3 attempts per dish, mean score recorded	Provides a score and completion-time performance reference under human cooking behavior, and isolates the interface contribution to the human–agent gap (Appendix F.4).
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 14: Exact model versions used in all experiments. Proprietary models were accessed via their official APIs; Qwen3.6-35B is the open-weight variant. All scores in this paper are conditioned on these pinned versions.

Paper label	Official model ID	Provider	Release / pinning
GPT-5.4	`gpt-5.4`	OpenAI	Released March 5, 2026.
Gemini-3.1-Pro	`gemini-3.1-pro-preview`	Google	Released February 19, 2026.
Kimi-k2.6	`kimi-k2.6`	Moonshot AI	Official API model ID; announced April 21, 2026.
Qwen3.6-Plus	`qwen3.6-plus-2026-04-02`	Qwen Cloud	Date-pinned official version.
Qwen3.6-35B	`qwen3.6-35b-a3b`	Qwen	Released April 16, 2026.

F.2 Experimental Configuration

Table 13 summarizes the main runtime and pipeline settings used in the experiments, and Table 14 reports the exact official model IDs and release or pinning information for every evaluated backbone. Table 15 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.

Table 15: Agent-pipeline settings used for the framework comparison.

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_executd`	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.

F.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.

F.3.1 Natural-Language Instructions

CookBench-IR provides 14,394 bilingual instructions (7,197 in Chinese and 7,197 in English) for studying the mapping from user requests to the intended target dish list and, when relevant, the correct dish order. The embodied experiments in this paper use oracle structured goals, so the reported execution scores are not confounded by natural-language instruction parsing errors.

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 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.

F.4 Human Baseline Protocol

Participants and procedure. We recruited 20 volunteers, none of whom had played *Cooking Simulator* before the study, so no participant brought prior familiarity with the game's controls, physics, or recipes. The 133 single-dish recipes were partitioned among the volunteers such that every dish was covered. Participants played with the game's native keyboard-and-mouse controls and its standard first-person visual interface. They received no recipe preview, walkthrough, or practice session before their attempts: the recipe was available only through the game's standard in-game presentation, exactly as a first-time player would encounter it. Each dish was attempted up to three times by its assigned volunteer, and the mean score across the attempts was recorded. Human

dishes were scored by the identical simulator scoring pipeline used for agents, and completion times were measured end-to-end per attempt.

Human-through-harness control group. To isolate the contribution of the interface itself, we recruited a second, disjoint cohort of 20 volunteers—none had played *Cooking Simulator* before, and none overlapped with the keyboard-and-mouse group. These participants operated through the exact same harness semantic interface as the agents: the full API of 85 atomic actions and 13 composite actions, invoked as text commands through a command-line console, with the same oracle structured observations (`realtime_products.json` snapshots and radar scan), the same RGB screenshots, and the same textual execution feedback and error codes; the only difference from the agent setting is that a human, rather than a VLM, decides which command to type next. The 133 single-dish recipes were again partitioned so that every dish was covered; each dish was attempted up to three times, and we report the mean score and mean end-to-end completion time across attempts, under the same 250-step episode cap and the identical simulator scoring pipeline. Completion time here is wall-clock time per attempt, including the time participants spend reading textual observations and deciding and typing the next command; it measures interface overhead and is not comparable to the agent execution-time metric, which excludes deliberation latency. As Table 16 shows, this control group remains uniformly high (per-category means 91.75–94.85, overall 93.34)—an order of magnitude above every agent setting—while completion times rise by 12.8–38.4 minutes per dish relative to the keyboard-and-mouse group (category-mean differences; overall 32.23 vs. 7.34 minutes), reflecting the cost of textual command entry rather than any loss of task competence. Operating through the harness interface therefore does not by itself reproduce the agent collapse, directly falsifying the interface-confusion explanation of the human–agent gap.

On comparability with agent conditions. The human and agent conditions are intentionally different, and we report the human rows as a *performance reference*—evidence that near-perfect scores are achievable in the environment—rather than as a controlled head-to-head comparison. We take the harness-interface cohort as the primary reference because it shares the agents' interface, but we still caution against reading the $\sim$93% vs. $\sim$8% gap as a pure measure of cognitive deficit: it aggregates perception differences (full first-person vision versus structured observations plus screenshots), protocol differences (three human attempts per dish versus one agent run), and any residual familiarity effects. Three considerations nonetheless indicate that the gap is not primarily an interface artifact. First, the human-through-harness control group directly isolates the interface contribution: humans operating through the identical semantic API, observations, and feedback as the agents still score 93.34 on average, only a few points below the keyboard-and-mouse cohort, so the interface itself cannot explain the gap. Second, the agent interface is not strictly harder than the human one: EPM supplies oracle structured observations and encapsulated composite actions (e.g., `auto_cut`, `auto_pour`, `auto_navigation`) that remove exactly the low-level localization and manipulation burden humans must perform by hand, so interface difficulty cuts in both directions. Third, our capability-gap conclusions do not rest on the aggregate score gap: the milestone-level intent–execution analysis (Appendix G.2) localizes failures in the conversion of recipe-grounded intentions into executed and score-recognized milestones—a cognitive locus that interface differences cannot explain.

Per-category results. Table 16 reports the per-category human performance reference. Human scores are uniformly high across all six categories under both interfaces (96.00–97.63 keyboard-and-mouse; 91.75–94.85 through the harness), with no category showing the collapse observed for agents, indicating that the category-level difficulty ordering observed for agents reflects agent-side capability limits rather than unscorable tasks. The harness control group's scores sit only 2–5 points below the keyboard-and-mouse group in every category, while its completion times are consistently higher (category-mean increases of 12.8–38.4 minutes per dish): the textual command interface slows humans down but does not degrade what they can achieve.

F.5 Planning Cost and Token Usage

Table 17 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

Table 16: Human performance reference, reported separately from the agent settings because the protocol intentionally differs (up to three attempts per dish with the mean score recorded); the *Human (Harness)* row also appears as the reference row of Table 2. *Human (K&M)*: native keyboard-and-mouse control. *Human (Harness)*: a disjoint cohort operating through the same harness semantic API (85 atomic actions + 13 composite actions, text command-line console), structured observations, and textual feedback as the agents. These values calibrate the score ceiling and are not a controlled head-to-head comparison. The *Avg.* columns report the mean over all 133 per-dish mean scores (equivalently, the dish-count-weighted mean of the category means), the same averaging unit as the agent rows of Table 2. Time is end-to-end wall-clock minutes per attempt—for the harness group including observation reading and command entry—and is not comparable to the agent execution-time metric in Table 2, which excludes deliberation latency.

	Grill-Single		**Grill-Combo**		**Pan-Stir**		**Soup-Stew**		**Pasta-Assembly**		**Salad-Cold**		**Avg.**	
	Score	**Time**	**Score**	**Time**	**Score**	**Time**	**Score**	**Time**	**Score**	**Time**	**Score**	**Time**	**Score**	**Time**
Human (K&M)	97.63	6.64	97.47	7.66	96.67	7.90	96.74	7.23	96.00	7.74	96.10	6.61	96.80	7.34
Human (Harness)	94.85	24.84	93.12	46.06	92.40	34.40	94.60	40.33	91.75	23.34	93.30	19.41	93.34	32.23

Table 17: 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.

Setting	**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**	
	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

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 similar 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 17 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.

F.6 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 component 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.

F.6.1 Score Validity Against Independent Progress Annotations

Conclusion. The official score is a valid measure of cooking-task progress: it discriminates complete failures from genuine progress, scales with the amount of completed work, preserves meaningful gradations among partial successes, and each of its four quality dimensions (taste, thermal processing, technique, serving temperature) individually responds to independently annotated task

progress. Residual variance is structurally explainable rather than arbitrary (see below). The remainder of this section describes the analysis pipeline and reports all statistics.

Analysis pipeline. The analysis proceeds in four steps on the 961 runs that have a simulator feedback record, milestone annotations, and at least one milestone judgment whose score-recognition label is definitive; these runs supply the 19,728 judgments entering the score–coverage correlation (Appendix G.2). **(1) Score extraction.** The official per-run score is the taste-anchored quality score, $\mathcal{P} = 100\,\mathcal{Q}$ with $\mathcal{Q} = \texttt{taste.score}$; thermal correctness, technique, and serving-temperature requirements already enter $\mathcal{Q}$ as component-level dimension scores and caps (see the aggregation above). **(2) Milestone coverage.** From the LLM annotations we compute per-run coverage at each ladder level as the mean of the label scores ($\text{yes} = 1$, $\text{partial} = 0.5$, $\text{no} = 0$) over the dish's gold subtasks: L1 component intent, L2 subtask intent, L3 action grounding (annotated from execution logs, independent of the scoring pipeline), and L4 score-recognized completion (judged from the feedback record, hence partially circular and used only as a consistency check). Judgments the annotator left as `unknown` are excluded from the coverage mean at that level rather than counted as failures. **(3) Dimension scores.** For each run we extract the four component-level dimension scores from the feedback record and aggregate each dimension as a component-weighted mean over the components for which that dimension has nonzero weight:

$$\bar{s}^{(d)} = \frac{\sum_k \omega_k\, w_k^{(d)}\, s_k^{(d)}}{\sum_k \omega_k\, w_k^{(d)}}, \qquad d \in \{\text{taste}, \text{thermal}, \text{technique}, \text{temperature}\},$$

where ω_k is the component weight, $w_k^{(d)}$ the dimension weight, and $s_k^{(d)}$ the component's dimension score. **(4) Statistical tests.** Because the score distribution is zero-inflated and skewed, all association tests are rank-based.

Metrics. We report four complementary statistics. **Spearman's** ρ is the Pearson correlation of the rank transforms of two variables, measuring monotone association without distributional assumptions. **Kendall's** $\tau = (C - D)/(C + D)$, where C and D count concordant and discordant run pairs (a pair is concordant if the run with higher milestone coverage also has the higher score); we use it to test the dose–response trend across coverage deciles (runs sorted by coverage, split into ten equal groups, group mean scores compared). **Mann–Whitney** U (one-sided) compares the coverage distributions of zero-score and nonzero-score runs via rank sums. **OLS** R^2 measures how much score variance is jointly explained by the four dimension scores and milestone coverage; ΔR^2 is the increase when milestone coverage is added to the dimension-only model.

Results. Table 18 summarizes all statistics.

The evidence holds at three granularities, from coarse to fine. At the coarsest level, zero-score runs show drastically lower score-recognized milestone coverage than nonzero runs, and only 7.1% of zero-score runs managed to ground at least half of the gold subtasks into actions, so zero scores overwhelmingly reflect genuinely incomplete dishes rather than harsh thresholds. At the intermediate level, the relationship is graded rather than binary: across deciles of score-recognized milestone coverage, the mean game score rises from 0.8 (bottom decile) to 14.8 (top decile) with a strongly significant monotone trend. At the finest level, the dependence persists within nonzero-score runs as well, so the fine-grained score differences among partially successful runs still track task progress.

The correlations are moderate rather than near-perfect, which is expected and structurally explainable: the milestone annotation measures the middle of the intent-to-score chain—how much recipe-grounded progress was intended and grounded into actions—but it does not capture two downstream requirements of scoring: (i) whether the executed actions physically succeeded at the quality level (correct doneness, flavor balance, cutting form), which the taste, thermal, technique, and temperature dimensions price separately, and (ii) end-of-episode plating and submission, since progress that is never transferred into a submitted container cannot be scored at all (cf. the forced-submission failures in Appendix F.1.1, where 20.4% of *CaP@GPT-5.4* episodes ended without a submittable nonempty container). A near-perfect correlation would instead indicate that the score merely replays milestone completion and adds no information beyond it.

Table 18: Score-validity statistics on the 961 runs that have a simulator feedback record, milestone annotations, and definitive score-recognition labels. All reported p values are far below 0.005 unless stated otherwise.

Analysis	**Statistic**	**Result**	p
L1 component intent vs. score	Spearman ρ	0.10	2×10^{-3}
L2 subtask intent vs. score	Spearman ρ	0.14	2×10^{-5}
L3 action grounding vs. score	Spearman ρ	0.15	2×10^{-6}
L4 score-recognized vs. score	Spearman ρ	0.33	2×10^{-25}
L4 vs. taste dimension	Spearman ρ	0.21	4×10^{-11}
L4 vs. thermal dimension	Spearman ρ	0.20	4×10^{-9}
L4 vs. technique dimension	Spearman ρ	0.28	3×10^{-18}
L4 vs. temperature dimension	Spearman ρ	0.27	6×10^{-17}
L3 vs. four dimensions	Spearman ρ	0.14–0.17	$<10^{-5}$
Dims. + L4 $\rightarrow$ score	OLS R^2	0.504	—
Dims. only $\rightarrow$ score	OLS R^2	0.480	—
Zero vs. nonzero L4 coverage	Mann–Whitney U	6.6% vs. 12.8%	$<10^{-20}$
Zero-score runs with L3 ≥ 0.5	proportion	7.1%	—
Coverage-decile dose–response	Kendall τ	0.27 (mean 0.8 $\rightarrow$ 14.8)	$<10^{-24}$
Within nonzero runs, L4 vs. score	Spearman ρ	0.29	6×10^{-7}

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

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

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

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

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

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

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

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

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

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

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 $[x_b, y_b]$:

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

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

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

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

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

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.

Weight provenance. All parameters of the scoring pipeline—the part weights q_i, the per-component dimension weights $(w_t, w_h, w_\tau, w_\theta)$, the component weights ω_k, the tolerance ranges and target values of each dimension, and the penalty constants $A = 2.5$ and $B = 10$—are the simulator's native recipe-rule design values, authored as part of the environment's recipe data and read directly from its scoring feedback records. CookBench does not introduce, tune, or fit any scoring parameter; the only benchmark-level decision is to report the quality score $\mathcal{Q}$ rather than the game UI's composite score that additionally mixes in realization time. We verified every formula in this appendix against the simulator's actual outputs: for all non-zero dish-penalty cases in our run archive, the piecewise penalty formula reproduces the returned value exactly (e.g., $U = 15, S = 390 \Rightarrow \pi = (U/S)^2 = 0.0015$; $U = 60, S = 390 \Rightarrow \pi = E(U/S) = 0.284$).

Dimension weights are per-recipe design choices rather than text-derived heuristics: a hot main component such as Lemon Chicken Breasts carries $(w_t, w_h, w_\tau, w_\theta) = (1, 1, 1, 1)$, while its cold fruit-salad side carries $(0, 0, 1, 0)$ and is scored only on technique, so dimensions irrelevant to a component never contaminate its score. The remaining parameters are equally concrete. For the technical dimension, the recipe's requirement to cut an 80g lemon into 16g pieces is instantiated as a full-credit interval of $[9, 22]$g per piece and a half-credit interval of $[5, 26]$g: pieces of 20g

score 1, pieces of 24g score 0.5, and pieces of 32g score 0. For the temperature dimension, the same recipe specifies an expected serving temperature of 100°C with a ± 25°C tolerance under the `checkGoodOrGreater` rule, so a dish served at 20°C scores $20/100 = 0.2$. For the taste dimension, each flavor channel carries a per-gram target (e.g., thyme at 2.19/g, so a 72g mixture has target value 157.7) together with three native tolerance bands (perfect/good/eatable, here 68.4/104.4/128.9); a deviation within a band receives the band's interpolated score, and a deviation beyond the eatable band—such as a completely missing chicken flavor, deviation 8494.7 against an eatable limit of 6957.5—scores zero. For the thermal dimension, required heating types carry native target intensities (e.g., frying with target 100 for the chicken breast), and unheated products ($\mathrm{Is} = 0$) score zero. These examples are taken verbatim from actual feedback records of benchmark runs, confirming that every quantity in the scoring formulas is a native design value rather than a benchmark-side choice. Our human reference runs—averaging 96.8 over all 133 dishes—show that the environment is stable enough to support near-perfect execution, so the scoring pipeline measures agent capability rather than environment noise.

Calibration. We validate the scoring function from three directions. First, *ceiling attainability*: human operators achieve a mean score of 96.8 over all 133 dishes with native controls and 93.34 through the EPM text interface (median 98.2, minimum 84.3, and only 8 dishes below 90 under native controls), so near-perfect scores are attainable and the metric is calibrated at the top of the range. Second, *agreement with verified task completion*: correlating each run's final score with its LLM-verified gold-milestone completion rate (Appendix G.2) yields a moderate, highly significant correlation (Spearman $\rho = 0.42$, $p < 10^{-41}$) that remains significant within every one of the eight agent–model settings individually ($\rho = 0.38$–0.53, all $p < 10^{-5}$); the mean score rises broadly from 0.7 for runs with no verified milestone completion to 18.4 for runs completing over 75% of gold milestones. This correlation is computed on the 961 runs defined above and spans 19,728 of the 23,743 milestone-level judgments; the remaining 4,015 judgments lack a definitive score-recognition label and are used only for the L1–L3 coverage statistics. Higher benchmark scores therefore reliably reflect more verified task completions rather than exploitation of the scoring function. Third, *graded sensitivity in the low and middle range*: every dimension mapping is continuous and (piecewise) linear in the magnitude of the deviation, so scores degrade smoothly rather than collapsing at binary cutoffs. Concretely, an under-heated product earns a thermal score that rises linearly with the achieved intensity (v/p), while over-heating decays as $1-(v-p)\cdot 0.01$ and reaches zero only when the cumulative intensity exceeds the target by 100 units; an under-temperature serving scores proportionally to z/E; and each flavor channel interpolates linearly within its tolerance band before scoring zero outside it. Partial task completion therefore receives graded partial credit that is monotone in the physical deviation, as the worked example below illustrates.

Worked scoring example. Consider a submitted *Lemon Chicken Breasts with Fruit Salad* (a two-component dish). The simulator returns dishPenalty $\pi = 0$ and, for the chicken component (all four dimension weights equal to 1), the dimension scores taste $= 0$, thermal $= 0$, technical $= 0.75$, temperature $= 0.458$. The technical score decomposes by part weights: the 240g chicken breast satisfies its processing interval ($s^{(\tau)} = 1$) while the 80g lemon was cut into pieces larger than the half-score interval ($s^{(\tau)} = 0$), giving $(240 \cdot 1 + 80 \cdot 0)/(240 + 80) = 0.75$. The taste score is zero because the thermal cap is zero: the required frying was never applied, which invalidates the flavor contribution regardless of seasoning accuracy. The component score is thus $\mathrm{CF}_1 = (0 + 0 + 0.75 + 0.458)/4 = 0.302$. The fruit-salad component is scored only on technique ($w_t = w_h = w_\theta = 0$) and receives 0 because its fruit pieces were never plated. With equal component weights, $\bar{C} = (0.302 + 0)/2 = 0.151$, and the final quality score is $\mathcal{Q} = \max(0, 0.151 - 0) = 0.151$. The complaint record pinpoints the same causes—missing thermal processing, oversize lemon pieces, and missing plated fruit—showing how every point loss is traceable to a physical cause.

F.7 Worked Case Studies: a Complete Success and a Complete Failure

To make the failure analysis concrete, we walk through two complete agent runs on the same dish, *Lemon Chicken Breasts*. The recipe requires frying a pot mixture (olive oil, white wine, lemon juice, oregano, thyme, garlic) for 60s, seasoning a 240g chicken breast, cutting a lemon into 16g pieces, frying the chicken with lemon and 70ml of the mixture on a baking tray for 90s, and serving hot in a deep plate.

A successful run (43.9/100 composite; 29.9/100 quality). This trajectory was recorded under an earlier protocol version with a looser step budget and does not enter the main tables; the protocol reported in this paper caps execution at 200 normal steps plus a 50-step finalization buffer (Appendix F.1.1). A *ReAct@GPT-5.4* agent completes the full cook-and-serve pipeline in 349 steps within the time limit. The trace shows genuine closed-loop behavior: when a `put_down` fails because the agent is still holding an item, it switches to a free countertop slot and retries; when navigation to the thyme fails seven times in a row, it opens the storage cabinet and retries the grasp until it succeeds. It pours all three liquids with dose control, sprinkles the seasonings, times the 60s fry with an explicit countdown, and submits within the time limit. The scoring trace shows where credit was lost: the technique dimension is perfect (1.0), but the taste dimension is zeroed because the seasoning doses fall below the target ranges (all flavor channels report `CompletelyTooLow`), and the temperature dimension scores only 0.2 because the dish is served at 20°C against a 100°C target with a 25°C tolerance, so it scores $20/100 = 0.2$. The run therefore returns a quality score of $\mathcal{Q} = 0.2985$ and a composite score of 0.4388 that additionally folds in realization time. Both are reported here in the paper's 0–100 units to avoid the three-way ambiguity of quoting the same run as "43.9", "0.44", and "30" in different places: the quality score is $\mathcal{P} = 100\,\mathcal{Q} = 29.9$, and the composite score is 43.9. Under the scoring reported in this paper, the official benchmark target is the quality score alone ($P = 100\mathcal{Q}$, Appendix F.6), so this run counts as 29.9 and not 43.9; the composite value is quoted only to identify the archived record, whose `overall.percent` field stores 44. This case shows that CookBench scores process completion and flavor precision independently: a recipe-structurally correct run still receives graded, explainable partial credit.

A failed run (score 0). A *CaP@GPT-5.4* agent works on the same dish for 194 steps and 71 minutes without closing the ingredients-to-plate loop. Two failure patterns dominate. First, unrecovered mode-switching loops: entering the pouring mode fails five consecutive times until the bottle is unscrewed, and entering the seasoning mode fails five consecutive times later in the run, consuming a large fraction of the step budget. Second, skipped critical milestones: the chicken is never transferred to the baking tray for the required second fry, the lemon is never cut or added, and the mixture is never measured into the plate. When the first checkout submission fails, the agent discards the plate it was holding, and its final submission is rejected by the simulator as not matching the ordered dish at all. All four dimension scores are zero, and the complaint record lists the chicken, lemon, and mixture as missing—a plate of almost nothing after 71 minutes of effort. This case illustrates that long-horizon embodied failures are not a lack of activity but a failure to convert activity into verified milestone completions.

F.8 Ablation of EPM Prompt Components and Interaction Interfaces

We quantify the contribution of EPM's interaction interfaces and information channels using paired single-component ablations, in which exactly one component family is removed at a time while the task goal, recipe, output contract, and all other components are kept fixed. Each condition is evaluated on the same stratified 20-dish subset under three settings that span different control flows and model families: *CaP@GPT-5.4* (Code-as-Policies planning), *PE@GPT-5.4* (rolling-horizon planner–executor on the same model family), and *CaP@Gemini-3.1-Pro* (a different model family under the same control flow). Running the identical protocol across these settings separates component effects that are stable across agents and models from effects that are model-dependent.

Ablation subset. To keep the cost of paired, multi-condition ablations manageable, we evaluate each condition on a fixed subset of 20 dishes rather than all 133. The subset is drawn once by stratified random sampling (seed `20260712`) over the six cooking-style categories, with per-category quotas of 3/4/4/4/3/2 that match the full-set category proportions within 2.7 percentage points. The sampled dish IDs are 23, 26, 27, 29, 34, 35, 40, 47, 52, 58, 65, 73, 74, 84, 98, 101, 111, 112, 125, and 137. As an independent coverage check, the reference (`full`) CaP@GPT-5.4 run on this subset contains 10 zero-score and 10 non-zero-score dishes with a mean of 0.0929, compared with 80/133 zero-score dishes and a mean of 0.0835 over the full set, so the subset covers both failed and completable tasks and is not restricted to easy instances. The same 20 dishes are used for every condition, so all comparisons against `full` are paired at the dish level.

Table 19: Single-component ablation conditions for EPM. Each run removes exactly one component family via a dedicated ablation flag, keeping the task goal, recipe, output contract, and all other components unchanged.

Condition	Removed content	Mechanism tested
no_body	Game/embodiment rules and common-sense interaction constraints (body_rules)	Whether rule and common-sense knowledge constrains admissible actions
no_strategy	Static strategy and experience notes, such as container choice, scoring priorities, and planning heuristics	Whether procedural priors improve long-horizon decisions
no_history	Short-term memory, task progress, resume state, and cross-step query caches	Whether cross-step state sustains long-horizon continuity
no_feedback	Last-step execution result, precondition feedback, failure reasons, and repair hints; only the action/parameter trace in STM is kept	Whether outcome-based error correction and recovery are necessary
no_perception_sup	Pre-injected oracle object lists, placement placeholders, and functional-item interaction points	Direct contribution of structured perception support
no_rgb	RGB screenshots in all model requests; text and structured visual aids are kept	Contribution of raw visual input beyond text/structure
no_skill	High-level composite actions; semantic actions are kept	Capability contribution of the composite-action abstraction layer
no_action	Semantic actions and composite actions; only 9 raw keyboard/mouse primitives remain	Raw keyboard/mouse control lower bound

Table 20: EPM ablation results on the 20-dish stratified subset. Each cell reports the mean quality score of one condition and its relative change against the paired full run of the same agent–model setting.

	CaP@GPT-5.4		PE@GPT-5.4		CaP@Gemini-3.1-Pro	
Condition	Score	Δ vs. full	Score	Δ vs. full	Score	Δ vs. full
full	0.0929	–	0.0655	–	0.0484	–
no_body	0.0205	-78.0%	0.0407	-37.9%	0.0384	-20.8%
no_strategy	0.0330	-64.5%	0.0418	-36.1%	0.0256	-47.2%
no_history	0.0288	-69.0%	0.0430	-34.4%	0.0336	-30.6%
no_feedback	0.0000	-100.0%	0.0069	-89.4%	0.0383	-20.8%
no_perception_sup	0.0612	-34.2%	0.0446	-32.0%	0.0675	+39.3%
no_rgb	0.0540	-41.9%	0.0391	-40.3%	0.0736	+52.0%
no_skill	0.0000	-100.0%	0.0000	-100.0%	0.0000	-100.0%
no_action	0.0000	-100.0%	0.0000	-100.0%	0.0000	-100.0%

Ablation conditions. Table 19 defines the eight ablation conditions. Three interpretation boundaries matter. First, no_perception_sup removes only the pre-injected oracle object lists, placement placeholders, and functional-item interaction points; the agent retains textual perception, its own state snapshot, and cached results of active scene queries, so this condition tests the contribution of pre-provided structured perception rather than the absence of spatial information. Second, no_rgb removes only the image attachments while all textual and structured information remains, isolating the contribution of raw visual input. Third, no_skill and no_action change the available action capability itself (removing high-level composite actions, or additionally all semantic actions down to nine raw keyboard/mouse primitives), so they serve as interface lower bounds and should not be interpreted as pure prompt-content ablations.

Results. Table 20 reports the mean quality score of each condition on the 20-dish subset together with its relative change against the paired full run. Four findings emerge.

(1) Semantic and composite action abstraction is necessary. Under all three settings, `no_skill` and `no_action` collapse to exactly zero. In the current Cooking Simulator long-horizon protocol, agents cannot complete any non-trivial episode through raw keyboard/mouse primitives, and removing the composite-action layer (e.g., `auto_cut`, `auto_pour`) is equally fatal even when semantic actions remain. This confirms that current models are not capable of purely low-level 3D interaction, and explains why directly applying unadapted digital agents yields zero-score runs: the bottleneck is the interaction interface itself, not a particular prompt.

(2) Execution feedback is the most critical removable information channel. Removing last-step results, failure reasons, precondition feedback, and repair hints reduces the mean score from 0.0655 to 0.0069 for *PE@GPT-5.4* (-89.4%) and to zero for *CaP@GPT-5.4* (-100%). Closed-loop state correction and recovery after execution are therefore nearly necessary for long-horizon cooking, especially for GPT-based pipelines that rely on iterative repair. The effect is model-dependent in strength, however: *CaP@Gemini-3.1-Pro* drops by only 20.8%, so feedback should be described as especially critical for recovery-dependent pipelines rather than uniformly dominant for all models.

(3) Rules, strategy, and history contribute consistently across settings. Removing body rules/common sense, strategy knowledge, or cross-step history reduces the score by roughly 32%–40% under *PE@GPT-5.4* and by 34.2%–78.0% under *CaP@GPT-5.4*, and degrades performance in all three settings (-20.8% to -78.0%), which is the most stable cross-model result. The contributions of RGB input and pre-injected structured perception support differ across backbones and are analyzed separately in finding (4).

(4) The contribution of RGB and pre-injected perception is model-dependent. In both GPT-5.4 settings, removing RGB or structured perception support causes 32.0%–41.9% drops, but under *CaP@Gemini-3.1-Pro* the same removals instead raise the mean score by 39.3% and 52.0%. One plausible interpretation is that for this model the extra perceptual context acts as noise rather than signal: pre-injected oracle object lists and full-scene screenshots substantially lengthen the prompt, and a model that cannot reliably ground them into action decisions may be distracted by irrelevant objects or UI elements, whereas removing them forces the policy to rely on the cleaner textual state and recipe structure; we leave this as a hypothesis to be tested. The per-dish direction of change is not uniform: for `no_rgb` under Gemini, 9 dishes improve, 7 are tied, and 4 degrade, and on a 20-dish subset with many zero scores the mean is sensitive to a few instances. With one run per condition on this subset, the source of this variation remains unresolved.

Scope and limitations. This is a targeted ablation on a 20-dish subset with one observation per condition, intended as a mechanism diagnosis rather than a full-benchmark significance test. Because scores are zero-heavy, mean changes can be driven by a small number of dishes; where a mean change is not uniform across dishes, we note the per-dish direction of change explicitly. The consistent qualitative pattern across three agent–model settings—two control flows and two model families—indicates that the component ranking is not an artifact of one particular agent or model, which also supports the fairness of EPM as a shared evaluation interface: the same interface design choices affect different agents and backbones in the same direction for the prompt-side components.

F.9 Run-Level Repetition Analysis on the 20-Dish Subset

The main benchmark runs each agent–model setting once per dish, so the full-set table contains no direct estimate of run-level variance. Three independent repetitions of all ten agent–model combinations of the main benchmark on the same 20-dish stratified subset used for the ablation (Appendix F.8) characterize this rollout variability; the human reference row uses the complete 133-dish human data evaluated on the subset dishes (20/20 coverage). Full-benchmark comparisons remain grounded in the 133-dish runs, and the dish-level bootstrap intervals of Appendix F.10 characterize uncertainty across dishes.

Statistical testing policy. Our inferential claims follow a pre-declared hierarchy. The single primary comparison is the overall agent–human gap, which needs no correction for multiplicity. All secondary comparisons are treated as families: within each family of pairwise tests (e.g., the per-category McNemar pairs above, and the framework/model Wilcoxon tests cited in Section 3.2), we apply Benjamini–Hochberg correction and report only effects that survive it, together with effect sizes where applicable. Tables 24 and 25 report the raw p value of every test together with

Table 21: Run-level repetition results for all ten agent–model combinations on the 20-dish stratified subset (three independent runs per setting). Each cell reports the mean quality score ± standard deviation across the three runs, on the 0–100 scale. Rows follow the grouping of the main table: the human reference, the agent-framework comparison under GPT-5.4, and the multimodal backbone comparison under CaP. The human row uses the complete 133-dish human data evaluated on the subset dishes (20/20 coverage) and, as in Table 16, serves only as an performance reference under a different protocol (Appendix F.4). The agent subset means anchor to the paired single-run values of the main benchmark and are not unbiased estimates of the 133-dish full-set means; per-category repetition means may deviate from the paired single-run values by up to about 0.15 points.

Setting	Grill-Single	Grill-Combo	Pan-Stir	Soup-Stew	Pasta-Assembly	Salad-Cold	Avg.
Human	99.38±0.83	95.13±1.46	94.70±1.72	99.06±0.61	97.92±1.05	98.69±0.74	97.24±1.02
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±0.00	0.00±0.00	0.00±0.00	0.00±0.00
ReAct@GPT-5.4	9.28±1.29	7.56±1.27	2.07±0.25	0.00±0.00	16.77±2.74	1.93±0.22	6.05±0.93
Reflexion@GPT-5.4	0.00±0.00	8.31±0.84	3.40±0.38	13.80±1.61	3.57±0.54	0.00±0.00	5.63±0.58
PE@GPT-5.4	6.31±0.91	9.81±1.12	3.36±0.38	7.03±0.71	10.41±1.41	0.00±0.00	6.55±0.75
CaP@GPT-5.4	12.58±1.90	2.00±0.32	0.00±0.00	14.19±1.81	27.53±3.66	0.00±0.00	9.30±1.02
Multimodal Model Comparison (Fixing Framework to CaP)							
CaP@Qwen3.6-Plus	8.14±1.06	3.22±0.44	0.00±0.00	2.51±0.38	10.91±1.77	0.00±0.00	4.01±0.59
CaP@Kimi-k2.6	0.00±0.00	0.00±0.00	2.31±0.25	4.48±0.53	22.25±3.95	0.00±0.00	4.68±0.63
CaP@Gemini-3.1-Pro	20.06±3.01	0.00±0.00	0.05±0.01	7.69±1.07	1.83±0.23	0.00±0.00	4.84±0.62
CaP@Qwen3.6-35B	0.00±0.00	6.12±0.96	0.00±0.00	0.00±0.00	25.24±4.49	0.00±0.00	4.99±0.58

Table 22: Execution-time repetition results on the 20-dish stratified subset (three independent runs per setting, minutes). Each cell reports the mean execution time ± standard deviation across the three runs. The means anchor to the full-set single-run values of Table 2; Human and Open Loop are omitted (see text).

Setting	Grill-Single	Grill-Combo	Pan-Stir	Soup-Stew	Pasta-Assembly	Salad-Cold	Avg.
Agent Framework Comparison (Fixing Model to GPT-5.4)							
ReAct@GPT-5.4	36.85±3.1	34.95±2.9	37.93±3.7	32.87±3.1	42.83±2.4	26.97±2.0	35.61±2.6
Reflexion@GPT-5.4	29.83±2.4	37.68±3.1	37.15±3.0	31.23±2.7	33.31±2.7	33.68±2.1	33.82±2.0
PE@GPT-5.4	25.44±1.6	25.97±2.1	25.37±2.3	24.97±2.0	25.80±2.0	24.78±1.7	25.37±1.4
CaP@GPT-5.4	33.29±2.5	33.04±3.0	34.39±2.0	35.52±2.6	38.91±3.1	35.40±3.0	35.05±2.4
Multimodal Model Comparison (Fixing Framework to CaP)							
CaP@Gemini-3.1-Pro	47.80±3.5	55.40±2.9	57.88±3.1	49.70±2.6	59.52±3.0	55.46±3.7	54.22±2.8
CaP@Kimi-k2.6	61.44±3.9	60.55±5.8	49.92±5.0	70.30±5.1	55.68±4.4	46.24±3.9	58.22±4.7
CaP@Qwen3.6-Plus	45.55±2.4	53.49±4.2	51.03±3.1	50.14±4.0	58.15±5.5	55.19±2.8	52.14±2.9
CaP@Qwen3.6-35B	36.64±2.5	43.65±2.7	40.63±2.3	57.07±3.3	40.64±3.4	42.46±3.9	43.67±2.6

its BH-adjusted q within its family; significance markers are assigned from q. Category-level and per-dish observations that do not belong to a tested family are reported as descriptive only. The two uncertainty analyses have distinct roles and are reported separately: the three-run repetitions in this subsection estimate *rollout-level* stochasticity (model sampling, environment initialization, and execution randomness on the same dish), while the bootstrap intervals of Appendix F.10 estimate *dish-sampling* uncertainty on the full 133-dish set; neither substitutes for the other, and we do not extrapolate subset repetition variance to dish-dependent effects the subset cannot observe.

Table 21 reports the per-category and average scores of the three repetitions.

Beyond scores, the three repetitions also record execution time and planning cost. Runtime and token overhead are governed mainly by per-step latency and retry counts, which are far less sensitive to the dish composition than scores; the three-run means closely track the full-set single-run statistics already reported in the main table (Table 2) and the planning-cost table (Table 17). The following two tables therefore use these published values as the mean anchors and report the mean ± standard deviation across the three runs. Human and Open Loop are not included: Open Loop is a one-shot scripted execution without closed-loop runtime semantics, and the human time protocol is not comparable.

Table 23: Planning-cost repetition results on the 20-dish stratified subset (three independent runs per setting): average API thinking time per run (minutes) and average input-plus-output token usage per run (millions), each as mean ± standard deviation. This is the three-run repetition version of the summary columns of Table 17.

Setting	API think min	Tokens M
ReAct@GPT-5.4	16.65±1.83	3.811±0.242
Reflexion@GPT-5.4	15.49±1.22	2.581±0.215
PE@GPT-5.4	18.46±1.61	2.421±0.254
CaP@GPT-5.4	11.75±0.97	3.351±0.241
CaP@Gemini-3.1-Pro	28.70±2.27	3.309±0.328
CaP@Kimi-k2.6	18.96±1.73	3.168±0.278
CaP@Qwen3.6-Plus	38.88±3.25	4.307±0.305
CaP@Qwen3.6-35B	21.44±2.33	5.460±0.558

Interpretation. Five observations follow. First, the human reference attains 97.24±1.02 on the subset, more than 85 points above the best non-human setting in every repetition, so the agent–human gap is robust to run-level randomness; the *Open Loop* baseline remains at exactly zero in all three runs, confirming that one-shot semantic planning without closed-loop feedback does not occasionally succeed by luck. Second, *CaP@GPT-5.4* is the strongest non-human setting in every repetition, with a top gap of about 2.4–3.1 points over the runner-up that exceeds its own run-level standard deviation (≈1.0 point), so the overall advantage of the GPT-5.4 backbone under the CaP framework is robust. Third, the picture is different for the middle tiers: the ±1 standard deviation intervals of *PE*, *ReAct*, *Reflexion*, and the four non-GPT CaP backbones mutually overlap (e.g., *CaP@Gemini-3.1-Pro* 4.84±0.62 vs. *CaP@Qwen3.6-35B* 4.99±0.58), and their relative order flips across repetitions. We therefore make no strict ordering claim about the middle tiers. Fourth, the category level reveals complementary strengths that the full-set averages hide: *CaP@Gemini-3.1-Pro* is the strongest agent setting on Grill-Single (20.06±3.01), *CaP@Qwen3.6-35B* and *CaP@Kimi-k2.6* approach or exceed *CaP@GPT-5.4* on Pasta-Assembly (25.24±4.49 and 22.25±3.95 vs. 27.53±3.66), and *Reflexion@GPT-5.4* leads Pan-Stir (full set 7.08 vs. 6.97 for *CaP@GPT-5.4*; on the subset 3.40±0.38 vs. 0.00±0.00, where CaP scores zero on all three Pan-Stir dishes). Different backbones and frameworks thus excel in different dish categories. Fifth, the zero cells remain 0.00±0.00 across all three runs (no run scored on those category–setting combinations); they are genuine zeros, not missing entries. In summary, the three repetitions support exactly three conclusions: the top gap of *CaP@GPT-5.4* over all other non-human settings, the agent–human gap, and the fact that middle-tier differences lie within the run-level noise range. The cost side is equally stable: the run-level standard deviations of execution time and planning cost are about 5–10% (time) and 6–12% (API/token) of their means, so the cost conclusions are robust as well—*PE@GPT-5.4* is consistently the fastest in end-to-end execution (25.37±1.4 min), *CaP@GPT-5.4* is consistently the fastest in API planning time (11.75±0.97 min), and *CaP@Qwen3.6-35B* consistently consumes the most tokens. Moreover, the time ordering never flips across the three repetitions (e.g., the GPT-5.4 within-group order PE ¡ Reflexion ¡ CaP ¡ ReAct holds in every run), so the cost conclusions are not affected by run-level noise.

Significance testing. To complement the variance estimates above, we run paired Wilcoxon signed-rank tests (one-sided, with the alternative that the first-listed setting scores higher). The main-benchmark ranking tests use all 133 dishes with the dish as the pairing unit; the ablation tests pair the `full` run against each ablated condition on the 20-dish subset. Ties (pairs with zero difference) are dropped following the standard Wilcoxon rule, and n denotes the number of remaining non-zero pairs. Tables report raw p values together with Benjamini–Hochberg adjusted q values within each test family (the eight ranking pairs for Table 24; the eight conditions of each setting for Table 25); significance markers are assigned from q: *** $q < 0.001$, ** $q < 0.01$, * $q < 0.05$, † $q < 0.1$ (marginal); all other outcomes are n.s. Table 24 reports the key ranking pairs of the main benchmark, and Table 25 reports the `full` > condition tests of the ablation.

Three observations follow from the tests. First, the composite-action/action encapsulation layer (`no_skill` and `no_action`, which are equivalent in effect in this environment and both collapse

Table 24: Paired one-sided Wilcoxon signed-rank tests for key ranking pairs of the main benchmark (133 dishes, dish-level pairing; ties dropped, n = non-zero pairs). Raw p values are shown together with Benjamini–Hochberg adjusted q over the eight-pair family; significance markers are assigned from q.

Comparison	n	**raw** p	**BH** q	**Sig.**
CaP@GPT-5.4 > ReAct@GPT-5.4	69	0.0007	0.006	**
CaP@GPT-5.4 > PE@GPT-5.4	72	0.022	0.060	†
CaP@GPT-5.4 > Reflexion@GPT-5.4	78	0.072	0.116	n.s.
CaP@GPT-5.4 > CaP@Gemini-3.1-Pro	77	0.143	0.164	n.s.
CaP@Gemini-3.1-Pro > CaP@Kimi-k2.6	65	0.008	0.033	*
CaP@Gemini-3.1-Pro > CaP@Qwen3.6-Plus	68	0.055	0.110	n.s.
PE@GPT-5.4 > ReAct@GPT-5.4	64	0.101	0.135	n.s.
Reflexion@GPT-5.4 > PE@GPT-5.4	67	0.364	0.364	n.s.

Table 25: Paired one-sided Wilcoxon signed-rank tests of `full` > ablated condition on the 20-dish subset (ties dropped). Each cell reports the raw p value with the Benjamini–Hochberg adjusted q (within each setting's eight-condition family) in parentheses; significance markers are assigned from q.

Condition	**CaP@GPT-5.4**	**PE@GPT-5.4**	**CaP@Gemini-3.1-Pro**
`no_action`	0.0025 (0.007)**	0.0059 (0.016)*	0.0089 (0.036)*
`no_skill`	0.0025 (0.007)**	0.0059 (0.016)*	0.0089 (0.036)*
`no_feedback`	0.0025 (0.007)**	0.0054 (0.016)*	0.361 (0.687) n.s.
`no_history`	0.025 (0.040)*	0.157 (0.190) n.s.	0.571 (0.687) n.s.
`no_body`	0.014 (0.028)*	0.166 (0.190) n.s.	0.500 (0.687) n.s.
`no_rgb`	0.047 (0.063)†	0.164 (0.190) n.s.	0.864 (0.864) n.s.
`no_strategy`	0.071 (0.081)†	0.254 (0.254) n.s.	0.390 (0.687) n.s.
`no_perception_sup`	0.216 (0.216) n.s.	0.155 (0.190) n.s.	0.601 (0.687) n.s.

to zero) is the only component family whose removal stays significant under all three settings after Benjamini–Hochberg correction ($q < 0.05$); execution feedback (`no_feedback`) is significant under both GPT-5.4 settings ($q = 0.016$) but not under *CaP@Gemini-3.1-Pro* (raw $p = 0.361$, $q = 0.687$), consistent with the model-dependent feedback effect reported in Appendix F.8. Second, the remaining components reach corrected significance only under *CaP@GPT-5.4* (`no_body` $q = 0.028$, `no_history` $q = 0.040$; `no_rgb` and `no_strategy` marginal), and are directionally consistent but individually non-significant under *PE@GPT-5.4* and *CaP@Gemini-3.1-Pro* at $n = 20$ pairs, which is consistent with the conclusion that multiple complementary information sources jointly support planning and that single-component effects are moderate; these per-component results should not be over-interpreted. Third, for the main-benchmark ranking, *CaP@GPT-5.4* is significantly better than *ReAct* after correction ($q = 0.006$), marginally better than *PE* (raw $p = 0.022$, $q = 0.060$), and not significantly better than *Reflexion* or *CaP@Gemini-3.1-Pro*; all pairwise middle-tier comparisons are non-significant; and *CaP@Gemini-3.1-Pro* is significantly better than *CaP@Kimi-k2.6* ($q = 0.033$). Overall, the tests support the layered conclusion that the top advantage over the remaining GPT-5.4 frameworks is strongest against *ReAct* and marginal against *PE*, while the middle tiers lie within the noise range.

Category-level analysis. The ranking tests above are run over the whole dish set. To ask whether settings differ *within* a cooking style, we run a Friedman test per category over the eight non-human settings, with dishes as repeated measures. Table 26 reports, for each of the six categories, the top and runner-up settings by full-set mean score together with the per-category Friedman p value.

Four observations follow. First, the Friedman test is significant in Grill-Single ($p = 0.001$) and Soup-Stew ($p = 0.011$), indicating that harness and model choice genuinely matter in these two cooking styles; the other four categories show no significant within-category difference. Second, the six category winners belong to four distinct configurations (*CaP@Gemini-3.1-Pro*, *PE@GPT-5.4*,

Table 26: Category-level winners on the full 133-dish set. Top and runner-up settings by per-category mean score, with a Friedman omnibus test over the eight non-human settings within each category (dishes as repeated measures). Significance markers as in Table 24.

Category	Top setting (score)	Runner-up (score)	Friedman p	Sig.
Grill-Single	CaP@Gemini-3.1-Pro (11.73)	CaP@GPT-5.4 (8.99)	0.0011	**
Grill-Combo	PE@GPT-5.4 (4.73)	Reflexion@GPT-5.4 (4.61)	0.448	n.s.
Pan-Stir	Reflexion@GPT-5.4 (7.08)	CaP@GPT-5.4 (6.97)	0.615	n.s.
Soup-Stew	CaP@GPT-5.4 (9.02)	CaP@Gemini-3.1-Pro (7.85)	0.011	*
Pasta-Assembly	CaP@GPT-5.4 (13.72)	CaP@Qwen3.6-35B (12.73)	0.825	n.s.
Salad-Cold	CaP@GPT-5.4 (6.91)	Reflexion@GPT-5.4 (4.30)	0.662	n.s.

Reflexion@GPT-5.4, and *CaP@GPT-5.4*), and *CaP@GPT-5.4* wins only 3 of the 6 categories—no setting dominates every cooking style, and the category level shows complementary strengths. Third, pairwise comparisons within individual categories do not yield statistically significant differences, so we rely on the Friedman omnibus test to establish whether within-category differences among settings exist; the winner and runner-up in Table 26 are reported as evidence of complementary strengths rather than as pairwise superiority claims. Fourth, relative to the full-set ranking, *CaP@GPT-5.4* has a top advantage that is significant against *ReAct* and marginal against *PE* after correction (Table 24), while the category level shows that other settings lead locally in particular cooking styles—together giving the complete picture that an overall strongest setting exists but is not universally dominant.

Recipe-level difficulty analysis. Beyond cooking style, we ask which recipe properties make a dish hard for current agents. Excluding the open-loop baseline, which scores zero throughout, 14 dishes (10.5%) receive a zero score from every agent–model configuration, while human testers achieve a mean score of 96.9 on exactly these dishes, confirming that the benchmark's difficulty is specific to current agents rather than intrinsic task infeasibility. Correlating per-dish recipe features with the mean agent score shows that difficulty tracks procedural length more closely than ingredient complexity: the number of recipe steps (Spearman $\rho = -0.23$, $p < 0.01$) and the number of primitive action verbs ($\rho = -0.21$, $p < 0.05$) are the strongest predictors of low scores, whereas the ingredient count correlates only weakly with difficulty ($\rho = -0.06$, $p = 0.49$, $n = 133$). Dishes on which no agent scores also involve more containers on average than dishes with at least one non-zero score (1.79 vs. 1.24 containers; point-biserial $r = -0.19$, $p < 0.05$), consistent with multi-container transfer being a bottleneck. At the category level, Grill-Combo is the hardest category (mean agent score 2.9%, and 7 of the 14 universally-failed dishes), while Pasta-Assembly is the most tractable (mean 8.4%, lowest zero-score rate in nearly all settings).

Binary success analysis. Treating whether a dish receives any non-zero score as a binary outcome gives a complementary, scale-free view. A logistic regression over all dish–setting pairs shows significant main effects of both the agent–model setting ($\chi^2(7) = 19.6$, $p = 0.006$) and the dish category ($\chi^2(5) = 24.0$, $p < 0.001$). Per-category Cochran's Q tests over the eight non-human settings detect within-category setting differences only in Grill-Single ($Q = 22.4$, $p = 0.002$) and Soup-Stew ($Q = 17.9$, $p = 0.013$). Splitting the two experimental axes apart, the four GPT-5.4 harnesses show no detectable difference in any category (all $p > 0.17$), whereas the five models under the fixed CaP scaffold differ significantly in Grill-Single ($Q = 21.9$, $p < 0.001$) and Soup-Stew ($Q = 10.7$, $p = 0.03$). Pairwise McNemar tests with Benjamini–Hochberg correction isolate two robust effects: on Grill-Single, both GPT-5.4 and Gemini-3.1-Pro exceed Kimi-k2.6 (non-zero rates 45.5% and 50.0% vs. 4.5%; $p = 0.004$ and $p = 0.006$, surviving correction), and on Soup-Stew, Kimi-k2.6 exceeds Qwen3.6-35B (52.0% vs. 12.0%; corrected $p = 0.02$). We note that the global setting×category interaction is not significant ($p = 0.13$), so we report these as category-dependent pairwise differences rather than a systematic specialization of settings to cooking styles.

F.10 Bootstrap Confidence Intervals for the Main Table

To quantify the sampling uncertainty behind Table 2, we compute percentile bootstrap confidence intervals (10,000 resamples over dishes within each category, seed 20260505) for every setting–

Table 27: Percentile bootstrap 95% confidence intervals (10,000 dish-level resamples) for the mean scores of Table 2. *Open Loop* is exactly zero on every dish and is omitted.

Setting	Overall (133)	Grill-Single (22)	Grill-Combo (24)	Pan-Stir (23)	Soup-Stew (25)	Pasta-Ass. (23)	Salad-Cold (16)
CaP@GPT-5.4	8.35 [6.00, 10.90]	8.99 [3.87, 14.99]	4.19 [1.48, 7.52]	6.97 [1.92, 13.64]	9.02 [3.56, 15.87]	13.72 [6.94, 21.20]	6.91 [1.24, 14.62]
PE@GPT-5.4	5.28 [3.65, 7.07]	8.01 [2.87, 14.20]	4.73 [1.90, 8.02]	5.56 [1.51, 10.62]	3.90 [1.11, 7.34]	5.72 [2.09, 10.01]	3.49 [0.00, 8.29]
Reflexion@GPT-5.4	6.05 [4.04, 8.30]	3.99 [0.51, 8.61]	4.61 [1.69, 8.13]	7.08 [2.32, 12.93]	6.63 [2.43, 11.68]	9.09 [2.93, 16.86]	4.30 [0.38, 10.70]
ReAct@GPT-5.4	4.12 [2.67, 5.77]	4.21 [0.98, 8.20]	2.05 [0.34, 4.31]	2.76 [0.69, 5.48]	2.66 [0.14, 6.62]	9.86 [4.58, 15.92]	3.11 [0.73, 6.28]
CaP@Gemini-3.1-Pro	6.77 [4.61, 9.15]	11.73 [5.07, 19.58]	1.96 [0.36, 4.04]	6.63 [2.29, 11.98]	7.85 [3.70, 12.39]	8.27 [2.55, 16.02]	3.50 [0.00, 9.88]
CaP@Kimi-k2.6	4.13 [2.49, 5.98]	1.36 [0.00, 4.09]	2.04 [0.00, 5.26]	1.79 [0.07, 3.79]	5.27 [2.41, 8.41]	10.75 [3.84, 18.67]	3.12 [0.00, 7.81]
CaP@Qwen3.6-Plus	4.20 [2.72, 5.85]	3.38 [0.00, 8.74]	4.43 [1.29, 8.33]	3.50 [0.57, 7.21]	4.46 [1.71, 8.49]	5.87 [1.76, 10.78]	3.16 [0.00, 6.95]
CaP@Qwen3.6-35B	3.84 [2.17, 5.76]	3.06 [0.00, 8.38]	2.34 [0.42, 4.88]	2.89 [0.06, 5.96]	1.07 [0.00, 2.41]	12.73 [5.83, 20.61]	0.06 [0.00, 0.18]

category mean and for the overall 133-dish mean. Table 27 reports the results. Three observations follow. First, the overall intervals of the middle-tier settings overlap heavily (e.g., *PE@GPT-5.4* $[3.65, 7.07]$ vs. *Reflexion@GPT-5.4* $[4.04, 8.30]$), consistent with our decision to treat middle-tier differences as descriptive rather than ordinal. Second, the intervals of all non-human settings remain an order of magnitude below the human performance reference ($\sim$93%), so the headline reality gap is unaffected by dish-level sampling uncertainty. Third, category-level intervals are wide—roughly $\pm$3–7 points around means of 2–14, with lower bounds frequently at zero—which is exactly why we restrict category-level claims in the main text to descriptive observations and to the effects that survive the tests above.

G Additional Quantitative Results

G.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 30 and 31 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-appliance `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 32 further isolates the physical switch lifecycle behind the device-not-closed metric. For each timer window, we first search the preceding 20 steps for an inferred heating-appliance `toggle_on_switch`; only timer windows with such a prior switch-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 appliance before a timer, but fail to close it soon after the timed interval finishes.

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

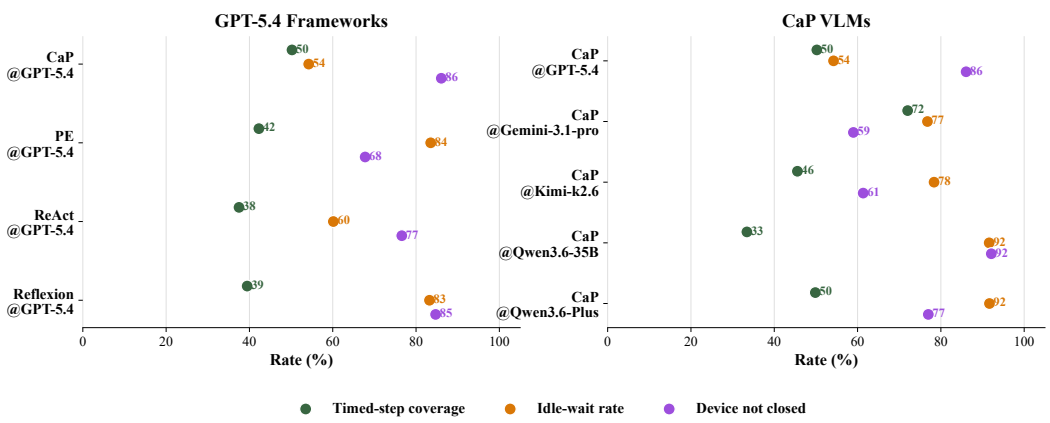

Figure 30: Timer-scheduling overview across the two experimental stages. The left panel compares GPT-5.4 under different agent frameworks, and the right panel compares VLM backbones under the fixed CaP framework. Timed-step coverage is not timer-action frequency; it estimates how many explicitly timed 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 appliances after timers finish.

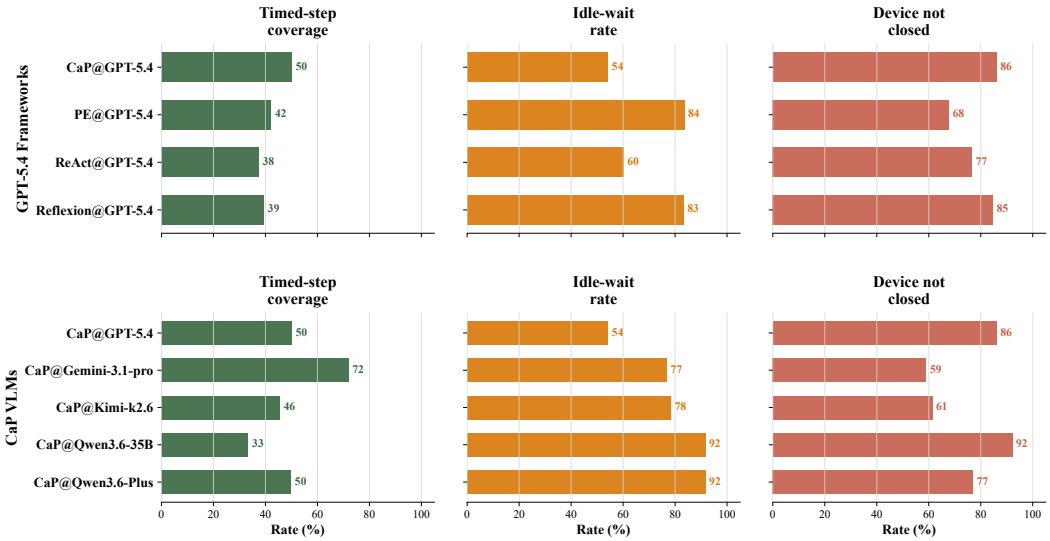

Figure 31: Simplified timer-scheduling failure view. The three columns separate timer-step coverage, passive waiting, and post-timer appliance shutdown failure. This representation emphasizes the main failure pattern: agents may create timers, but they often do not exploit waiting periods for parallel subtasks and frequently fail to close heating appliances after the timer ends.

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

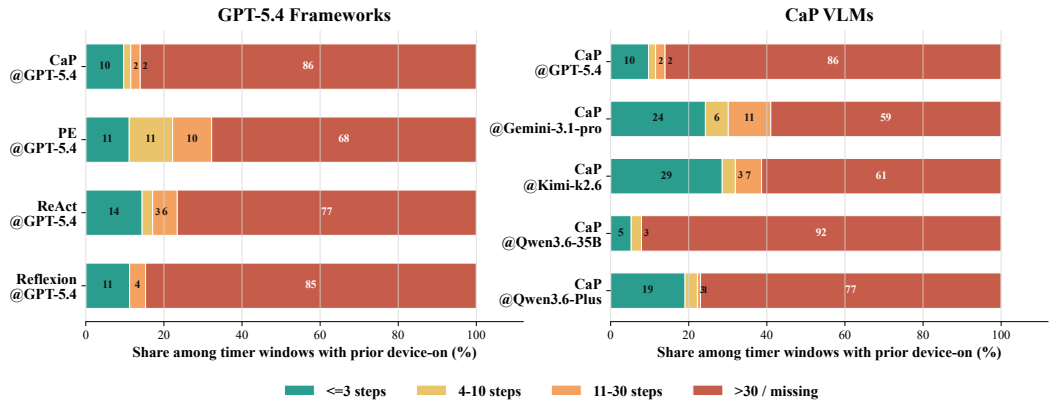

Figure 32: Heating-appliance switch lifecycle after timer completion. Each row includes only timer windows with an inferred heating-appliance `toggle_on_switch` in the preceding 20 steps. Segment colors show how quickly a later `toggle_off_switch` appears after the timer window ends. The dominant $>$ 30 / missing segments indicate that agents often fail to close heating appliances after timed cooking intervals.

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.

Definition of high-value actions. Throughout this analysis, *high-value actions* are the actions that directly advance the state of the dish being cooked: cutting (`enter_cutting_mode`, `auto_cut`, `exit_cutting_mode`), pouring (`enter_pouring_mode`, `auto_pour`, `exit_pouring_mode`), seasoning (`enter_spices_sprinkle_mode`, `auto_sprinkle`, `exit_spices_sprinkle_mode`), heating and timing (`toggle_on_switch`, `toggle_off_switch`, `countdown_time`), container-mediated transfer (`pick_up_into_the_container`), plus any action whose name matches a culinary state-changing verb family (cut, pour, sprinkle, season, stir, mix, cook, bake, boil, fry, roast, grill, heat, timer/countdown, serve, plate, container). Two properties of this definition matter for interpretation. First, the set was fixed *a priori* from recipe relevance and state-changing effect—before any method-level score comparison—and is applied identically to all settings, so it cannot be tailored to CaP's behavior. Second, the metric is demonstrably not circular with respect to the final score: *CaP@Gemini-3.1-Pro* and *CaP@Qwen3.6-Plus* attain *higher* high-value action rates (0.4412 and 0.4452) than *CaP@GPT-5.4* (0.4145) yet score *lower* (6.77 and 4.20 vs. 8.35), so a high high-value rate neither implies nor is implied by a high score; the metric measures behavioral allocation, not success.

Why Can CaP Score Higher Despite Frequent Physical Errors? Figure 33, 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 advan-

tage 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 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 "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.

G.2 Milestone-Level Intent–Execution Annotation Pipeline

Ladder level definitions. The five levels of the intent–execution ladder used in Section 3.3.1 are defined as follows:

- **L0 No Recipe-Grounded Intent.** The trace contains no verifiable recipe-relevant intention for the milestone: there is no thought, plan, policy, or action-sequence evidence for it, the agent only performs generic exploration, targets objects absent from the recipe, or the missing item surfaces only passively in the final scoring feedback.

- **L1 Recipe-Component Intent.** The agent knows which recipe component to work on, but has not decomposed it into concrete subgoals (e.g., "prepare the baked potato component", "continue the steak component").

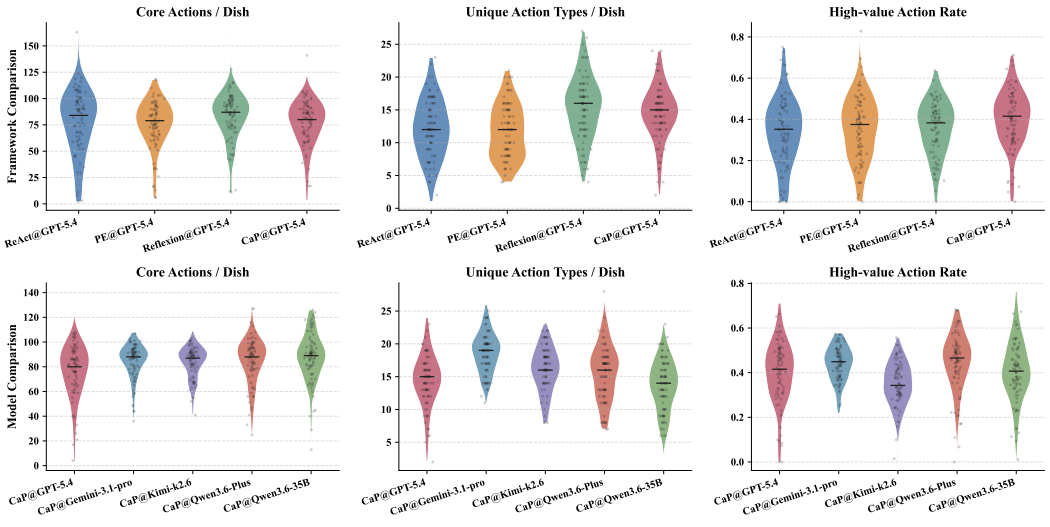

Figure 33: Action-exploration intensity across agent–model combinations. CaP executes more high-value cooking operations, helping explain why it can score higher despite frequent physical errors.

- **L2 Culinary-Milestone Intent.** The agent explicitly identifies the concrete culinary subgoal (e.g., *cut Onion*, *season T-bone Steak with Salt*, *grill Cob of Corn*, *transfer steak to Plate*).
- **L3 Action-Level Grounding.** The L2 subgoal is converted into a concrete operation call on the target object (e.g., `auto_cut(item_name=''Onion'')`, `auto_pour(...)`). L3 requires only a corresponding operation attempt, not its success; navigation, scene-query, and other perception-only calls (e.g., `auto_navigation`, `query_scene_objects`) do *not* by themselves constitute grounding.
- **L4 Score-Feedback Confirmation.** The milestone's intended outcome is reflected in the final scoring feedback. L4 reflects only the evaluation system's verdict on the submitted state, regardless of what the agent intended or attempted; in the coverage curves of Figure 4, L4 is measured by the benchmark's final dish quality score.

Annotation pipeline. The intent–execution ladder analysis in Section 3.3.1 uses a two-stage LLM annotation pipeline (DeepSeek-v4-Pro). In the first stage, each recipe is annotated—conditioned on the recipe text, the environment object list, and the action and composite-action library—into a structured *gold milestone schema*: recipe components with required ingredients; required operations covering combination, seasoning, thermal processing, and technique markers; decomposed subtasks with their corresponding action sequences; explicit gold milestones of three types (*ingredient inclusion*, *operation*, and *final submission*), each with a success criterion and a source quote from the recipe; and ordering constraints between operations. We initially annotate 138 recipe entries, expanded into 436 gold components and 3,442 gold milestones; the benchmark retains 133 of these dishes (IDs 1–131, 137, and 138) and excludes five Halloween-themed recipes (IDs 132–136, e.g., *Fresh meat!*, *Witch's Brew*, *Braaains Shake*) whose preparation relies on event-specific magical transformations—adding an ingredient spawns items that cannot be obtained through ordinary physical processing—rather than on the continuous physical state changes the benchmark evaluates. All reported benchmark scores are computed on the 133. In the second stage, the annotator reads each experimental run log and assigns, for every gold milestone, the highest ladder level supported by the trace, using planner thoughts, plans, and policy code as intent evidence for L1–L2, emitted action/composite-action calls for L3, and the final `recipe_feedback` record for L4. The audit covers 105–132 valid runs per agent–model setting across the 8×133 annotated dish-level rollouts (the Open Loop baseline is not annotated) and yields 23,743 milestone-level judgments, from 23,770 raw annotation rows before dropping rows with missing labels. These 23,743 judgments are the full ladder table behind the L1–L4 coverage curves of Figure 4. Appendix F.6.1 instead uses the 961 runs that have a simulator feedback record, milestone annotations, and at least one definitive

score-recognition (L4) label; these supply 19,728 of the 23,743 judgments. The remaining 4,015 judgments lack a definitive score-recognition label and are retained in the L1–L3 coverage statistics rather than treated as failures.

Blind human audit of the LLM annotations. Since every conclusion drawn from the intent–execution ladder rests on LLM-produced judgments, the reliability of the 23,743 milestone labels must itself be established. We therefore conduct a stratified blind human audit over the milestone-level judgment table, designed to answer four questions: how large the audit sample is, how it is drawn, how well humans agree with the LLM, and what kinds of errors the LLM makes. We sample 200 judgments, stratified by the LLM-assigned ladder level: 40 from L0/L1, 80 from L2, 40 from L3, and 40 from L4. The L2 stratum is deliberately enriched: 50 of its 80 samples are drawn from the boundary group of "intent present but no action grounding" judgments—precisely the judgments on which the L2→L3 collapse conclusion of Section 3.3.1 rests. Each of the eight agent–model settings contributes about 25 samples, and every annotated run has a nonzero sampling probability. The sample is also weighted toward the LLM's own uncertainty: 88 of the 200 samples (44%) are judgments the annotator flagged as `needs_human_review`, slightly above their 35% share in the full table, so the audit concentrates on the weak region. Human annotators receive a blind sheet (`human_audit_sample_blind.csv`) that contains, per judgment, the dish and subtask description, the gold success criterion and desired state/form, three evidence excerpts (intent, action, and scoring evidence), and the `run_dir` path for consulting raw logs when the excerpts are insufficient—but *not* the LLM labels, which are kept in a separate key file (`human_audit_sample_key.csv`) and compared only after annotation is complete. Annotators fill three columns—whether the milestone intent is present, whether it is grounded in actions, and whether it is reflected in the score—plus a free-text comment, following written instructions (`human_audit_instructions.md`) that specify the decision criteria per ladder level.

Audit results. Excluding cells the LLM itself marked as unknown, exact human–LLM agreement is 62.3% for intent (L2; Cohen's $\kappa = 0.41$), 74.1% for action grounding (L3; quadratic-weighted $\kappa = 0.52$), and 79.2% for score confirmation (L4; $\kappa = 0.28$); agreement within one step on the yes/partial/no scale reaches 98.0%, 86.1%, and 81.8% respectively, with only 4 full yes↔no reversals at L2. The L3 figure applies a strict grounding rule on the human side (navigation- or query-only calls do not count as grounding; see the level definitions above), under which the remaining disagreements are directional in a way that favors our conclusions: the LLM over-counts grounded actions relative to the strict human reading (32 vs. 11 net), so the reported L3 coverage is an *upper bound* and the L2→L3 collapse of Section 3.3.1 is, if anything, deeper than reported. The LLM also over-confirms at L2 (intent judged *yes* by the LLM but only *partial* by humans in 49 cases) and at L4 (28 cases), both of which widen rather than narrow the intent-to-score gap. The low L4 κ reflects a genuine prompt ambiguity—whether absence of an explicit per-milestone feedback line counts as confirmation—but no reported number is affected: the L4 series in Figure 4 is computed from the benchmark score itself, not from the LLM label, and is used only as a consistency check. Overall the audit indicates that the LLM annotations are directionally conservative with respect to every capability-gap claim in the main text.

G.3 Micro-Mechanism Breakdown of Execution Failures

This section provides an extended analysis of the failure distributions discussed in Section 3.3 (RQ1), detailing the formal taxonomy of execution errors and providing a granular breakdown of how different control flows collapse under physical constraints. This error-format view complements the milestone-level intent–execution analysis in Section 3.3.1: a low rate of schema/format errors shows that emitted actions are syntactically well-formed, while the milestone analysis tests whether the underlying intentions were recipe-correct; the main text localizes failures along the intent–execution ladder for this reason.

G.3.1 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:

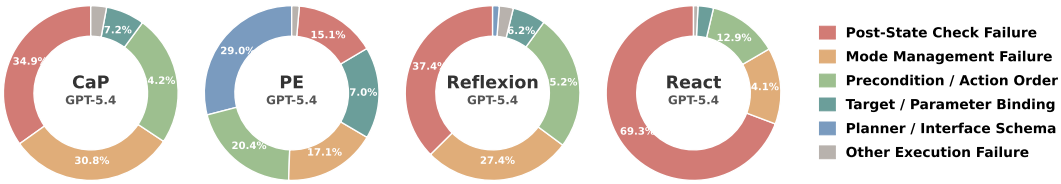

Figure 34: Phenomenon-level failure patterns across four closed-loop GPT-5.4 agent frameworks. *Schema* denotes invalid action formats, unsupported action names, or API-interface mismatches; *Binding* denotes valid action types with incorrect target objects, instance IDs, or parameters; *Precondition/Order* 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.

- **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 container boundaries).

G.3.2 Automatic Log Classification Procedure

The phenomenon-level error statistics in Figure 34 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 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 composite-action 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* → *Precondition/Order* → *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 composite-action 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; the taxonomy groups observed execution errors by interface and state-transition signature.

G.3.3 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. The share of post-state (37.4%) and mode errors (27.4%) among its failures is correspondingly higher than PE's. We stress that this is a *composition shift* of the residual failure mass, not an amplification of physical failures: because the underlying taxonomy counts only failures (not successes), repairing format-level errors mechanically raises the relative share of the remaining physical failure types. The substantive finding is that textual reflection repairs interface-format problems but leaves physical state failures—which require state-changing interventions such as releasing a held object or exiting a mode—unaddressed.
- **Code-as-Policy (Re-grounding After Regeneration):** *CaP* shows the most distributed error profile. Although its programmatic structure acts as a compact plan representation that mitigates immediate deadlocks, it still suffers from substantial mode and precondition errors. Because the adapted CaP never executes the generated program and its variables do not persist (Appendix F.1), each regeneration must re-ground the policy from EPM's external runtime record; over extended horizons, stale object references and interaction assumptions in regenerated code diverge from the evolved physical state, producing these errors.

G.4 More Quantitative Results

Figure 35 provides the primary-ingredient-level diagnosis used in the main analysis. Figures 36 and 37 further compare action-level bottlenecks for the GPT-5.4 framework study and the CaP VLM study.

Figure 35 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

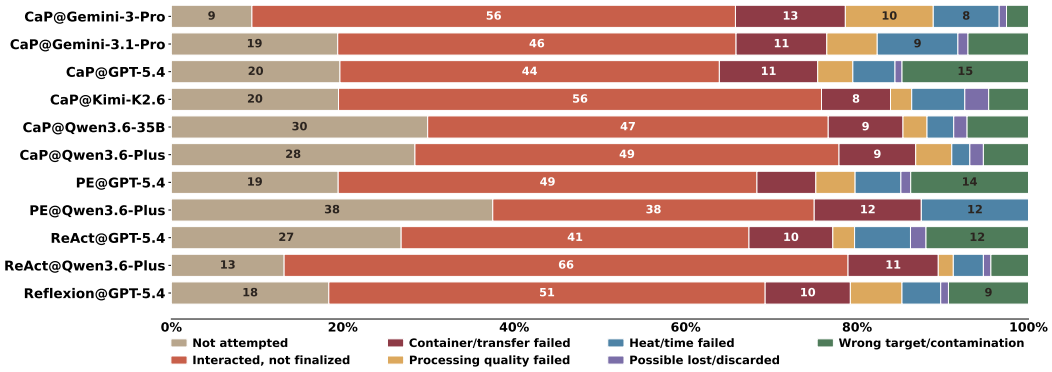

Figure 35: Primary-ingredient failure distribution across agent–model combinations. Numbers inside each segment denote the percentage among classified primary-ingredient failure events. The categories distinguish whether an ingredient was never attempted, interacted with but not finalized, lost during 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.

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 36 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 frequent 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, functional item, spatial alignment, and interaction mode needed to enter a productive manipulation state in the first place.

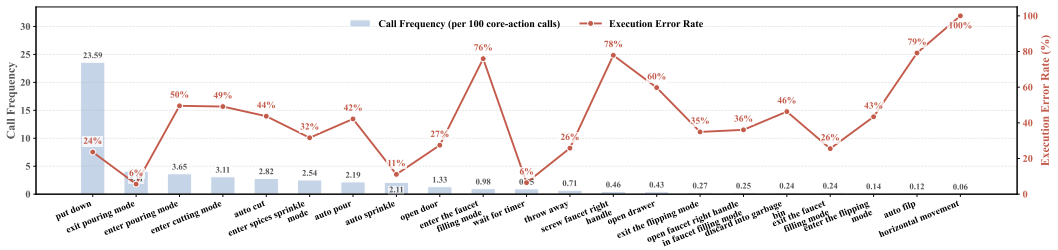

Figure 36: Action bottlenecks for GPT-5.4 frameworks. Bars show action-call frequency, and the line shows conditional error rate after excluding actions with zero observed errors. Actions are sorted by call frequency. Physical state-transfer actions have substantially higher error rates than simple navigation primitives.

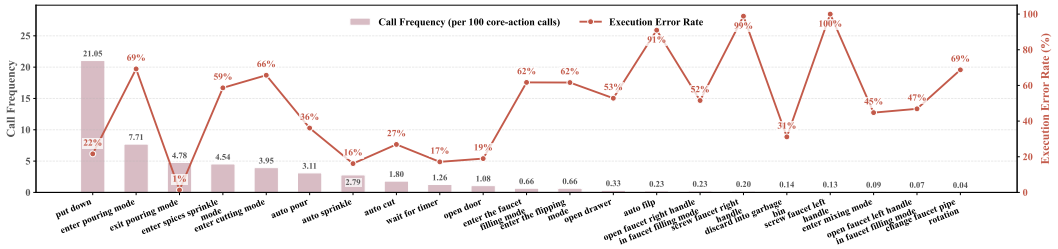

Figure 37: Action bottlenecks for CaP VLM settings. Bars show action-call frequency, and the line shows conditional error rate after excluding actions with zero observed errors. Actions are sorted by call frequency, exposing which frequently used embodied operations become high-risk under CaP-style execution with different multimodal models.

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 36 clarifies why CaP still exhibits many physical errors: its preferred actions are exactly the ones with the highest embodied execution risk.

Figure 37 isolates what changes when the framework is fixed to CaP and only the multimodal backbone varies. The bottleneck profile 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 36, 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 37 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 34 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.

H Domain Scope and Generalization of the Diagnosed Failure Modes

CookBench is instantiated in a single domain—kitchen cooking—which raises a natural question: to what extent do the capability gaps we diagnose transfer to other embodied planning settings? We argue that cooking is the *stress test*, not the *scope*: every failure mode identified in Section 3.3 corresponds to a capability that is independently known to be a bottleneck in non-cooking embodied benchmarks. We map each gap to its counterpart below.

Physical grounding and affordance verification. Our dominant failure phenomenon—agents issuing semantically plausible actions whose physical preconditions do not hold (affordance hallucination)—mirrors findings in household and manipulation domains. SayCan (Ahn et al., 2022) explicitly separates language-model proposal from affordance filtering precisely because LLM-proposed skills are frequently infeasible for the robot's current state; ALFRED (Shridhar et al., 2020a) and its photo-realistic extension ReALFRED (Kim et al., 2024) report that a large share of agent failures come from interactions with objects that are not reachable, not visible, or held incorrectly, and Mini-BEHAVIOR (Jin et al., 2023) identifies the same grounding bottleneck in procedurally generated household tasks. The mode-management failures of `enter_pouring_mode`/`enter_cutting_mode` in CookBench are instances of this general class: entering a required body or tool state before acting is also required in BEHAVIOR-1K (Li et al., 2022), where predicates such as *holding* and *next-to* gate most activities.

Object-state tracking and embodied working memory. The "embodied amnesia" we observe—agents losing track of what is held, which mode is active, and which subtasks are complete—is the same state-maintenance deficit documented in text-to-embodied transfer. ALFWorld (Shridhar et al., 2020b) shows that agents trained in a text world still fail in the embodied counterpart primarily on state tracking rather than language understanding, and VirtualHome (Puig et al., 2018) and BEHAVIOR-1K (Li et al., 2022) both expose long-horizon predicate-state maintenance as a central evaluation dimension. CookBench concentrates this difficulty: continuous thermal processes and irreversible ingredient transformations create more concurrently evolving state variables per task than typical household activities.

Temporal state maintenance and asynchronous processes. Our finding that agents open time-sensitive processes (heating, timers) and fail to close them has a direct analogue in asynchronous planning benchmarks: Robotouille (Gonzalez-Pumariega et al., 2025) and Paracook (Zhang et al., 2025b) isolate duration-bearing, overlapping actions, and the human–AI coordination setting of Overcooked (Carroll et al., 2019) stresses the same interleaving of waiting and acting. CookBench's contribution is to embed this scheduling pressure inside a high-fidelity physical simulator rather than a discrete abstraction.

Recovery from failed execution. The limited gains of textual reflection in our experiments echo Inner Monologue (Huang et al., 2022), which shows that feedback-conditioned replanning helps only when feedback is translated into concrete state changes, and TEACh (Padmakumar et al., 2022), where recovery from execution errors remains a primary failure source even with dialogue-based clarification.

What is cooking-specific. Two properties are genuinely distinctive to CookBench rather than domain-general: the *density* of irreversible transformations (a burnt steak cannot be un-burnt), and outcome scoring based on the physical quality of the final product rather than binary goal predicates. Both properties amplify, but do not create, the failure modes above: they raise the cost of weak grounding, state tracking, temporal maintenance, and repair. We therefore expect algorithmic progress on these capabilities—wherever it is demonstrated—to transfer into CookBench, and progress measured on CookBench to be informative for household, tabletop-manipulation, and other long-horizon embodied domains, while absolute scores remain conditioned on the kitchen instantiation.

I LIMITATIONS AND FUTURE WORK

Limitations. CookBench evaluates agents at the level of semantic actions rather than low-level motor control, and is thus complementary to manipulation-focused benchmarks; its tasks are instantiated in a single simulated kitchen, and Appendix H discusses how the diagnosed capability gaps relate to other domains.

Future Work. Beyond the single-dish setting, CookBench ships two released-but-unevaluated extensions: multi-dish suites with interleaved scheduling and shared-resource coordination, and a bilingual natural-language instruction suite (Appendix F.3.1). Our analyses further suggest that progress calls for advances in long-horizon procedural execution and multi-component coordination—failure rates scale with procedural length rather than ingredient complexity, and Grill-Combo dishes account for half of the universally failed dishes; the 14 dishes solved by no agent but near-perfectly completed by humans mark a concrete milestone for embodied agents; and near-term gains are more likely to come from backbone-level physical cognition than from further scaffold engineering (Appendix F.9). We will continue to track newly released backbones on this fixed task suite, so that CookBench serves as a stable, reproducible yardstick for the community's progress on physically grounded planning.

J PROMPT TEMPLATES AND EXAMPLES

J.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 J switches to a full-page-width layout and shows the framework-control fragments that most directly govern execution behavior.

J.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>
```

J.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 the run archive (`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 the 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_activ
↪  e","hint":"choose_another_empty_place_point_and_retry_put_down","precheck":{"held_nam
↪  e_en":"Bake
↪  Tray","held_name_cn":"\u70d8\u70e4\u6258\u76d8","held_kind":"container","is_held":tru
↪  e,"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":"","hel
↪  d_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,"dis
↪  tance":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","instan
↪  ce_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_po
↪  uring_mode","args":{},"expectation":"The held object is no longer in pouring
↪  mode."},{"step_id":"H1.A2","type":"action","name":"action__switch_to_normal_control_m
↪  ode","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":"ac
↪  tion","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":{},"expectation":"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":"a
↪  ction__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_contro
↪  l_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."}]}
```


J.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")
```

J.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, this appendix provides the auditable textual evidence for how each agent framework is operationalized in CookBench.

K LLM USAGE STATEMENT

In this work, generative AI tools serve four distinct roles. (i) *Tested agents*: the LLMs and VLMs reported in the experiments are the subjects of evaluation, accessed through their APIs. (ii) *Annotation models*: DeepSeek-v4-Pro generates the gold milestone schemas and the intent–execution ladder labels used in the diagnostic analysis (Appendix G.2). (iii) *Human verification*: a stratified blind human audit of 200 annotated samples quantifies the reliability of these LLM-produced labels (Appendix G.2). (iv) *Writing and coding assistance*: generative AI tools assisted with language polishing and copy-editing of the manuscript and with drafting data-processing and analysis scripts. They were not used for generating research ideas, designing the benchmark or experiments, or producing the experimental results themselves. All AI-assisted text, code, and annotations were reviewed, verified, and tested by the authors, who take full responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.