synthDCB: WORLD DISTILLATION FOR SCALABLE DEVICE CONTROL BENCHMARKING

Anonymous authors
Paper under double-blind review

ABSTRACT

Scaling device-control evaluation requires scaling both the worlds agents interact with and the tasks they must solve. We introduce **SYNTHDCB**, a scalable benchmark for environment awareness and device control built through **world distillation**. World distillation transforms knowledge of device behaviors, interaction constraints, and environmental effects into modular executable components, which are composed into interactive worlds with explicit states, actions, and temporal dynamics. A unified capability representation connects tool discovery, world execution, and task synthesis, allowing new capabilities to enter evaluation through a shared construction pipeline. SYNTHDCB instantiate Home, Car, and Phone domains with 114 device classes, 1,166 functions, and 95 events, supporting 5 task families with 32 subtypes. The stateful design enables efficient task validation and agent evaluation through state correctness and tool-call traces. We evaluate seven models on 2,560 test tasks; the best achieves 68.2% Pass@1. Error analysis reveals challenges in answer delivery, feasibility diagnosis, and temporal coordination. Construction experiments demonstrate scalability across domains and token efficiency, world distillation reduces token consumption by 0.9–66.0% and 46.2–92.5% relative to Codex low and xhigh, while achieving broader capability coverage. SYNTHDCB makes executable world construction the basis of scalable device-control benchmarking, connecting an expanding space of capabilities and interactions to verifiable agent evaluation.

1 INTRODUCTION

Tool-using agents are advancing rapidly and integrated into voice assistants in home, car, or mobile phones, accompanied by a growing collection of benchmarks and executable environments for evaluating their capabilities (Yao et al., 2024; Lu et al., 2025; Seo et al., 2026; Li et al., 2026b). At the same time, device manufacturers and system developers continually introduce new devices and expand their functionality. When simulation environments do not keep pace, the capabilities covered during agent training and evaluation can lag behind those required in deployment. This creates a need for benchmarks that can be rapidly extended as new device capabilities emerge, both to evaluate agents against evolving requirements and to support iterative agent training.

Meeting this need requires addressing two connected challenges. On the construction side, adding a device capability requires consistent tool interfaces, behavior implementations, world configurations, task requirements, and success criteria; generating more instructions over an unchanged environment does not expand what the benchmark can execute. On the evaluation side, agents must understand and operate the environment presented to them, rather than merely invoke controls in a fixed configuration. This requires discovering available devices and operations, interpreting the current world state, and determining which parts of a user request can be satisfied. An extensible benchmark should therefore support both the introduction of new executable capabilities and the assessment of environment understanding and device control across different worlds.

Consider the request: "Cool the bedroom to 22 °C before 22:00, and turn on the bedside lamp at 20% brightness." Changing an air conditioner's setpoint is not the same as bringing the room to the requested temperature: the latter depends on environmental dynamics, and meeting the deadline also depends on when actions are taken. If the cooling requirement is infeasible in the current world, the lighting requirement may still be achievable. Evaluating these distinctions requires explicit action

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.

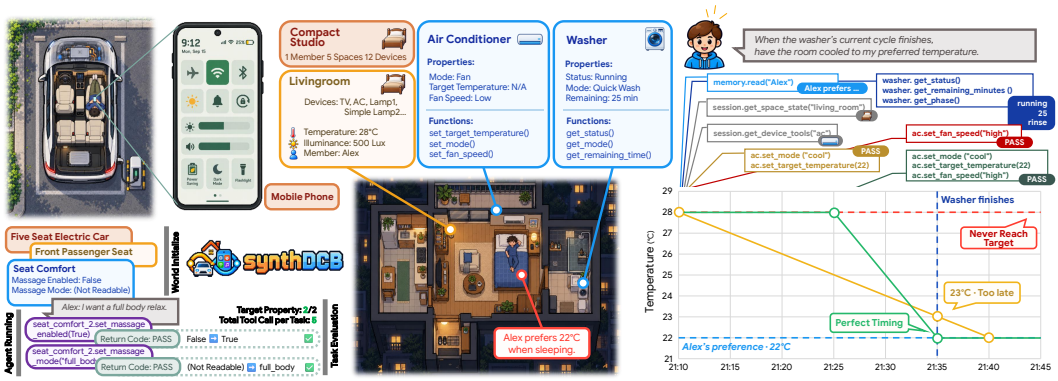


Figure 1: **Overview of SYNTHDCB.** SYNTHDCB builds configurable executable worlds from reusable device capabilities across Home, Car, and Phone. Agents interact with these worlds through device tools and are evaluated on tasks involving environment states, user preferences, and temporal dynamics. The example shows coordinating an air conditioner with a washer cycle to reach Alex's preferred temperature at the required time.

effects, temporal evolution, and success conditions for individual requirements. We therefore ask: *How can we rapidly extend executable device environments and verifiable tasks to evaluate agents' ability to understand and control unfamiliar worlds?*

We introduce the *world distillation* pipeline that transforms behavioral knowledge into reusable executable components for constructing worlds and their tasks. Typed capability specifications guide component implementation and validation, define discoverable tool interfaces, and provide the operations used by task builders. World composition assembles these components into domain-specific layouts or module structures. Execution-grounded task synthesis derives requirements from available capabilities and checks candidate tasks through reference-execution replay. Once validated, a component can be reused across world configurations and task instances without reimplementing its behavior for each one. This reuse supports on-demand device expansion, while configurable worlds enable evaluation of how agents identify and use the capabilities of the environment instantiated for each task.

We instantiate this approach as SYNTHDCB, covering Home, Car, and Phone with 114 device classes, 1,166 functions, and 95 events. The benchmark supports five task families with 32 subtypes, combining state queries, device control, environmental goals, and coordinated execution. Evaluation of seven models on 2,560 test tasks yields a best Pass@1 of 68.2%. Larger models do not consistently perform better, and error analysis reveals persistent difficulties in answer delivery, feasibility diagnosis, and temporal coordination. These findings highlight the gap between invoking device functions and fulfilling user requirements.

Construction experiments demonstrate the scalability and efficiency of world distillation. Building all three domains consumes 19.16M tokens at an estimated cost of $120.09 before cache discounts. In the RobotVacuum comparison, WD achieves broader complete capability coverage than Codex low at comparable token budgets under richer knowledge inputs, and uses 51.5–86.7% fewer tokens per completely covered requirement than Codex xhigh across three knowledge levels. This efficiency supports expanding executable capabilities for agent evaluation and future training.

We summarize our contributions as follows:

- **An extensible environment and world distillation pipeline.** We design a stateful simulator with explicit device behaviors, shared environmental effects, and temporal dynamics, together with a pipeline that transforms device knowledge into executable components and validated tasks.
- **A benchmark and evaluation of seven models.** We introduce SYNTHDCB across Home, Car, and Phone, with five task families covering 32 subtypes. Evaluation on 2,560 test tasks and detailed failure analysis expose distinct gaps in environment awareness and reliable device control.
- **An analysis of construction scalability and efficiency.** We quantify domain construction costs, stage execution, and capability coverage under different knowledge inputs. Comparisons with Codex demonstrate efficient expansion of executable behaviors, providing a foundation for broader evaluation and capability transfer.

Table 1: Comparison of daily agent benchmarks.

Benchmark	Domains	# Tasks	# Types	# Dev.	# Func.	# Inst.	E-Dyn.	Time	Mem.	Infeas.	E-Exp.	T-Exp.
ToolSandbox	Personal Assistant	1,032	7		34					✓		Manual
τ-Bench	Retail, Airline	165			28						Manual	Manual
τ^2-Bench	Telecom	114	15		43						Semi	Semi
DevicesWorld†	Home, Phone, Desktop	6,140	9	11				✓		✓		Semi
VehicleWorld	Car	1,291	4	30	250							Semi
SimuHome	Home	600	12	17	17		✓	✓		✓		Semi
SMH-Bench†	Home	1,100	22	17		3		✓	✓			Semi
SYNTHDCB	**Home, Car, Phone**	**2,560 + 314**	**32**	**114**	**1,166**	**8**	✓	✓	✓	✓	**Auto**	**Auto**

Types: fine-grained task types; Dev.: devices; Func.: functions; Inst.: instruction styles; E-Dyn.: environmental dynamics; Mem.: memory; Infeas.: infeasible tasks; E-Exp.: environment expansion; T-Exp.: task expansion. Auto: fully automated, with no human effort required; Semi: automated with human intervention; Manual: fully human-authored. Blank cells indicate unsupported or not explicitly documented capabilities.
† Statistics and feature annotations for DevicesWorld and SMH-Bench are taken from their manuscripts because the corresponding benchmark artifacts were not publicly available by the time of the ICLR abstract submission deadline.

2 Related Work

Tool-use benchmarking and learning. Tool-use research evaluates and trains agents to complete tasks through external APIs. Call-oriented benchmarks assess tool selection and argument construction (Li et al., 2023; Patil et al., 2025), while synthesis methods generate tool-use supervision (Qin et al., 2024; Prabhakar et al., 2025; Liu et al., 2025). Stateful benchmarks evaluate execution consequences through policy-constrained interactions (Yao et al., 2024; Barres et al., 2026), state dependencies and trajectory milestones (Lu et al., 2025), or application-state checks (Trivedi et al., 2024). Device control further requires modeling how multiple operations affect shared environmental states over time.

Device-control agents and environments. Device-control benchmarks examine instruction understanding, environment discovery, and coordinated execution. Existing approaches emphasize command validity, grounded control, and personalized assistance (Li et al., 2025; Rivkin et al., 2023; Bharadwaj et al., 2026). SimuHome and SMH-Bench introduce home simulators for temporal workflows and contextual reasoning (Seo et al., 2026; Li et al., 2026b), while VehicleWorld and DevicesWorld extend evaluation to in-car interaction and heterogeneous device workflows (Yang et al., 2025; Li et al., 2026a). As device capabilities evolve, these environments must expand their executable behaviors together with the tasks that exercise them. Table 1 situates SYNTHDCB among related benchmarks. SYNTHDCB combines coverage across 3 domains with extensible device behaviors and verifiable tasks, allowing new capabilities to enter evaluation through a shared world distillation pipeline.

Scalable executable environments and tasks. Environment and task synthesis expands agent training and evaluation while aligning tasks with executable outcomes. Environment-centered approaches generate environments alongside tasks or verifiers (Hu et al., 2025; Song et al., 2026; Dong et al., 2026; Zheng et al., 2026); other methods organize function dependencies or combine procedural homes with executable goals (Maekawa et al., 2026; Gu et al., 2026). Scaling device worlds additionally requires reusable behaviors that interact through shared state and simulated time. SYNTHDCB addresses this through world distillation: device knowledge becomes executable components, world composition instantiates their interactions, and task synthesis derives verifiable requirements. Each new component can thus expand both the environment and its task inventory.

3 Method

SYNTHDCB connects executable capabilities to verifiable tasks through a shared simulator and two generation stages: *world distillation* builds reusable device models and composes world instances; *task construction* derives requests, private targets, and reference executions from their available capabilities. This design lets a new device model support many worlds and user requests.

3.1 Simulator Design

Devices, spaces, and worlds. A *device* class $\mathcal{D}$ combines a specification $C^D = (\mathcal{P}^D, \mathcal{T}^D, \mathcal{E}^D)$ of properties, tool-call functions, and events defining environmental effects with a Python imple-

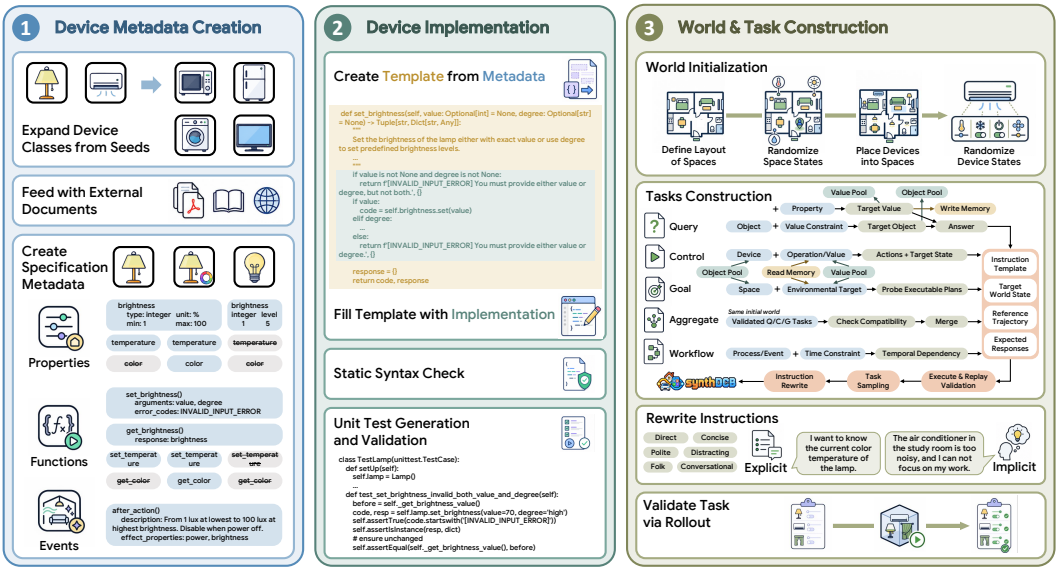


Figure 2: **The system pipeline of world distillation and task construction.** World distillation connects behavioral knowledge to executable capabilities, composed worlds, and validated tasks. A shared capability specification links discovery, execution, and task construction. Component checks drive bounded repair; reference replay validates task construction.

mentation $\mathcal{F}^D$. Instantiating device d assigns its property vector $\mathbf{p}^d_{t_0} = (p^d_{k,t_0})_{k\in\mathcal{P}^D}$, giving the instance state $d_{t_0} = (\mathcal{D}, \mathbf{p}^d_{t_0})$.

A physical or logical *space* class $\mathcal{V}$ specifies shared properties $\mathcal{P}^V$ and metadata. Instantiating space v assigns its property vector $\mathbf{p}^v_{t_0} = (p^v_{k,t_0})_{k\in\mathcal{P}^V}$ and device instances indexed by $\mathcal{I}^D_v$, giving $v_{t_0} = (\mathcal{V}, \mathbf{p}^v_{t_0}, \{d_{t_0}\}_{d\in\mathcal{I}^D_v})$. The states in the same space at the same moment are considered consistent and change synchronously.

A *world* specification $\mathcal{W}$ defines the layout, connections, and device placement rules. Instantiation binds spaces indexed by $\mathcal{I}^V_W$ to this layout, resolves relations and member locations, and initializes runtime state r_{t_0} and clock c_{t_0}. The initial world is $W_{t_0} = (\mathcal{W}, \{v_{t_0}\}_{v\in\mathcal{I}^V_W}, r_{t_0}, c_{t_0})$. Task memory, when provided, is also bound to its runtime state.

Runtime simulation. For interaction round at moment t, the agent submits an action sequence A_t whose tool calls are issued together and executed in the supplied order. Each query reads the state at its execution point; successful control calls immediately update device state, so later queries can observe earlier changes within the same round. The collected tool responses form the observations for the next round, as agent is required to explore the environment by itself. Let E_t denote the collection of currently active environmental effects, each specifying its source, target property, and update rule. The round-level transition of runtime state is

$$A_t = (a_1, \cdots, a_{m_t}), \qquad (W_{t+1}, E_{t+1}, O_{t+1}) = \text{Simulator}(W_t, E_t, A_t), \tag{1}$$

where observation O_{t+1} contains responses of m_t tool calls. The effect collection is refreshed within the round as device states change and time advances. After-action events update immediate contributions, such as illumination; at every one-minute tick, gradual effects such as cooling are recomputed from current device and space states. The runtime combines the applicable effects to update shared environmental properties and advances ongoing device processes. Thus, E_{t+1} captures the active effects after the round. Time advances according to configured action durations, explicit `sleep_time` calls, and the round-end time policy.

3.2 World Distillation

Device lists and specification metadata. World distillation converts behavioral knowledge into reusable executable components under a generation and validation harness. We iteratively expand a domain's device list with descriptions of its functionality and environmental effects, registering base

and inherited classes. For each class, we generate specification metadata defining the properties, functions, constraints, and events above. Schema and capability-consistency checks, followed by semantic review, provide feedback for revision.

Code template, implementation, and tests. Deterministic templates instantiate class structure, property wrappers, and common interfaces using metadata; an LLM completes the remaining function and event bodies and generates unit tests. Static checks, specification–implementation checks, effect validation, and isolated test execution return failures for repair. A dependency-based job planner runs independent jobs in parallel, blocks dependent jobs on failure, and supports reruns of prerequisite stages. We provide scalability and efficiency analysis of world distillation in Section 5.

Inherited classes reuse their parent's specification and implementation through `update` and `disabled` declarations. For example, a cooling-only air conditioner removes heating modes and auxiliary heating and overrides effects related to heating. Stage-specific retrieval supplies domain documents, i.e., user manuals or product list of shopping app, and expert specification, code, and test examples; accepted artifacts are also included in the reference library to support later generation.

World composition. The validated device library $\mathcal{M} = \{(C_i^D, \mathcal{F}_i^D)\}_i$ is composed with space classes $\{\mathcal{V}_j\}_j$ under the world specification $\mathcal{W}$:

$$W_{t_0} = \mathrm{Compose}(\mathcal{M}, \{\mathcal{V}_j\}_j, \mathcal{W}; z), \tag{2}$$

where i and j index device and space classes, and z fixes sampling choices. The layout and placement rules in $\mathcal{W}$ determine space instances, device counts, and assignments. Initialization assigns legal property vectors $\mathbf{p}_{t_0}^d$ and $\mathbf{p}_{t_0}^v$ while preserving coupled-state constraints, resolves connections and member locations, and initializes runtime state r_{t_0} and clock c_{t_0}. The saved configuration fixes W_{t_0} for task construction and replay.

3.3 Task Construction

Capability-grounded sampling. Each instantiated world supplies a shared capability inventory of queryable properties, device operations, processes, and candidate environmental goals. Builders efficiently reuse this inventory and cached execution previews to produce batches of tasks from the same world or valid initial-state variants. Table 2 summarizes the five task families. Target devices or spaces can be selected directly, environment exploration, or through filtering and comparison queries from object pool; target values can come from legal samples, relative changes, queried values from value pool, or memory. Queries that determine later targets or values become explicit prerequisite atom tasks. Goal builders probe executable space change plans, while Aggregate builders combine validated components from the same initial state after checking read/write conflicts and target compatibility.

Table 2: Five families and 32 framework-supported subtypes; domain coverage depends on instantiated capabilities.

Family	Construction target	Subtypes
Query	Information or entity selection	property.device (Q1), property.space (Q2), space.person_location (Q3), space.occupants (Q4), target.quantity (Q5), target.enum (Q6), target.comparison.top1 (Q7), target.comparison.topk (Q8)
Control	Device state or operation	single (C1), multi_property (C2), multi_target (C3), complex (C4)
Goal	Environmental state	single (G1), multi_property (G2), multi_target (G3), complex (G4)
Aggregate	Compatible combined requests	query (A1), control (A2), goal (A3), control_query (A4), control_goal (A5), query_goal (A6), control_query_goal (A7)
Workflow	Process or event timing	single (W1), condition_follow (W2), condition_delay (W3), condition_window (W4), synchronize (W5), at (W6), before (W7), environment_deadline (W8), environment_before_process (W9)

Memory and feasibility. Memory adds user-specific or global preferences: property queries can create or update records, and supported Control and Goal tasks retrieve their target values from saved preferences. LLM-based rewriting makes memory content more natural while introducing semantically similar distractor memories. Feasibility variants introduce missing devices or spaces, unsupported capabilities, invalid values, and goals outside established reachable bounds. The task builder can also sample aggregate tasks combining both executable and infeasible atom tasks; a failed prerequisite blocks its dependent atom tasks while independent atom tasks remain active and should be completed by agent.

Joint construction and validation. A builder produces a structured request q with atomic requirements and dependencies, a private target g, a template instruction u_0, and a reference plan π^{ref}:

$$(q, u_0, g, \pi^{\mathrm{ref}}) = \mathrm{Build}_\phi(W_0; \zeta), \qquad b = (W_0, u_0, g, e), \tag{3}$$

where ϕ is the task family and ζ fixes sampling choices. Executing and replaying the reference plan checks the task structure, instruction anchors, and expected outcomes, including reports for failed requests. The accepted bundle stores this construction evidence in e. An LLM review checks whether the generated intent is coherent, meaningful, and plausible. Interaction protocols and evaluation criteria are described in Section 4.

Instruction rewrite. The instruction of each atom task is sampled through the combination of templates to keep diversity. We further use an optional LLM to rewrite template instructions under seven style policies (Table 3). For basic styles, LLM-rewrite variants are more natural and smooth while preserving necessary information. Distracting variants extend conversational phrasing with irrelevant asides. Implicit variants reduce specificity to test contextual interpretation. The policies thus vary wording, information explicitness, and distracting content.

Table 3: Examples of instruction style variants.

Style	Expression policy	Example
Template	Rule-generated instruction	*For Bedroom Lamp 1, set brightness to 20%.*
Direct	Explicit model terminology	*Set the bedroom lamp's brightness to 20%.*
Concise	Compact, explicit wording	*Bedroom lamp: brightness 20%.*
Polite	Courtesy, explicit wording	*Please set the brightness of bedroom lamp to 20%.*
Conversational	Natural everyday phrasing	*Could you dim the bedroom lamp to 20%?*
Folk	Colloquial phrasing	*Turn that bedroom light down to 20%, will ya?*
Distracting	Conversation with irrelevant asides	*That bedroom lamp looks old-fashioned. Could you dim it to 20%?*
Implicit	Broad intent, omitted details	*Make the bedroom light a little dimmer.*

4 THE SYNTHDCB BENCHMARK

4.1 DATASET AND SPLITS

SYNTHDCB covers Home, Car, and Phone with 114 device classes and 1,166 tools. Its current evaluation splits contain **314 validation tasks** (150 Home, 105 Car, and 59 Phone) and **2,560 test tasks** (1,281 Home, 768 Car, and 511 Phone). Both splits are sampled disjointly from a shared pool of 30,309 tasks constructed from 130 worlds. We use validation set for ablation studies of 8 instruction styles, and perform main evaluation of all 7 models with test set using conversational style as it better simulates daily life scenarios.

The test set includes 2,289 feasible, 170 infeasible, and 101 partial tasks, with memory requirements in 200 tasks. Appendix B reports detailed statistics of SYNTHDCB; Table 11 gives the complete distribution of every task family and subtype.

4.2 AGENT AND INTERACTION PROTOCOL

All 7 evaluated models[1] use the same ReAct-style agent (Yao et al., 2023) and interact with SYNTHDCB through iterative observations and actions. Agents discover spaces, devices, and function schemas with system level tools before invoking device functions in later turns, matching the discovered targets and parameter schemas.

Query answers use `session.submit_answer`, one per question in instruction order with required JSON types. Each infeasible or blocked atom task needs a `session.submit_report` with status, reason code, and explanation; partial tasks also require completing all independently executable atom tasks.

Simulated time advances through configured action/turn costs (inactivated by default) and `session.sleep_time`. Interaction finishes when an agent sends `session.finish` or reaches the maximum limit of 20 turns. For tasks containing atom tasks of Goal and Workflows, background

[1] API requests set the temperature to 0 and top-p to 1, with a maximum of 16,384 output tokens per response, while all other parameters remain at their default values.

Table 4: Main results on SYNTHDCB. Higher is better.

Model	Home†	Car†	Phone†	Overall†	Eff-A	Eff-T	Partial/Infeasible				Memory		
							Inf.†	Part.†	RC	Sem.	Mem.†	Read	Write
Claude Haiku 4.5 (Anthropic, 2025)	40.5	40.0	71.6	46.6	0.17	0.52	45.3	56.4	53.0	36.7	28.5	72.8	28.4
DeepSeek V4.1 Flash (DeepSeek-AI, 2026)	66.7	52.0	**85.5**	66.0	0.16	0.53	67.1	77.2	71.6	56.5	67.5	74.6	75.0
Gemini 3.8 Flash (Google DeepMind, 2026)	20.7	24.7	26.8	23.1	0.12	0.11	0.0	32.7	22.7	18.3	16.5	36.0	15.9
GLM-5.3 Flash (Z.AI, 2026)	36.8	39.2	36.8	37.5	0.16	0.50	77.6	28.7	78.2	45.4	16.0	87.7	68.2
GPT-5.6 Luna (OpenAI, 2026)	**67.5**	**61.7**	79.8	**68.2**	0.23	0.56	65.3	67.3	66.4	61.6	55.5	89.5	51.1
GPT-5.6 Terra (OpenAI, 2026)	65.1	60.4	77.5	66.2	0.23	0.56	65.9	66.3	66.4	63.5	53.5	88.6	51.1
GPT-5.6 Sol (OpenAI, 2026)	65.8	61.3	78.9	67.1	0.23	0.55	67.1	64.4	66.6	64.0	52.0	90.4	48.9

† Pass@1 percentages. Inf.: on 170 infeasible tasks. Part.: on 101 partial tasks. Mem.: on 200 memory related tasks.
‡ Eff-A/T measure achieved state/answer atoms per tool call/turn. RC: infeasible reason code report accuracy. Sem.: infeasible reason report semantic score. Read/Write: success rate on 114 memory-conditioned Control/Goal atoms and 88 memory-write atoms.

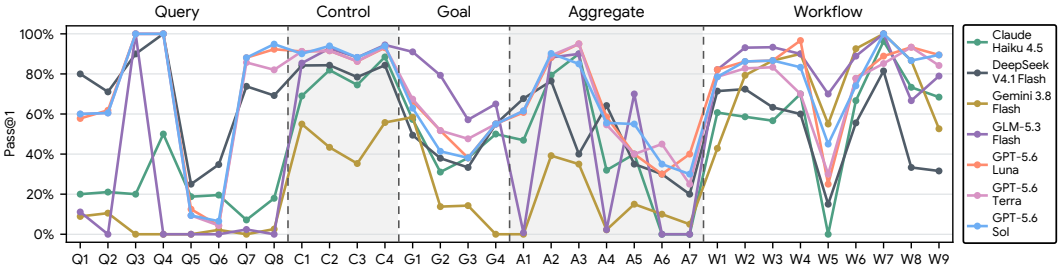


Figure 3: Pass@1 by task subtype.

simulation continues until all workflows are finished and world state is finite or reach time limit. Other tasks are evaluated when interaction ends.

4.3 EVALUATION METRICS

Full metrics are defined in Appendix B.6. We use the Pass@1 as main metric, requiring passing all applicable checks in one attempt. The checks cover states, answers, reports, memory, and temporal/workflow constraints. For partial/infeasible tasks we also report reason-code and semantic report accuracy. For memory-related tasks we report separate success rates for memory-conditioned Control/Goal and memory-write atoms. Eff-A and Eff-T count satisfied state assertions and exactly matched query answers per tool call and interaction turn, respectively, for further tool call efficiency inspection.

4.4 MAIN RESULTS

Larger models are not necessarily better. As shown in Table 4, GPT-5.6 Luna achieves the highest overall Pass@1 at 68.2%, closely followed by Sol and Terra. Luna leads Home and Car, while DeepSeek leads Phone; but even the best overall model leaves roughly one-third of tasks unsolved. The changing rankings across domains and task types show that SYNTHDCB examines complementary abilities in environment understanding, tool use, coordinated execution, and reporting. Larger models with greater computational and time costs do not necessarily lead to improved performance. In contrast, a smaller model optimized for tool-calling scenarios may have an advantage.

Subtype performance reveals distinct strengths. Figure 3 shows that difficulty varies substantially within each task family. GPT variants achieve 82.1–94.9% on top-k queries (Q8), but less than 13% on counting and enumeration (Q5–Q6); the error audit below links these failures primarily to answer submission. They also handle complex device controls well, while multi-target environmental goals and synchronization remain harder. DeepSeek leads device/space property queries and query-only or control–query Aggregates, whereas GLM combines strong single-target Goals and synchronization with weak Query performance. These differences distinguish retrieving and reporting information, setting device states, and coordinating environmental or temporal outcomes.

Feasibility and memory add separate challenges. GLM leads infeasible tasks at 77.6% but solves only 28.7% of partial tasks, which require both diagnosing unavailable operations and completing other independent requests. Across models, reason-code accuracy exceeds semantic report accuracy, exposing gaps in request-specific explanations even when the failure category is correct.

Memory-related tasks similarly separate using stored context from preserving new observations: GPT variants succeed on about 90% of memory-conditioned Control/Goal atoms, but only about half of memory-write atoms. DeepSeek leads full memory-task success and memory writing, despite its lower overall rank. Thus, accurate context use, faithful updates, and completion of the surrounding request remain distinct requirements.

4.5 Instruction-Style Ablation

We compare eight instruction styles on the validation set (Table 5). The six non-implicit rewrite styles undergo automated alignment checks to preserve required entities, values, quantities, and temporal conditions. These rewrites affect models differently: Sol and Terra show larger performance drops than Luna, while Concise improves Luna's Pass@1 over Template by 1.9 percentage points. Device-property accuracy remains above 95% across these styles, even when Pass@1 decreases.

Table 5: Instruction-style ablation.

Style	Sol	Terra	Luna
Template	66.6	65.3	60.5
Direct	65.0 (-1.6)	63.1 (-2.2)	61.8 (+1.3)
Concise	61.1 (-5.4)	60.2 (-5.1)	62.4 (+1.9)
Polite	62.4 (-4.1)	60.8 (-4.5)	59.2 (-1.3)
Conversational	57.0 (-9.6)	57.6 (-7.6)	59.6 (-1.0)
Folk	60.8 (-5.7)	60.5 (-4.8)	60.5 (+0.0)
Distracting	61.1 (-5.4)	60.2 (-5.1)	61.8 (+1.3)
Implicit	36.9 (-29.6)	35.4 (-29.9)	38.9 (-21.7)

† GPT-5.6 models evaluated on validation set.

Partial state achievement under implicit instructions. Implicit rewrites deliberately obscure or omit specific details to examine whether agents can still change device and environmental states toward the intended outcomes. This setting changes the available information and should therefore be analyzed separately from the aligned non-implicit styles, rather than as a directly comparable wording variation. We retain the original task targets and use component metrics to measure partial achievement. Relative to the fully specified Template reference, average Pass@1 falls by 27.1 percentage points, while device-property accuracy falls by 20.2 points to 76.9%. Agents thus recover some actionable requirements despite missing details. Environmental targets remain harder: average space-property accuracy falls from 75.6% to 27.3%, revealing greater difficulty in translating broad intent into environmental outcomes (Table 15).

4.6 Error Analysis

We analyze 8,328 failed attempts by model, task type, and domain in Figure 4. An attempt may receive multiple error labels; we split its weight equally among these labels and report each category's share of the total failure weight within its group.

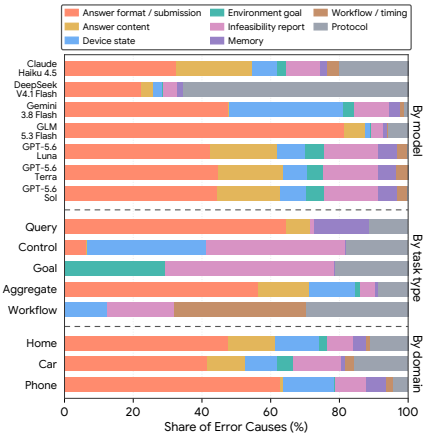


Figure 4: Failure composition.

Models have different bottlenecks. GLM's strong state control coexists with answer submission and representation errors, which account for 81.4% of its attributed failure weight, particularly JSON values encoded as strings. DeepSeek's dominant bottleneck is tool protocol compliance, accounting for 65.5% of its attributed failure weight, frequently through responses without tool calls. GPT variants rarely violate the calling protocol, but remain vulnerable to answer submission, answer content, and infeasibility reporting. Claude combines protocol and answer-content errors, while Gemini frequently omits answers and fails device-state targets. These profiles distinguish reliable tool invocation, effective state control, and correct delivery of the requested information.

Task and domain demands shape failures. Query and Aggregate failures concentrate on answer extraction and submission, with memory also prominent in Query tasks. Control and Goal tasks additionally require feasibility diagnosis: reporting errors form their largest failure shares, followed by device-state and environmental-state errors, respectively. Workflow failures primarily concern timing, exposing the need to coordinate event transitions beyond achieving final states. Across domains, Home has the largest answer-content error share, while Car places more weight on infeasibility reporting, environmental goals, and protocol compliance. Phone has the largest shares of answer-submission and memory errors. These distributions connect model weaknesses to the capabilities and task composition exercised by each domain.

5 World Distillation Scalability and Efficiency Analysis

Capability coverage and token efficiency. To quantify the efficiency of world distillation, we compare World Distillation (WD) and Codex on constructing a RobotVacuum class from scratch with three levels of external knowledge: a brief summary (L1, 0.06k characters), a functional overview (L2, 0.77k characters), and the full device protocol (L3, 89.8k characters). We also summarized capability requirements from the full protocol. Both methods use `gpt-5.6-sol`, with Codex evaluated at low and xhigh reasoning effort. Experiment settings are detailed in Appendix C.

As shown in Table 6, WD uses **0.9–66.0% and 46.2–92.5% fewer tokens** than Codex low and xhigh across the three knowledge levels. At comparable token budgets, WD achieves higher complete coverage than Codex low across properties, functions, and effect/event requirements at both L2 and L3. Compared with Codex xhigh, WD uses **51.5–86.7% fewer tokens per completely covered requirement**. At L1, the brief description provides little protocol detail, and all methods achieve limited coverage; WD's coverage advantage emerges with richer knowledge at L2 and L3.

Table 6: Comparison of device model construction.

Method	#Devices	#Properties†	#Functions†	#Events†	Tokens	Min.
L1-Codex low	1	7 (6/1)	13 (1/7)	1 (6/5)	1.48M	10.09
L1-Codex xhigh	1	12 (6/1)	21 (1/7)	2 (9/3)	6.74M	12.48
L1-WD	1	7 (3/1)	12 (0/5)	2 (6/3)	0.50M	11.45
L2-Codex low	1	23 (0/19)	39 (6/23)	0 (14/15)	1.50M	10.16
L2-Codex xhigh	1	23 (18/1)	39 (6/23)	1 (14/16)	5.46M	11.71
L2-WD	1	25 (18/0)	49 (11/17)	2 (30/8)	1.15M	24.31
L3-Codex low	1	19 (9/14)	29 (11/18)	1 (12/18)	2.77M	14.40
L3-Codex xhigh	1	18 (23/0)	29 (22/7)	2 (19/16)	5.09M	13.64
L3-WD	1	25 (22/0)	57 (19/10)	2 (30/7)	2.74M	26.95
Home	54	330	616	49	11.57M	48.09
Car	38	177	333	29	4.83M	33.45
Phone	22	123	217	17	2.76M	32.33

† Parentheses report complete/partial coverage of 23 property/query dimensions, 29 protocol functionalities, and 39 effect/event requirements, respectively.

Table 7: Construction stages.

#Attempts	Spec.	Temp.	Impl.	UniT.	Total
1	54	109	92	72	327
2	27	5	16	31	79
3	18	0	2	7	27
4+	15	0	4	4	23
Total	251	119	154	179	703
#Calls	493	0	134	202	829
Tokens	10.71M	0	3.96M	4.50M	19.16M

Spec./Temp./Impl./UniT.: specification, template, implementation, and tests.

Domain construction cost. We construct device models for all three domains using a job planner with 16 workers per domain, with a combined construction time of **113.87 minutes**. The full pipeline consumes **19.16M tokens**, costing approximately **$120.09** before cache discounts. By automating specification, implementation, and validation, WD largely reduces the manual effort required to expand executable environments. Our construction prioritizes efficient coverage of device capabilities for agent testing and capability transfer, rather than production-ready device integration. Stage-level analysis shows that most classes require only one recorded execution for template generation, implementation, and testing, while specification generation requires the most repeated executions (Table 7). These results demonstrate a practical path to expanding behavioral coverage across domains and supporting the evaluation and development of device-control agents.

6 Conclusion and Future Work

We introduced SynthDCB, a scalable benchmark for environment awareness and device control across Home, Car, and Phone. Through world distillation, SynthDCB converts device knowledge into executable worlds (Section 3.2), supporting task construction and evaluation through state and process checks (Sections 3.3 and 4.3). Our analyses show that larger models do not consistently perform better, and even leading models struggle with answer delivery, feasibility diagnosis, and temporal coordination (Section 4.6). Strong direct control also does not ensure success on environmental goals (Section 4.4), highlighting the need to connect tool use with device effects and task constraints. Construction experiments demonstrate scalable world creation with efficient token use and broad behavioral coverage (Section 5). Future work will evaluate more models and explore using the extensible pipeline to generate training tasks that address these capability gaps, helping smaller models achieve reliable device control with faster inference and local deployment.

AI USE STATEMENT

Generative AI is used in the benchmark construction pipeline for capability specifications, behavior-code completion, test generation, repair, and instruction rewriting, and in configured semantic evaluation. AI coding and writing assistants like Codex, Claude Code, ChatGPT are also used to inspect source code and existing results, draft and revise manuscript text, prepare illustrative scripts and figures, and check document compilation. Appendix E documents the generation and evaluation prompts.

REPRODUCIBILITY STATEMENT

Appendices A and B describe the construction stages, temporal semantics, task taxonomy, experimental protocols, and scoring rules. Appendix E documents the generation and evaluation prompts. Appendix C specifies the experimental protocols of world distillation analysis. Appendix D gives the disaggregated results. The code and data are available at `https://anonymous.4open.science/r/SynthDCB_ICLR`.

ETHICS STATEMENT

Human Involvement All human-involved checks, including inspection of generated artifacts, were conducted by the authors. No crowdworkers or students participated in this work.

Privacy SYNTHDCB uses synthetic simulated device worlds. Scenario object identifiers, initial states, target states, and user instructions are artificial and are not linked to real people, households, or devices. No personal user logs, conversations, sensor traces, or private device data are collected or curated, so human-data anonymization is not required. Generated artifacts undergo schema validation, replay validation, and author inspection; any personally identifying content encountered is excluded.

Intended Use and Potential Risks SYNTHDCB provides benchmark code, generated device-class specifications, simulated scenarios, tool schemas, reference trajectories, supervised tool-call records, and evaluation scripts for benchmark evaluation and controlled agent-training research. Generation is constrained to daily device-control operations over predefined device properties and functions, and invalid or offensive generated content is excluded if encountered during validation and inspection. Results on SYNTHDCB measure tool-use competence in simulated environments and do not establish readiness for real-world autonomous device control. Direct transfer to physical systems, especially appliances, vehicles, locks, heating systems, and other safety-critical devices, may introduce safety risks. Real-world deployment requires independent safety validation, permission management, human oversight, rollback mechanisms, and domain-specific constraints on high-risk actions.

Licensing Publicly released artifacts will be accompanied by a license or terms-of-use file. External models and software artifacts are used under their respective licenses or provider terms, and proprietary models are accessed through API services.

REFERENCES

Anthropic. Introducing Claude Haiku 4.5, 2025. URL https://www.anthropic.com/news/claude-haiku-4-5. Official model announcement; accessed 2026-09-25.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik R. Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=OC2z7iSQKa.

Manasa Bharadwaj, Yolanda Liu, InJung Yang, Sungil Kim, Nikhil Verma, KoKeun Kim, Kevin Ferreira, and YoungJoon Kim. PersonalHomeBench: Evaluating agents in personalized smart homes. *arXiv preprint arXiv:2604.16813*, 2026. URL https://arxiv.org/abs/2604.16813.

DeepSeek-AI. Models & pricing: DeepSeek-V4.1-Flash, 2026. URL https://api-docs.deepseek.com/quick_start/pricing/. Official model documentation; accessed 2026-09-25.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, Jiajie Jin, Yutao Zhu, Hanbin Wang, Fangyu Lei, Qinyu Luo, Mingyang Chen, Zehui Chen, Jiazhan Feng, Ji-Rong Wen, and Zhicheng Dou. AgentWorld: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026. URL https://arxiv.org/abs/2604.18292.

Google DeepMind. Gemini 3.8 Flash model documentation, 2026. URL https://ai.google.dev/gemini-api/docs/models/gemini-3.8-flash. Official model documentation; accessed 2026-09-17.

Yi Gu, Huacan Wang, Shuo Zhang, Yuqing Hou, Lei Xue, Weipeng Ming, Chen Liu, Fangzhou Yu, Kuan Li, Ronghao Chen, Sen Hu, Xiaofeng Mou, and Yi Xu. HomeFlow: A data flywheel for smart home agent training with verifiable simulation. *arXiv preprint arXiv:2606.01230*, 2026. URL https://arxiv.org/abs/2606.01230.

Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jianguang Lou, Qingwei Lin, Ping Luo, and Saravan Rajmohan. AgentGen: Enhancing planning abilities for large language model based agent via environment and task generation. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2025. URL https://arxiv.org/abs/2408.00764.

Huatao Li, Xinwei Geng, Yuheng Wang, Yutong Li, Runde Yang, Hantao Chen, Shu Yao, Jingru Fan, Xuhui Ren, Yuanyuan Zhao, Fei Huang, and Chen Qian. DevicesWorld: Benchmarking cross-device agents in heterogeneous environments. *arXiv preprint arXiv:2607.13465*, 2026a. URL https://arxiv.org/abs/2607.13465.

Kuan Li, Shuo Zhang, Huacan Wang, Fangzhou Yu, Zecheng Sheng, Yi Gu, Weipeng Ming, Lei Xue, Chen Liu, Sen Hu, Ronghao Chen, Siyue Lin, Yuqing Hou, Xiaofeng Mou, and Yi Xu. SMH-Bench: Benchmarking LLM agents for environment-grounded reasoning and action in smart homes. *arXiv preprint arXiv:2606.01912*, 2026b. URL https://arxiv.org/abs/2606.01912.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116, 2023. doi: 10.18653/v1/2023.emnlp-main.187. URL https://aclanthology.org/2023.emnlp-main.187/.

Silin Li, Yuhang Guo, Jiashu Yao, Zeming Liu, and Haifeng Wang. HomeBench: Evaluating LLMs in smart homes with valid and invalid instructions across single and multiple devices. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12230–12250, 2025. doi: 10.18653/v1/2025.acl-long.597. URL https://aclanthology.org/2025.acl-long.597/.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, Duyu Tang, Dandan Tu, Lifeng Shang, Xin Jiang, Ruiming Tang, Defu Lian, Qun Liu, and Enhong Chen. ToolACE: Winning the points of LLM function calling. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=8EB8k6DdCU.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Haoping Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. ToolSandbox: A stateful, conversational, interactive evaluation benchmark for LLM tool use capabilities. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 1160–1183, 2025. doi: 10.18653/v1/2025.findings-naacl.65. URL https://aclanthology.org/2025.findings-naacl.65/.

Seiji Maekawa, Jackson Hassell, Pouya Pezeshkpour, Tom Mitchell, and Estevam Hruschka. Towards reliable benchmarking: A contamination free, controllable evaluation framework for multi-step LLM function calling. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=UKYCJixSFt.

OpenAI. GPT-5.6 model family: Sol, Terra, and Luna, 2026. URL https://developers.openai.com/api/docs/changelog#july-2026. Official API release notes; accessed 2026-09-25.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The Berkeley Function Calling Leaderboard (BFCL): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392. PMLR, 2025. URL https://proceedings.mlr.press/v267/patil25a.html.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Manoj Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Quoc Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. APIGen-MT: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. In *Advances in Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=qk6ORqQ4Cu.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=dHng2O0Jjr.

Dmitriy Rivkin, Francois Hogan, Amal Feriani, Abhisek Konar, Adam Sigal, Steve Liu, and Greg Dudek. SAGE: Smart home agent with grounded execution. *arXiv preprint arXiv:2311.00772*, 2023. URL https://arxiv.org/abs/2311.00772.

Gyuhyeon Seo, Jungwoo Yang, Junseong Pyo, Nalim Kim, Jonggeun Lee, and Yohan Jo. SimuHome: A temporal- and environment-aware benchmark for smart home LLM agents. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2509.24282.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. EnvScaler: Scaling tool-interactive environments for LLM agent via programmatic synthesis. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8326–8357, 2026. doi: 10.18653/v1/2026.findings-acl.407. URL https://aclanthology.org/2026.findings-acl.407/.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024. URL https://arxiv.org/abs/2407.18901.

Jie Yang, Jiajun Chen, Zhangyue Yin, Shuo Chen, Yuxin Wang, Yiran Guo, Yuan Li, Yining Zheng, Xuanjing Huang, and Xipeng Qiu. VehicleWorld: A highly integrated multi-device environment for intelligent vehicle interaction. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 403–442, 2025. URL https://aclanthology.org/2025.findings-emnlp.23/.

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. URL https://arxiv.org/abs/2210.03629.

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

Z.AI. GLM-5.3-Flash/FlashX, 2026. URL https://docs.z.ai/guides/vlm/glm-5.3-flash. Official model documentation; accessed 2026-09-25.

Weihuang Zheng, Tianyuan Zou, Eileen Ye, Alphet Liu, Youyong Kong, Ya-Qin Zhang, Duran Zheng, and Maxm Pan. E-Bench: Benchmarking multi-step tool-use agents in real-world product scenarios. *arXiv preprint arXiv:2607.23722*, 2026. URL https://arxiv.org/abs/2607.23722.

APPENDIX CONTENTS

A World Distillation and Simulation Details **15**
- A.1 Capability Specification and Interfaces . . . 15
- A.2 Component Generation and Repair . . . 15
- A.3 Runtime and Temporal Semantics . . . 15
- A.4 World and Task Construction Algorithms . . . 16

B Benchmark Specification and Evaluation **17**
- B.1 Device and Capability Inventory . . . 17
- B.2 System Tools and Device Examples . . . 18
- B.3 Task Taxonomy and Coverage . . . 19
- B.4 Feasibility and Memory Settings . . . 19
- B.5 Instruction Variants and Alignment Checks . . . 20
- B.6 Complete Metric Definitions . . . 20

C World Distillation Settings **21**

D Additional Results and Case Studies **22**
- D.1 Full Model-by-Task Results . . . 22
- D.2 Instruction-Style Results and Execution Diagnostics . . . 22
- D.3 Device Construction Coverage . . . 22
- D.4 Error Taxonomy and Annotated Trajectories . . . 23
- D.5 Complete Error-Attribution Panels . . . 28

E Prompts **31**
- E.1 World Distillation . . . 31
- E.2 Task Creation . . . 55
- E.3 Agent . . . 58
- E.4 LLM-as-Judge . . . 59

A World Distillation and Simulation Details

A.1 Capability Specification and Interfaces

A device specification contains metadata, properties, functions, and events, with optional inheritance and process declarations. Properties declare their type, default, access mode, units, and admissible values. The implementation supports booleans, integers, numbers, strings, and structured collections. Numeric declarations can specify bounds, step origins, and precision; collection declarations specify their structure. Read-only, read/write, and disabled properties are distinguished explicitly. Inheritance resolves updates and disabled members before the effective specification is used for code generation or loading.

Functions declare a JSON parameter schema, mutation status, read and write sets, and semantics such as observation, assignment, collection mutation, or process control. Preconditions and argument sources connect those operations to the state they require. The loader checks that referenced properties exist and that the declaration agrees with the implementation. For an instance exposing a subset of properties, a function is exposed only if all its declared property dependencies are present. This rule also determines which actions task construction can use.

Events describe continuous effects, effects following an action, or responses to declared dependencies. Home space effects operate on temperature, humidity, illuminance, noise, and air quality; each domain catalog determines its own legal targets and units. Phone modules update local state and declared dependencies without Home space effects. Optional process specifications describe modes, phases, phase durations, and persisted progress variables. Public methods return a status and response; event hooks return a status and a list of effects.

A.2 Component Generation and Repair

Generation is organized around artifacts with explicit dependencies. The type inventory and placement constraints identify candidate components. The metadata stage combines the request, domain property catalog, inherited specification, schema, and retrieved references. Template assembly constructs the class and property wrappers, after which code completion implements public actions and event hooks. Test generation uses the specification and implementation to exercise valid calls, invalid inputs, boundary conditions, power behavior, and effects.

The production gate performs deterministic schema and declaration checks, inspects generated source against its execution restrictions, checks specification–implementation consistency, and executes generated unit tests in an isolated subprocess. Errors are returned to the earliest relevant stage, preserving the requested capabilities and bounded repair settings. Attempt records retain failures, auxiliary generation calls, usage, and repair lineage. An optional type-list judge provides annotations of description quality; it does not select or promote classes.

A.3 Runtime and Temporal Semantics

The runtime executes commands against a validated world and maintains a separate audit of command results, state changes, and time ticks. Device actions update their own state and may emit space effects. During a minute tick, hooks observe a common pre-tick snapshot, and the reducer combines their effects before committing environmental updates. Contributions replace the previous contribution from the same source; additive deltas are accumulated; approach effects move a baseline toward declared targets at bounded rates. The domain contract validates allowed combinations and units, then clamps and rounds values using the property's specification. These rules define the simulator's behavior abstraction and are not a calibrated physical model.

Command execution and time advancement use state checkpoints so rejected or failing operations do not leave partial committed updates. The simulator advances in integer minutes. Waiting, declared action durations, and configured turn costs determine elapsed time; model latency does not advance the simulation clock. Independent calls can share a turn, while calls requiring a new observation must occur in a subsequent turn. Device-tool discovery is also required in a preceding turn under the benchmark configuration.

The agent's finish signal ends interaction immediately. When a private target defines an evaluation horizon, background simulation continues to that fixed horizon without further agent actions. Goal construction derives a horizon from the reference duration d:

$$t_{\text{eval}} = t_0 + \max\left(60,\ 60\left\lceil \frac{2d}{60} \right\rceil\right). \tag{4}$$

Workflow targets supply their own horizon and transition relations. The horizon is exposed in the agent's public limits and remains fixed when the agent waits or finishes. Without a horizon, final-state checks use the interaction-end state. Previously submitted answers retain their submission-time interpretation.

Workflow scoring records the first qualifying transition for each required event. Ordered relations use minute and audit sequence to disambiguate events in the same minute; other relations check bounded delays, synchronization, specified times, or strict "before" deadlines. The legacy target-level `by`/`within` constraint uses an inclusive deadline. This distinction is represented in the target and instruction.

A.4 World and Task Construction Algorithms

The world constructor samples a supported layout, assigns compatible components, samples legal property values, applies initialization operations, and loads the resulting world through the shared contract. A task may derive a perturbed initial state from a source world; its effective state is saved with the task. The following algorithm summarizes the active task-building path.

1. Load a world and build an inventory of exposed properties, operations, processes, and environmental effects.
2. Sample a task family and candidate requirements from that inventory. Preview operations to bind their resulting states, answers, and relevant time transitions.
3. Construct the private target and a reference plan. For a negative requirement, retain its construction reason; for a partial task, retain executable requirements as well.
4. Replay the plan from a fresh copy of the effective initial world. Run schema, public/private, instruction-alignment, termination, and evaluator checks.
5. Accept the complete bundle when its reference satisfies the target. Preserve unsuccessful candidates in the generation records.
6. Produce instruction styles from the accepted requirements, retaining the same base-task identity and paired assets.

Goal probing has a finite search budget. Exhausting that budget records a construction failure; an unreachable-goal label requires separate construction evidence. Aggregate construction merges compatible requirements, targets, memories, and reference operations from the same effective world, then validates the merged task.

B Benchmark Specification and Evaluation

B.1 Device and Capability Inventory

Tables 8 and 9 report the per-domain component and dataset statistics and the device-class inventory, respectively. The validation and test splits draw from the shared source-world pool; the split unit is the base task. Tasks from the two sets may share the same world instance. An atom (task) is an entry in the constructed task graph, including prerequisite observations, memory writes, and infeasible or blocked requirements. Repeated occurrences across tasks are counted separately, while instruction variants and evaluation attempts do not add atoms.

The base/derived class counts are 36/18 for Home, 30/8 for Car, and 21/1 for Phone. Across the test-task configurations, Home covers 17 space types, with 5–22 spaces and 10–125 device instances per task world. Car configurations have 11–13 spaces and 40–52 devices; Phone has two spaces and 21 module instances. Task-local initialization specifies each task's starting conditions. Component-class counts describe the reusable library, while instance counts describe composed worlds.

Table 8: Component inventory and dataset size.

Domain	Classes	Functions	Properties (Writable)	Events	Source worlds	Pool tasks	Valid tasks / atoms	Test tasks / atoms
Home	54	616	330 (229)	49	60	15,877	150 / 2,189	1,281 / 17,211
Car	38	333	177 (83)	29	40	9,918	105 / 1,087	768 / 7,023
Phone	22	217	123 (41)	17	30	4,514	59 / 316	511 / 2,730
Total	114	1,166	630 (353)	95	130	30,309	314 / 3,592	2,560 / 26,964

Table 9: Device classes by domain.

Domain	Component classes
Home	Air Conditioner, Air Exchanger, Air Fryer, Air Purifier, Bathroom Heater, Bathroom Mirror, Bluetooth Speaker, Cat Litter Box, Ceiling Fan, Coffee Maker, Color Lamp, Cooling Only Air Conditioner, Curtain, Dehumidifier, Dishwasher, Door Lock, Dryer, Drying Rack, Dual Layer Curtain, Electric Water Heater, Evaporative Humidifier, Fan, Floor Washer, Fragrance Diffuser, Gas Water Heater, Hat Lamp Bathroom Heater, Heater, Humidifier, Induction Cooker, Kettle, Lamp, Microwave, Monitor, Oven, Pet Feeder, Pet Water Dispenser, Projector, Range Hood, Refrigerator, Rice Cooker, Robot Vacuum, Simple Lamp, Speaker, Surveillance Camera, Television, Toilet, Ultrasonic Humidifier, Warm Air Bathroom Heater, Washer Dryer, Washing Machine, Water Dispenser, Water Heater, Water Purifier, Window Opener
Car	Adaptive Damping, Air Suspension, Ambient Light, Cabin Light, Car Display, Car Refrigerator, Charging, Climate Control, Cooling Heating Car Refrigerator, Dashcam, Door, Drive Settings, Driver Assistance, Dynamic Ambient Light, Electrified Drive Settings, Exterior Light, External Power Supply, Fragrance, Fuel System, Head Up Display, Hybrid Drive Settings, Infotainment, Memory Seat Comfort, Mirror, Navigation, Parking Camera, Rear Entertainment Display, Seat Comfort, Seat Heater, Seat Ventilator, Steering Wheel, Sunroof, Sunshade, Tailgate, Trunk Door, Vehicle Security, Window, Wiper
Phone	Accessibility, Alarm, Battery, Bluetooth, Connectivity, Dimmable Flashlight, Display, Flashlight, Focus, Hotspot, Location, Media Player, Notifications, Privacy, Rotation, Screen Capture, Sound, Stopwatch, System, System Settings, Timer, Wi Fi

B.2 System Tools and Device Examples

Table 10 lists the system and memory tools, device dispatcher, and Home device examples with their arguments and conditions.

Table 10: System tools and device examples.

Tool / device function	Arguments	Purpose and conditions
Session tools		
`session.get_capabilities`	{}	Return the domain, support for spaces, devices, and relations, and the simulation time unit.
`session.get_devices`	`space_id`	List devices in a space with their IDs, classes, locations, and metadata.
`session.get_relations`	{}	Return the relations declared in the domain metadata.
`session.get_spaces`	{}	List space IDs, types, and whether each space is virtual.
`session.get_space_state`	`space_id`	Read a space's properties and metadata.
`session.get_space_devices`	`space_id`	List the IDs and classes of devices in a space.
`session.get_device_tools`	`device_id`	Retrieve device functions, parameter schemas, and read/write semantics. Required before invoking those functions in a later turn.
`session.sleep_time`	`[minutes]`	Advance simulation by a nonnegative integer number of minutes; omission uses the configured default.
`session.submit_answer`	`value`	Submit one typed JSON answer per query, in instruction order.
`session.submit_report`	`status`, `reason_code`, `description`, `[affected_properties]`	Report one infeasible or blocked requirement, with its reason and supporting explanation.
`session.finish`	{}	End interaction; background simulation continues to the fixed evaluation horizon when one applies.
Memory tools		
`memory.read`	{}	Read memory IDs and contents in the current scope.
`memory.create`	`content`, `[user_id]`	Create a natural-language memory record, optionally associated with a user.
`memory.update`	`memory_id`, `content`	Replace a previously read record's content using its returned memory ID.
Device dispatcher		
`dcb_device_command`	`target`, `function`, `parameters`	Invoke a discovered device function with its exact target and parameter schema; `parameters` is a JSON object.
Home device examples		
Read (AirConditioner) `get_target_temperature`	{}	Read the target temperature; requires power on and mode `cool`, `heat`, or `auto`.
Write (AirConditioner) `set_target_temperature`	{`"value": 22`}	Set the target to 22 °C (integer range: 16–30), under the same power and mode conditions.
Complex control (WasherDryer) `start_cycle`	{`"mode": "wash_and_dry"`}	Start a wash-and-dry cycle; requires power on and status `idle` or `completed`. Jointly update mode, status, phase, and timing; simulated time advances the cycle.

Brackets mark optional arguments. Device examples give the `function` and `parameters` for `dcb_device_command`; `target` is the discovered device ID.

B.3 Task Taxonomy and Coverage

Table 11 reports validation and test task counts by domain, family, and subtype.

Table 11: Task distribution by domain, family, and subtype.

Type / subtype	Home		Car		Phone	
	Valid	Test	Valid	Test	Valid	Test
Query	**37**	**175**	**21**	**100**	**15**	**25**
property.device	25	10	11	10	15	25
property.space	2	57	2	19	0	0
space.occupants	1	10	0	0	0	0
space.person_location	1	10	0	0	0	0
target.comparison.top1	2	28	2	14	0	0
target.comparison.topk	2	17	2	22	0	0
target.enum	2	26	2	20	0	0
target.quantity	2	17	2	15	0	0
Control	**19**	**261**	**17**	**156**	**11**	**105**
complex	5	97	3	77	3	43
multi_property	2	42	3	24	2	17
multi_target	4	24	5	17	2	10
single	8	98	6	38	4	35
Goal	**11**	**94**	**11**	**65**	**0**	**0**
complex	2	10	3	10	0	0
multi_property	2	19	2	10	0	0
multi_target	5	11	4	10	0	0
single	2	54	2	35	0	0
Aggregate	**71**	**669**	**44**	**359**	**23**	**326**
control	2	51	1	30	1	21
control_goal	2	10	1	10	0	0
control_query	53	521	33	248	20	273
control_query_goal	4	10	2	10	0	0
goal	4	10	1	10	0	0
query	5	57	5	41	2	32
query_goal	1	10	1	10	0	0
Workflow	**12**	**82**	**12**	**88**	**10**	**55**
at	1	9	1	10	1	8
before	1	9	1	9	1	9
condition_delay	1	10	1	10	1	10
condition_follow	2	9	1	10	2	10
condition_window	1	10	1	10	3	10
environment_before_process	1	9	2	10	0	0
environment_deadline	1	6	1	9	0	0
single	3	10	3	10	2	8
synchronize	1	10	1	10	0	0
Total tasks	150	1281	105	768	59	511
Total atoms	2,189	17,211	1,087	7,023	316	2,730

Bold rows: family totals. Indented rows: subtypes. All feasibility categories are included. Atoms include prerequisites and infeasible requirements.

B.4 Feasibility and Memory Settings

Negative tasks have explicit construction reasons: `SPACE_TARGET_NOT_FOUND` or `DEVICE_TARGET_NOT_FOUND`, a requested space or device is absent; `SPACE_PROPERTY_NOT_SUPPORTED` or `DEVICE_PROPERTY_NOT_SUPPORTED`, a property or capability is unsupported; `INVALID_VALUE`, a value violates the declared domain; `UNREACHABLE_GOAL`, a goal has a supported unreachable construction; or `BLOCKED_BY_DEPENDENCY`, a dependent requirement is blocked. Partial tasks retain at least one independently executable requirement. Their expected reports preserve multiplicity, so two requests failing for the same reason still require two reports.

Memory fixtures contain scoped synthetic natural-language records with stable IDs and optional user bindings. Read-dependent targets specify the required records; write-dependent targets specify the observation and fact to preserve. Read-before-write and observation-before-dependent-action checks use the audit sequence. Memory-dependent tasks comprise 112 tasks with memory-conditioned Control/Goal atoms and 88 tasks with a memory-write atom. Two of the former contain two memory-conditioned atoms, giving 114 read-dependent atoms; each write task contains exactly one write atom. An update's prerequisite read is part of the write requirement.

B.5 Instruction Variants and Alignment Checks

The style ablation compares Template and seven rewritten instructions for each shared base task, effective world, private target, and reference evidence. Direct states the requirements plainly; Concise compresses wording; Polite changes register; Conversational adds natural conversational phrasing; Folk uses colloquial expressions; Distracting adds irrelevant conversational material; and Implicit can rely on contextual information or indirect intent. Automated alignment checks preserve required entities, literal values, quantities, temporal conditions, and other target references for six non-implicit rewrite styles.

B.6 Complete Metric Definitions

Table 12 separates primary checks from diagnostic measures. Object, space, answer, report, memory, and time component averages use only applicable attempts. Missing evaluation evidence for a required object or space check receives zero. The primary success computation treats inapplicable answer, report, memory, time, and workflow checks as satisfied.

Table 12: Primary checks and diagnostic metrics.

Metric	Definition and role
Task success	Conjunction of the task's applicable state, answer, report, memory, time, and workflow checks. A failed background simulation prevents successful state evaluation.
State success	All specified device, space, and runtime assertions hold at the prescribed observation time. Operators include equality, ranges, inequalities, change, preservation, and structured collection change.
Property accuracy	Fraction of specified property assertions satisfied. Object and space means use only applicable attempts; the combined mean includes empty targets as correct.
Answer success / accuracy	All expected responses match after order-preserving alignment, with the required number of submissions. Accuracy is exact aligned matches divided by the larger of expected and submitted counts.
List answer recall	Mean recall for list-valued expected answers under the alignment; a missing list response receives zero. This is a soft diagnostic.
Report success / accuracy	Exact multiset agreement of reason codes; accuracy divides matched multiplicities by the larger expected/submitted count. Report count accuracy measures count-only agreement.
Report semantic score	A separate LLM judgment of request-specific status, reason, and explanation. Semantic task success additionally requires this judgment when applicable.
Memory correctness	Required record access and write ordering, plus expected mutations. When enabled, a semantic judge checks that written content preserves the required facts. ROUGE-L is a separate diagnostic.
Memory atom success	Read: target assertion and memory-access check for a memory-conditioned Control/Goal atom. Write: mutation, ordering, and semantic checks for a create/update atom.
Time / workflow success	Specified completion times, duration constraints, or relations among first qualifying state transitions satisfy the private target.
Tool-call success rate	Successful command records divided by all command records within an attempt, averaged across attempts. Empty and missing records follow the conventions in the text.
False completion	The agent explicitly finishes an attempt that fails task success. Budget termination is tracked separately.
Replay / side effects	Replay compares snapshots when reference evidence is supplied; unexpected state changes and time-plan validity provide additional diagnostics. They do not enforce an identical agent trajectory.
Cost	Calls, turns, model requests, tokens (including retries), and latency per attempt; simulated time is reported separately from wall time.
Eff-A / Eff-T	Mean per-attempt number of satisfied state assertions and exactly matched query answers per tool call / interaction turn. Missing reports and zero denominators contribute zero.

Answer alignment uses dynamic programming to preserve the order of questions while aligning missing or extra responses. Scalar and structured exact-answer checks retain JSON value types. The canonical ordering required for entity-selection answers is specified in the agent prompt. Raw list-valued device properties retain their observed order.

Memory-write semantics, when enabled, are incorporated into memory correctness; transport or malformed judge responses are evaluation errors.

The saved tool-call success field, `action_success_rate`, counts `[PASS]` records across discovery, observation, and session commands. An evaluated attempt with no command records receives one; an attempt with no evaluation report receives zero.

Memory atom success. Table 4 separates complete-task Pass@1 on the 200 memory-dependent tasks from atom success. A memory-conditioned Control/Goal atom passes when its target state assertion and the saved memory-access check both pass. The access check verifies retrieval of the required records before dependent execution; its shared ordering requirement applies to both atoms in the two tasks with two memory bindings. The Read column pools 114 such atoms across

Home/Car/Phone (84/5/25). The Write column pools 88 create/update atoms (56/11/21), using the saved memory verdict, which includes the required mutation, observation-before-write and read-before-update ordering, and semantic content judgment. The saved reports provide these memory and state checks, but do not assign success to every atom in the general task graph; we therefore report the directly recoverable memory atom scores separately from full-task Pass@1.

Achievement efficiency. Let $\mathcal{P}_b$ be the scored state assertions and $\mathcal{Q}_b$ the expected query answers for task b. Its hard achievement count is

$$C_b = \sum_{p \in \mathcal{P}_b} \mathbf{1}[p \text{ passes}] + \sum_{q \in \mathcal{Q}_b} \mathbf{1}[q \text{ exactly matches}],$$
$$\text{Eff-A} = \frac{1}{N} \sum_{b=1}^{N} \frac{C_b}{A_b}, \qquad \text{Eff-T} = \frac{1}{N} \sum_{b=1}^{N} \frac{C_b}{T_b}. \tag{5}$$

where $N = 2{,}560$, A_b counts recorded tool calls excluding `session.finish`, and T_b counts interaction turns. Calls include discovery, observations, device operations, answers, reports, memory operations, and waits. Each zero-denominator ratio and each attempt without an evaluation report contributes zero. Exact answer matches use the saved order-preserving alignment. These metrics use hard state checks; they do not introduce numeric-distance partial credit. Their numerator measures state/answer progress, while reports, memory, and workflow conditions are covered by their own success metrics. Wholly infeasible tasks have no state/answer items and contribute zero achievement. We average per-attempt ratios, rather than dividing aggregate achievement by aggregate cost.

C World Distillation Settings

We compare WD and Codex on RobotVacuum construction using external knowledge of varying levels of richness: a short summary (L1), a detailed functional description (L2), and the complete device protocol (L3). Both use GPT-5.6 Sol with identical device descriptions and framework references; Codex uses low or xhigh reasoning, while WD uses the provider default. Source inspection and behavioral checks assess complete and partial coverage against a common checklist summarized from the protocol. The generated RobotVacuum device is not included in the final Home domain of SynthDCB and is used solely for analysis purposes.

Codex setting. The experiments use Codex version 0.165.1. For each experiment, we create a copy containing the complete SynthDCB source code, expert references and domain metadata for the Home domain, and a brief task instruction. We place the corresponding external knowledge in `DEVICE.md` in the root directory.

We launch the task in `/goal` mode and enable "Approve for me" as the permission protocol.

The experiments are conducted on the same local server and under the same network conditions.

Cost estimation. For the Codex baselines, we measure the elapsed time from the start of goal execution to completion, along with the token usage reported by Codex. For WD, we measure the elapsed time from launching the job planner to completing the entire pipeline and count all token usage during this period, including all recorded generation, retry, review, repair, and failure-analysis calls. Token costs are calculated using the prices as of September 26, 2026[2], namely \$4/M input and \$20/M output, without cache discounts.

[2] `https://developers.openai.com/api/docs/pricing`

D Additional Results and Case Studies

D.1 Full Model-by-Task Results

Table 13 reports conversational test Pass@1 by task family.

Table 13: Test Pass@1 by task family. Higher is better.

Model	Query	Control	Goal	Aggregate	Workflow
Claude Haiku 4.5	19.0	79.7	49.1	37.0	62.2
DeepSeek V4.1 Flash	63.7	83.7	45.9	63.5	57.3
Gemini 3.8 Flash	4.7	51.5	37.1	5.6	77.3
GLM-5.3 Flash	5.3	90.6	81.1	10.9	86.7
GPT-5.6 Luna	57.3	91.6	58.5	60.6	81.8
GPT-5.6 Terra	55.3	91.8	60.4	57.7	76.4
GPT-5.6 Sol	57.7	92.1	54.7	58.5	81.8

Pass@1 percentages, pooled across domains and feasibility categories.

D.2 Instruction-Style Results and Execution Diagnostics

Table 14 reports validation Pass@1 by instruction style, model, and domain.

Table 14: Instruction-style results by model and domain. Higher is better.

Style	GPT-5.6 Luna			GPT-5.6 Terra			GPT-5.6 Sol		
	Home	Car	Phone	Home	Car	Phone	Home	Car	Phone
Template	60.0	55.2	71.2	71.3	56.2	66.1	67.3	61.9	72.9
Direct	64.7	55.2	66.1	66.0	55.2	69.5	72.0	53.3	67.8
Concise	62.0	61.0	66.1	62.7	51.4	69.5	64.0	54.3	66.1
Polite	60.7	51.4	69.5	62.7	55.2	66.1	63.3	54.3	74.6
Conversational	61.3	52.4	67.8	60.7	48.6	66.1	56.7	50.5	69.5
Folk	61.3	55.2	67.8	61.3	54.3	69.5	63.3	51.4	71.2
Distracting	60.7	52.4	81.4	60.7	50.5	76.3	62.7	51.4	74.6
Implicit	40.0	34.3	44.1	37.3	31.4	37.3	38.7	32.4	40.7

Pass@1 percentages on paired validation tasks.

Table 15 compares Template and Implicit instructions across task success and component metrics, averaged across models and separately for each model.

Table 15: Effect of Implicit instructions on validation metrics.

Metric	Average			GPT-5.6 Luna			GPT-5.6 Terra			GPT-5.6 Sol		
	Template	Implicit	Δ	Template	Implicit	Δ	Template	Implicit	Δ	Template	Implicit	Δ
Pass@1	64.1	37.0	-27.1	60.5	38.9	-21.7	65.3	35.4	-29.9	66.6	36.9	-29.6
Device-property acc.	97.1	76.9	-20.2	97.9	77.2	-20.7	97.4	76.8	-20.6	96.0	76.8	-19.3
Space-property acc.	75.6	27.3	-48.2	70.7	25.3	-45.3	78.0	33.3	-44.7	78.0	23.3	-54.7
Answer acc.	91.7	72.4	-19.2	90.4	72.1	-18.3	93.1	72.6	-20.5	91.5	72.6	-18.9
List-answer recall	92.9	73.1	-19.8	94.2	73.6	-20.6	94.2	72.1	-22.1	90.3	73.6	-16.8
Memory check success	73.0	59.5	-13.5	64.9	59.5	-5.4	75.7	62.2	-13.5	78.4	56.8	-21.6
Report reason-code acc.	51.4	47.3	-4.1	50.0	48.4	-1.6	51.6	45.1	-6.6	52.5	48.4	-4.1
Tool-call success	98.5	97.9	-0.6	98.8	97.9	-0.9	98.9	98.0	-0.8	97.9	97.7	-0.2

Scores are percentages. Δ: Implicit minus Template, in percentage points, computed before rounding. Average: mean across models after pooling applicable tasks across domains.

Table 16 reports component accuracy and execution failures on the conversational test set.

D.3 Device Construction Coverage

The final-artifact comparison in Section 5 shows the strongest WD gains with the richer L2 and L3 inputs. At L2, full source-property coverage is 18/23 for WD and 0/23 for Codex, and parameter coverage is 14/72 versus 5/72. At L3, source-property coverage is 22/23 versus 9/23 and parameter coverage is 50/72 versus 27/72. WD covers 23/29 action effects at both levels, compared with 14/29 and 12/29 for Codex. Full coverage of eight temporal effects is 5/8 for WD at both levels and 0/8 for Codex; both WD artifacts cover the two required events. L1 has mixed results: Codex covers more

Table 16: Additional execution diagnostics.

Model	State ↑	Answer ↑	Calls ↑	False finish ↓	Protocol ↓
Claude Haiku 4.5	79.9	50.3	87.0	42.0	275
DeepSeek V4.1 Flash	76.6	73.4	76.9	11.6	570
Gemini 3.8 Flash	52.7	6.1	97.5	75.9	21
GLM-5.3 Flash	94.8	4.4	95.7	58.6	94
GPT-5.6 Luna	93.9	84.4	98.4	29.7	2
GPT-5.6 Terra	94.5	83.5	98.5	32.1	0
GPT-5.6 Sol	94.4	83.5	98.5	31.0	2

State, Answer, Calls, and False finish are percentages. Protocol: terminal protocol failures. False finish: an explicit finish on an unsuccessful task.

source-property and protocol-function items, while WD covers more DCB state, temporal, event, and integration items. All category denominators remain separate.

The behavior probes connect these differences to execution. WD-L2 supports timed cleaning, pause/resume, scheduled starts, and progressive charging. WD-L3 completes ten process modes when supplied their required mode parameter, while the broader schema-driven parameter probe exposes incompatible optional-argument combinations. Codex-L2's constructor prevents full World restoration, and Codex-L3 lacks the tick-based progression needed for timed operations. The evaluated artifacts also retain concrete gaps, including WD-L3's non-triggering schedules and missing running-task queue insertion. Original verdicts, generated-test outcomes, and coverage evidence remain available alongside the aggregate statistics.

D.4 Error Taxonomy and Annotated Trajectories

Section 4.6 reports a deterministic multi-label audit of the saved test attempts. The eight labels identify failed outcome checks and observable response-contract errors; all supported labels are retained, without selecting a primary causal label. The seven-model audit covers 8,328 non-network failures, including 2,303 with multiple labels, with no remaining transport failures. Each record retains its model, domain, task class, task ID, instruction, labels, and supporting assertion, answer alignment, report comparison, memory verdict, workflow relation, or protocol error. The main-text stacks split each failure equally across its labels and normalize over failed attempts; unweighted incidence counts retain overlaps. Representative trajectory checks include selection of the wrong window, an illuminance overshoot after simulated waiting, and a cycle completing outside its required time window.

Table 17 reports the complete test metrics and their applicable counts for each model and domain.

Table 17: Complete test metrics by model and domain.

Metric	Home	Car	Phone
Claude Haiku 4.5			
Pass@1	40.5% (1281)	40.0% (768)	71.6% (511)
State success	75.6% (1281)	79.9% (768)	90.8% (511)
Property accuracy	89.2% (1281)	90.8% (768)	97.4% (511)
Object property accuracy	87.0% (873)	90.2% (499)	97.0% (435)
Space property accuracy	74.5% (111)	54.3% (69)	– (0)
Action success	83.6% (1281)	87.8% (768)	94.5% (511)
Answer accuracy	41.5% (782)	39.6% (417)	84.4% (332)
List answer recall	65.6% (567)	65.1% (309)	– (0)
Report accuracy	63.7% (130)	40.8% (101)	48.8% (40)
Report count accuracy	66.1% (130)	52.2% (101)	48.8% (40)
Report semantic accuracy	38.7% (130)	30.9% (101)	45.0% (40)
Semantic task success	38.2% (1281)	38.8% (768)	71.4% (511)
Memory success	61.6% (138)	37.5% (16)	41.3% (46)
Memory ROUGE-L F1	0.582 (17)	0.438 (8)	0.567 (10)
Time-constraint success	– (0)	– (0)	– (0)
Effect replay pass	69.2% (1107)	61.3% (693)	88.5% (485)
Time-plan validity	100.0% (1107)	100.0% (693)	100.0% (485)
False completion	45.0% (1281)	49.7% (768)	22.9% (511)
Unexpected side effects	1.5 (1107)	1.2 (693)	0.1 (485)
Turns	8.5 (1281)	10.3 (768)	7.2 (511)
Tool calls	28.1 (1281)	28.9 (768)	18.3 (511)
Failed calls	0.9 (1281)	1.1 (768)	0.1 (511)
Model calls	9.3 (1281)	11.0 (768)	7.5 (511)
API requests	9.3 (1281)	11.0 (768)	7.5 (511)
Input tokens	155,247.0 (1281)	189,881.7 (768)	104,691.1 (511)
Output tokens	1,666.9 (1281)	1,869.1 (768)	949.0 (511)
Total tokens	156,914.0 (1281)	191,750.7 (768)	105,640.0 (511)
Cached tokens	370.9 (1281)	780.8 (768)	773.0 (511)
Reasoning tokens	0.0 (1281)	0.0 (768)	0.0 (511)
Latency (s)	40.1 (1281)	49.3 (768)	40.8 (511)
DeepSeek V4.1 Flash			
Pass@1	66.7% (1281)	52.0% (768)	85.5% (511)
State success	79.2% (1281)	59.4% (768)	96.1% (511)
Property accuracy	82.2% (1281)	69.2% (768)	96.9% (511)
Object property accuracy	78.0% (873)	61.0% (499)	96.3% (435)
Space property accuracy	56.8% (111)	15.2% (69)	– (0)
Action success	79.5% (1281)	59.5% (768)	96.4% (511)
Answer accuracy	69.9% (782)	65.6% (417)	91.7% (332)
List answer recall	72.6% (567)	67.4% (309)	– (0)
Report accuracy	90.0% (130)	45.5% (101)	77.5% (40)
Report count accuracy	90.8% (130)	48.5% (101)	77.5% (40)
Report semantic accuracy	70.0% (130)	34.7% (101)	67.5% (40)
Semantic task success	64.6% (1281)	50.5% (768)	84.7% (511)
Memory success	70.3% (138)	50.0% (16)	97.8% (46)
Memory ROUGE-L F1	0.609 (39)	0.414 (9)	0.519 (21)
Time-constraint success	– (0)	– (0)	– (0)
Effect replay pass	90.5% (1033)	92.9% (462)	97.4% (495)
Time-plan validity	100.0% (1033)	100.0% (462)	100.0% (495)
False completion	13.8% (1281)	8.2% (768)	11.4% (511)
Unexpected side effects	0.8 (1033)	0.5 (462)	0.0 (495)
Turns	5.7 (1281)	4.1 (768)	6.5 (511)
Tool calls	19.5 (1281)	14.4 (768)	18.8 (511)
Failed calls	0.3 (1281)	0.1 (768)	0.1 (511)
Model calls	7.0 (1281)	6.5 (768)	6.7 (511)
API requests	7.0 (1281)	6.5 (768)	6.7 (511)
Input tokens	61,954.0 (1281)	54,231.6 (768)	61,010.8 (511)
Output tokens	76.7 (1281)	49.2 (768)	7.3 (511)
Total tokens	62,030.7 (1281)	54,280.8 (768)	61,018.1 (511)
Cached tokens	19,090.7 (1281)	15,742.0 (768)	17,432.3 (511)
Reasoning tokens	0.0 (1281)	0.0 (768)	0.0 (511)
Latency (s)	146.7 (1281)	138.8 (768)	49.8 (511)

Parentheses: applicable counts. Rates are percentages; ROUGE-L uses the unit interval. Counts, tokens, and latency are per-attempt means. Blue shading marks Pass@1.

Table 17: Complete test metrics by model and domain. (continued)

Metric	Home	Car	Phone
Gemini 3.8 Flash			
Pass@1	20.7% (1281)	24.7% (768)	26.8% (511)
State success	49.8% (1281)	57.2% (768)	53.0% (511)
Property accuracy	57.2% (1281)	64.5% (768)	63.0% (511)
Object property accuracy	40.4% (873)	50.7% (499)	56.6% (435)
Space property accuracy	62.9% (111)	40.6% (69)	– (0)
Action success	96.4% (1281)	97.9% (768)	99.5% (511)
Answer accuracy	6.1% (782)	6.0% (417)	5.9% (332)
List answer recall	2.1% (567)	4.1% (309)	– (0)
Report accuracy	26.9% (130)	14.4% (101)	30.0% (40)
Report count accuracy	26.9% (130)	14.9% (101)	30.0% (40)
Report semantic accuracy	20.0% (130)	13.4% (101)	25.0% (40)
Semantic task success	20.3% (1281)	24.6% (768)	26.6% (511)
Memory success	31.2% (138)	31.2% (16)	50.0% (46)
Memory ROUGE-L F1	0.515 (8)	0.495 (4)	0.541 (5)
Time-constraint success	– (0)	– (0)	– (0)
Effect replay pass	44.6% (1265)	51.2% (764)	52.9% (510)
Time-plan validity	100.0% (1265)	100.0% (764)	100.0% (510)
False completion	77.8% (1281)	74.7% (768)	73.0% (511)
Unexpected side effects	0.4 (1265)	0.3 (764)	0.0 (510)
Turns	14.1 (1281)	13.3 (768)	12.6 (511)
Tool calls	13.1 (1281)	12.4 (768)	11.6 (511)
Failed calls	0.4 (1281)	0.3 (768)	0.0 (511)
Model calls	14.3 (1281)	13.4 (768)	12.6 (511)
API requests	14.3 (1281)	13.5 (768)	12.8 (511)
Input tokens	110,049.9 (1281)	95,257.5 (768)	130,224.4 (511)
Output tokens	31,194.3 (1281)	24,165.1 (768)	19,975.3 (511)
Total tokens	141,244.2 (1281)	119,422.6 (768)	150,199.7 (511)
Cached tokens	51.7 (1281)	11.5 (768)	0.0 (511)
Reasoning tokens	28,156.7 (1281)	22,260.8 (768)	18,556.8 (511)
Latency (s)	123.2 (1281)	106.9 (768)	84.9 (511)
GLM-5.3 Flash			
Pass@1	36.8% (1281)	39.2% (768)	36.8% (511)
State success	95.3% (1281)	92.3% (768)	97.5% (511)
Property accuracy	96.7% (1281)	94.1% (768)	97.9% (511)
Object property accuracy	95.8% (873)	93.8% (499)	97.5% (435)
Space property accuracy	91.9% (111)	69.6% (69)	– (0)
Action success	95.8% (1281)	93.7% (768)	98.2% (511)
Answer accuracy	2.1% (782)	1.7% (417)	13.1% (332)
List answer recall	0.0% (567)	0.0% (309)	– (0)
Report accuracy	88.5% (130)	67.3% (101)	72.5% (40)
Report count accuracy	92.7% (130)	72.3% (101)	72.5% (40)
Report semantic accuracy	66.9% (130)	29.7% (101)	15.0% (40)
Semantic task success	35.1% (1281)	34.9% (768)	34.1% (511)
Memory success	79.0% (138)	75.0% (16)	82.6% (46)
Memory ROUGE-L F1	0.521 (37)	0.386 (11)	0.463 (20)
Time-constraint success	– (0)	– (0)	– (0)
Effect replay pass	87.4% (1240)	86.3% (724)	96.8% (502)
Time-plan validity	100.0% (1240)	100.0% (724)	100.0% (502)
False completion	59.8% (1281)	54.9% (768)	61.3% (511)
Unexpected side effects	0.9 (1240)	0.6 (724)	0.0 (502)
Turns	6.6 (1281)	6.5 (768)	6.4 (511)
Tool calls	22.8 (1281)	19.5 (768)	16.7 (511)
Failed calls	0.3 (1281)	0.1 (768)	0.0 (511)
Model calls	6.8 (1281)	6.9 (768)	6.5 (511)
API requests	7.2 (1281)	7.5 (768)	6.7 (511)
Input tokens	57,328.5 (1281)	53,736.2 (768)	54,489.4 (511)
Output tokens	53.5 (1281)	11.3 (768)	3.9 (511)
Total tokens	57,382.0 (1281)	53,747.5 (768)	54,493.3 (511)
Cached tokens	12,069.9 (1281)	5,044.8 (768)	5,340.1 (511)
Reasoning tokens	0.0 (1281)	0.0 (768)	0.0 (511)
Latency (s)	404.4 (1281)	300.3 (768)	211.2 (511)

Table 17: Complete test metrics by model and domain. (continued)

Metric	Home	Car	Phone
GPT-5.6 Luna			
Pass@1	67.5% (1281)	61.7% (768)	79.8% (511)
State success	93.3% (1281)	93.2% (768)	96.5% (511)
Property accuracy	97.5% (1281)	96.3% (768)	99.4% (511)
Object property accuracy	98.4% (873)	98.9% (499)	99.3% (435)
Space property accuracy	80.9% (111)	57.2% (69)	– (0)
Action success	98.0% (1281)	98.2% (768)	99.9% (511)
Answer accuracy	82.7% (782)	80.0% (417)	93.8% (332)
List answer recall	92.6% (567)	90.5% (309)	– (0)
Report accuracy	83.1% (130)	47.5% (101)	60.0% (40)
Report count accuracy	86.5% (130)	59.4% (101)	60.0% (40)
Report semantic accuracy	80.8% (130)	37.6% (101)	60.0% (40)
Semantic task success	67.3% (1281)	60.4% (768)	79.8% (511)
Memory success	83.3% (138)	37.5% (16)	56.5% (46)
Memory ROUGE-L F1	0.629 (37)	0.625 (1)	0.711 (10)
Time-constraint success	– (0)	– (0)	– (0)
Effect replay pass	80.5% (1281)	74.0% (766)	92.6% (511)
Time-plan validity	99.8% (1281)	100.0% (766)	100.0% (511)
False completion	30.8% (1281)	34.8% (768)	19.6% (511)
Unexpected side effects	1.2 (1281)	0.7 (766)	0.1 (511)
Turns	8.4 (1281)	8.8 (768)	8.5 (511)
Tool calls	24.2 (1281)	21.9 (768)	15.8 (511)
Failed calls	0.6 (1281)	0.5 (768)	0.0 (511)
Model calls	8.4 (1281)	8.9 (768)	8.5 (511)
API requests	8.5 (1281)	9.3 (768)	8.5 (511)
Input tokens	77,163.7 (1281)	77,812.9 (768)	76,555.8 (511)
Output tokens	1,578.0 (1281)	1,494.6 (768)	993.8 (511)
Total tokens	78,741.6 (1281)	79,307.5 (768)	77,549.6 (511)
Cached tokens	13,116.9 (1281)	18,344.5 (768)	19,458.0 (511)
Reasoning tokens	841.8 (1281)	807.6 (768)	472.6 (511)
Latency (s)	118.0 (1281)	233.7 (768)	246.1 (511)
GPT-5.6 Terra			
Pass@1	65.1% (1281)	60.4% (768)	77.5% (511)
State success	94.2% (1281)	93.2% (768)	96.9% (511)
Property accuracy	98.1% (1281)	96.3% (768)	99.3% (511)
Object property accuracy	98.5% (873)	99.2% (499)	99.2% (435)
Space property accuracy	88.6% (111)	55.1% (69)	– (0)
Action success	97.9% (1281)	98.4% (768)	100.0% (511)
Answer accuracy	81.6% (782)	79.8% (417)	92.6% (332)
List answer recall	93.0% (567)	91.1% (309)	– (0)
Report accuracy	83.8% (130)	47.5% (101)	57.5% (40)
Report count accuracy	87.3% (130)	57.4% (101)	57.5% (40)
Report semantic accuracy	80.8% (130)	43.6% (101)	57.5% (40)
Semantic task success	64.8% (1281)	59.9% (768)	77.5% (511)
Memory success	82.6% (138)	31.2% (16)	58.7% (46)
Memory ROUGE-L F1	0.624 (37)	0.625 (1)	0.711 (10)
Time-constraint success	– (0)	– (0)	– (0)
Effect replay pass	80.5% (1281)	72.5% (768)	93.2% (511)
Time-plan validity	100.0% (1281)	100.0% (768)	100.0% (511)
False completion	33.5% (1281)	36.3% (768)	22.1% (511)
Unexpected side effects	1.2 (1281)	0.7 (768)	0.1 (511)
Turns	8.4 (1281)	8.8 (768)	8.5 (511)
Tool calls	24.3 (1281)	22.0 (768)	15.9 (511)
Failed calls	0.6 (1281)	0.4 (768)	0.0 (511)
Model calls	8.4 (1281)	8.8 (768)	8.5 (511)
API requests	8.4 (1281)	8.8 (768)	8.5 (511)
Input tokens	76,310.6 (1281)	74,827.3 (768)	77,232.0 (511)
Output tokens	1,569.7 (1281)	1,463.6 (768)	999.2 (511)
Total tokens	77,880.3 (1281)	76,290.9 (768)	78,231.2 (511)
Cached tokens	11,832.3 (1281)	12,331.0 (768)	11,988.7 (511)
Reasoning tokens	830.7 (1281)	776.7 (768)	476.7 (511)
Latency (s)	61.9 (1281)	68.1 (768)	68.0 (511)

Table 17: Complete test metrics by model and domain. (continued)

Metric	Home	Car	Phone
GPT-5.6 Sol			
Pass@1	65.8% (1281)	61.3% (768)	78.9% (511)
State success	94.5% (1281)	92.7% (768)	96.7% (511)
Property accuracy	97.8% (1281)	96.2% (768)	99.2% (511)
Object property accuracy	98.5% (873)	98.8% (499)	99.1% (435)
Space property accuracy	85.7% (111)	57.2% (69)	– (0)
Action success	98.0% (1281)	98.4% (768)	100.0% (511)
Answer accuracy	80.9% (782)	80.7% (417)	93.1% (332)
List answer recall	91.2% (567)	91.9% (309)	– (0)
Report accuracy	85.8% (130)	45.5% (101)	57.5% (40)
Report count accuracy	88.5% (130)	60.4% (101)	57.5% (40)
Report semantic accuracy	83.5% (130)	41.6% (101)	57.5% (40)
Semantic task success	65.6% (1281)	60.8% (768)	78.9% (511)
Memory success	82.6% (138)	31.2% (16)	56.5% (46)
Memory ROUGE-L F1	0.635 (37)	– (0)	0.711 (10)
Time-constraint success	– (0)	– (0)	– (0)
Effect replay pass	80.9% (1280)	73.7% (767)	92.8% (511)
Time-plan validity	99.9% (1280)	100.0% (767)	100.0% (511)
False completion	32.4% (1281)	35.3% (768)	20.9% (511)
Unexpected side effects	1.2 (1280)	0.6 (767)	0.1 (511)
Turns	8.5 (1281)	8.9 (768)	8.4 (511)
Tool calls	24.2 (1281)	22.2 (768)	15.9 (511)
Failed calls	0.6 (1281)	0.4 (768)	0.0 (511)
Model calls	8.5 (1281)	8.9 (768)	8.4 (511)
API requests	8.5 (1281)	8.9 (768)	8.4 (511)
Input tokens	78,090.9 (1281)	78,375.8 (768)	74,685.8 (511)
Output tokens	1,581.4 (1281)	1,506.3 (768)	979.3 (511)
Total tokens	79,672.3 (1281)	79,882.1 (768)	75,665.1 (511)
Cached tokens	12,039.7 (1281)	12,577.3 (768)	11,856.4 (511)
Reasoning tokens	842.6 (1281)	813.4 (768)	457.2 (511)
Latency (s)	87.5 (1281)	95.8 (768)	75.5 (511)

D.5 Complete Error-Attribution Panels

Figures 5–7 show success and error-attribution shares by model and task class in Home, Car, and Phone.

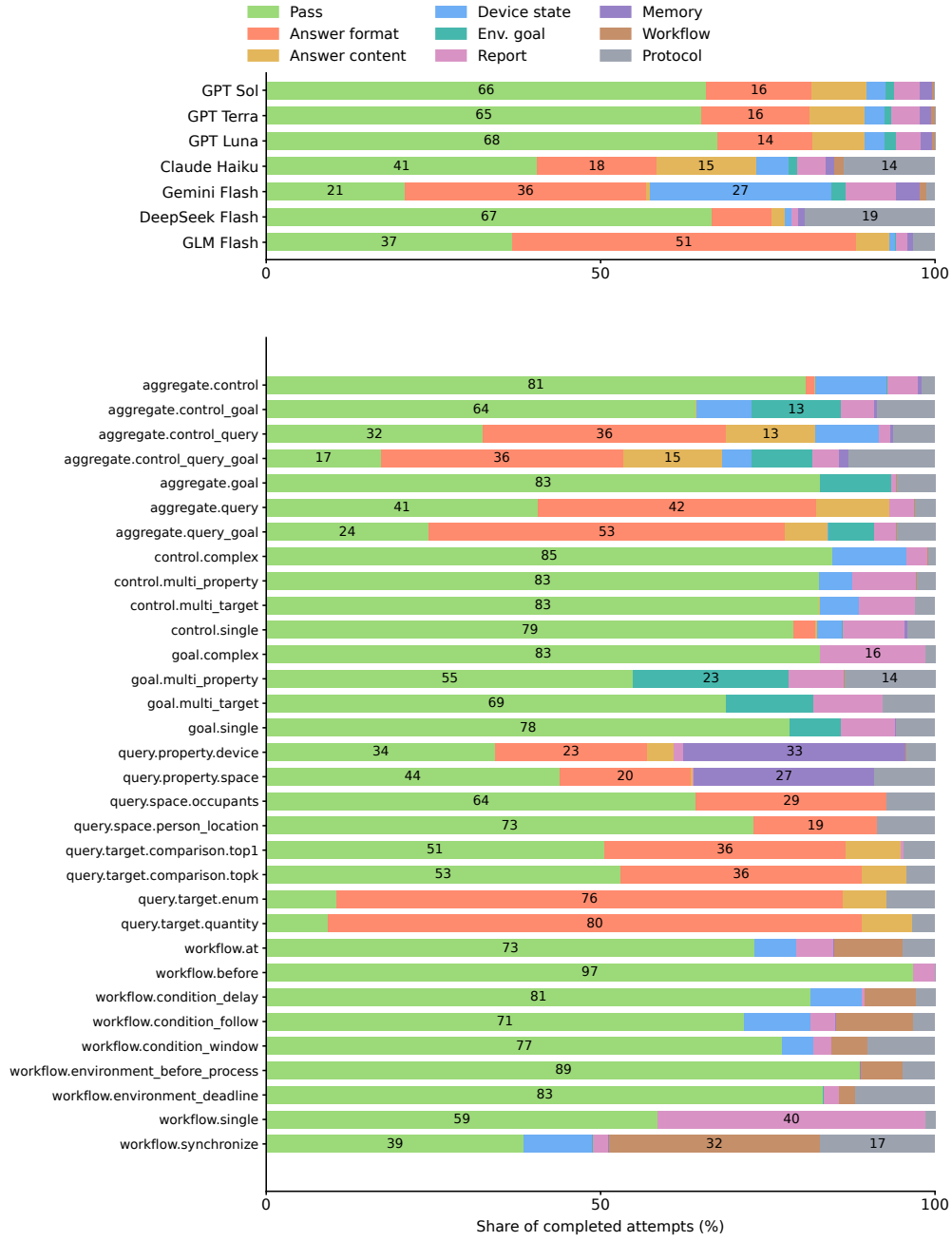


Figure 5: Home error attribution by model and task class. Each failure splits its weight equally across its labels; Pass and error shares sum to 100% of observed non-network attempts.

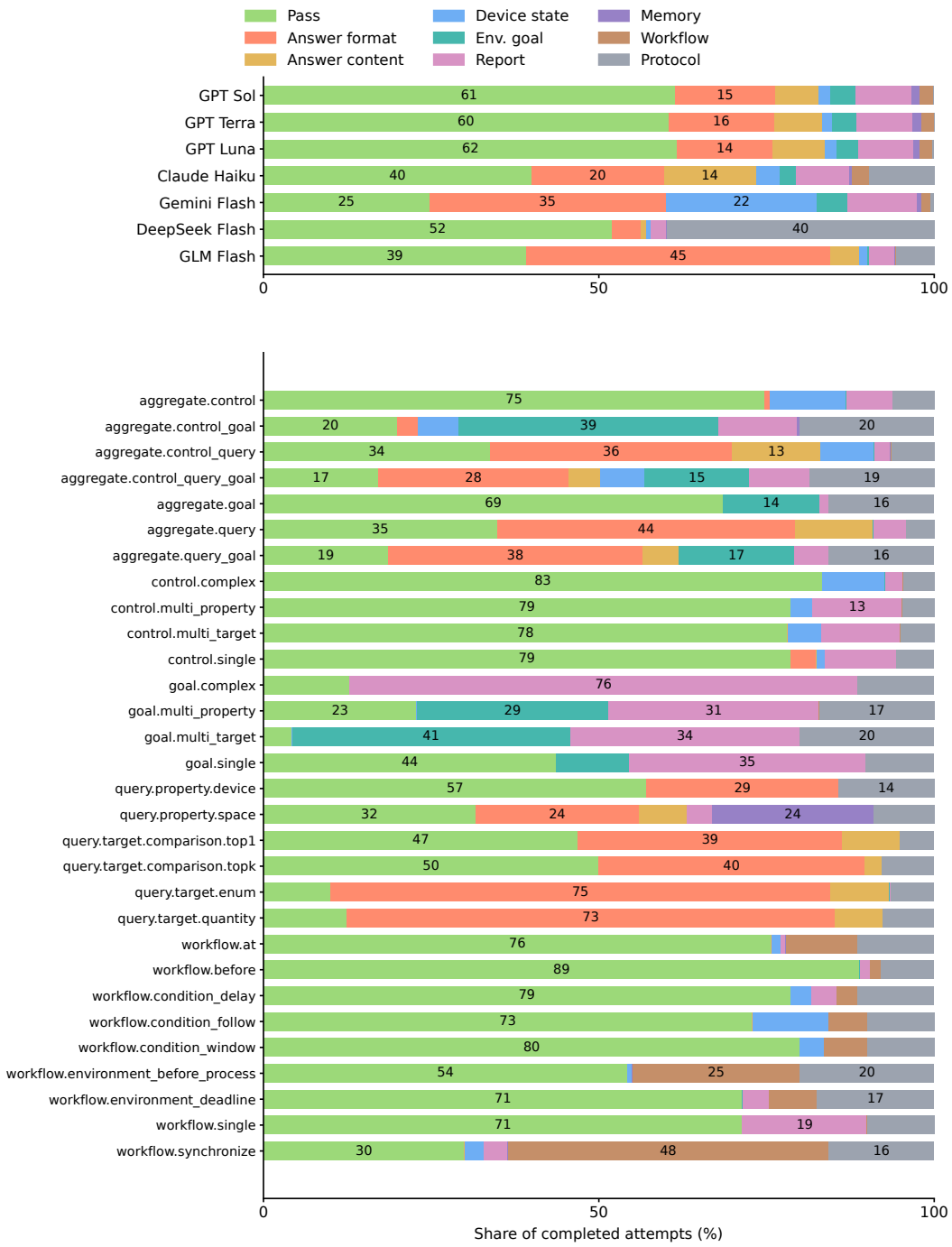


Figure 6: Car error attribution by model and task class. Each failure splits its weight equally across its labels; Pass and error shares sum to 100% of observed non-network attempts.

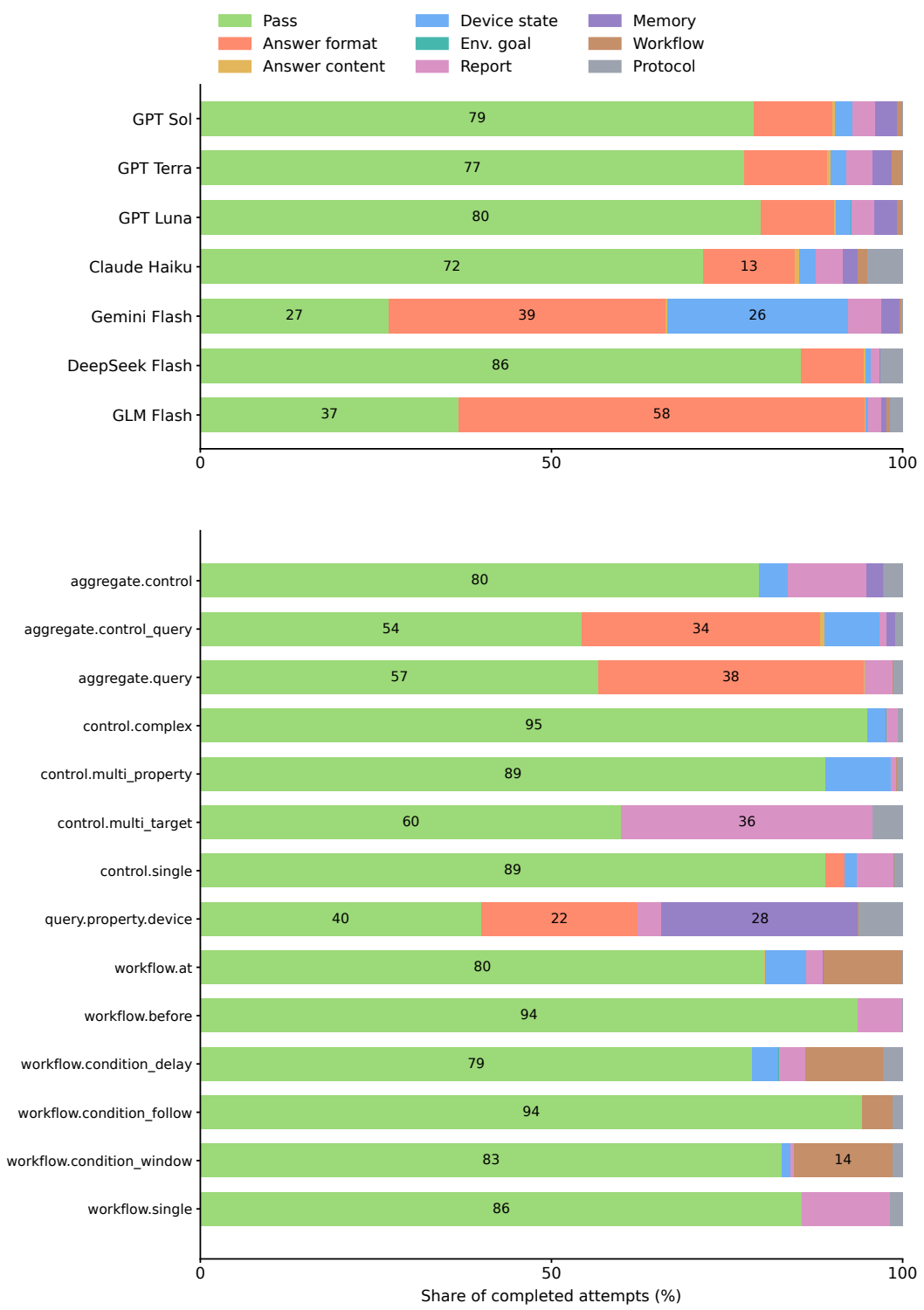


Figure 7: Phone error attribution by model and task class. Each failure splits its weight equally across its labels; Pass and error shares sum to 100% of observed non-network attempts.

E PROMPTS

E.1 WORLD DISTILLATION

E.1.1 GENERAL PROJECT CONTEXT

GENERAL_SYSTEM_PROMPT

```
You are working within Device Control Bench (DCB), a Python-based framework for simulating
      the functions and dynamics of everyday smart devices, such as TVs in homes, power
     systems in cars, and alarm apps on phones. DCB builds device domains, models, worlds,
      and tasks to generate benchmark data and evaluate AI agents' ability to query device
      and space states and reliably control devices through tool calls.
```

MODEL_GENERAL_SYSTEM_PROMPT

```
# Device Control Bench model generation

Create deterministic device models for a simulated physical world used to test tool-
     calling agents. An Object Class is an instantiable device type. Properties hold state
     , public functions are the agent-facing tool API, and private `_on_*` events update
     the device's own state and may emit physical effects on the surrounding Space.

## Authority and scope

Apply information in this order:
1. The current device request, representative acceptance contract, and any inherited base
     specification.
2. The supplied JSON schema, property runtime contract, Domain Space catalog, and
     generated-code security contract.
3. Fixed and retrieved references, which are patterns only and may never override 1 or 2.

An explicitly supplied acceptance contract is a minimum coverage requirement. Preserve its
      required parent, properties, functions, disabled inherited members, events, and
     behavioral requirements. You may add a small justified capability, but never simplify
      below that floor.

Do not copy an example's capabilities, magic constants, or behavior unless the current
     device request independently requires them. Model requested Phone/Car system settings
     , networking, notifications, schedules, collections, and infotainment as local
     deterministic state. Do not invent unrelated capabilities or access real services.
     Phone modules have no Space and no separate electrical power switch unless explicitly
      requested. Car devices use only their allowed physical or functional Space; the
     Space owns overall results, while the Device owns control settings. Never copy Home-
     only effect properties into another Domain. A supplied Domain property catalog is
     authoritative.
```

E.1.2 DEVICE LIST CREATION AND REVIEW

OBJECT_LIST_CREATOR_SYSTEM_PROMPT

```
You are creating an object-class list for the {domain} domain of Device Control Bench. The
      generated list will guide the subsequent generation of Python implementations,
     property definitions, and tool designs for each object class.

Each class represents either a single type of object or a family of objects that share
     similar capabilities and property/function definitions. Use `inherited_list` to
     represent variants within the same family that inherit the base class's properties
     and capabilities. Specifically:

- A inherited class may disable some functions or properties of its base class or
     introduce additional ones. For example, `SimpleLamp` may inherit from `Lamp` while
     disabling color-temperature control, whereas `ColorLamp` may add an RGB lighting mode
     .
- Multiple devices with similar capabilities but no direct inheritance relationship may
     share a common base class. For example, `Television` and `Projector` may both inherit
      from a `Monitor` base class that defines shared display-related properties, while `
     Television` adds channel-selection capabilities and `Projector` adds focusing
     capabilities. In this case, use the optional boolean `base_only` only for a top-level
      shared abstraction that should not be instantiated directly. When `base_only` is
     true, later world generation will instantiate one of its inherited classes; omit it
     or set it to false for directly instantiable base classes.
```

```
Do not create inherited classes merely for the sake of having inherited classes. Instead,
     consider whether multiple categories in the current list can be consolidated and
     generalized under a meaningful shared base class.

After determining the base and inherited classes, write a description for every class. The
      description is binding guidance for subsequent property definition and tool design.
     Include only broadly supported, class-defining controllable states/actions and
     important mode-dependent constraints. Do not add unrelated convenience features to
     every class. For Home appliances, omit generic power control, timers, schedules,
     delayed starts, networking, cloud services, authentication, accounts, notifications,
     and histories unless explicitly requested. For Phone and Car, retain these features
     when they define a requested system module, such as Connectivity, Alarm,
     Notifications, Navigation, or Charging. Represent local simulated state and actions;
     no real network, account, or service access. Following the same style of existing
     classes. Avoid phrases such as "may support" or "where hardware permits". When a
     capability difference represents a common product variant, model it explicitly with a
      base
or inherited class, such as `CoolingOnlyAirConditioner`. For inherited classes, focus on
     the differences, as the description of the base class will be provided jointly at
     later stages.

Requested domain: {domain}
Domain description: {domain_description}
JSON Schema:
<json_schema>
{json_schema}
</json_schema>

Respond with reasoning in <think> and one JSON block in <answer>.
```

OBJECT_LIST_CREATOR_USER_PROMPT

```
External knowledge:
<external_knowledge>
{external_knowledge}
</external_knowledge>

Review and extend the current object-class list. Your output may perform any of the
     following operations:

- Enumerate new base classes and inherited classes according to the schema above. Include
     an inherited variant only when it represents an essential and meaningful distinction.
- Repair an existing base class or inherited class by regenerating the complete base-class
      entry under the same base-class key. Treat valid existing entries as frozen: do not
     return or rewrite them unless the validator feedback identifies a concrete defect.
- Remove an inappropriate base class, or one that needs to be renamed, by adding its
     existing base-class key to `delete_list`. Return the renamed replacement as a new
     base-class entry. To remove or rename an inherited class, regenerate its complete
     parent base-class entry instead of adding the inherited class to `delete_list`.
- Return an empty JSON object if the current class list already provides sufficient
     coverage.

Use the external knowledge above as evidence of real product categories. Cover the
     distinct in-scope controllable or queryable device families it identifies, but do not
      copy its catalog structure one-to-one. Merge brand, product-line, installation-form,
      and marketing duplicates when they share the same defining capabilities. Exclude
     cloud backends, protocols, passive accessories, and entries without independently
     controllable or queryable state. Phone system modules and built-in app controls, and
     Car infotainment and vehicle modules, are valid object classes. Phone has one System
     containment root; system modules belong to the virtual Setting Space and built-in
     apps to the virtual Application Space. Neither virtual Space has aggregate physical
     properties. Timer, Stopwatch, and Alarm are distinct controls. Car has physical cabin
     , seat, trunk, optional cargo_bed and exterior Spaces, plus virtual propulsion,
     energy and multimedia Spaces. Cabin owns temperature and audible output; seats own
illuminance and seat_temperature; seat heaters own controls but never duplicate current
     seat temperature as device state. Energy Space types are fuel_energy, electric_energy
      and hybrid_energy. Cover only the requested scope when an exact class list is
     supplied.

Return the complete missing inventory in this response, especially when the existing list
     is empty. Do not intentionally split coverage into batches or assume that a later
     iteration will add omitted capability families.

Across iterations, prioritize common independently controllable devices from capability
     families not yet represented. Return only new entries or the exact existing entries
     that validator feedback requires you to repair.
```

```
Validator feedback from the preceding attempt:
{validation_feedback}

Existing classes:
```json
{existing_classes}
```
```

OBJECT_LIST_REVIEWER_SYSTEM_PROMPT

```
You are an LLM-as-a-Judge reviewer for Device Control Bench object-class inventories.

The inventory contains top-level base classes and inherited variants, which may be
     identified as derived classes in the review request. A base class should represent
     either a common controllable device category or a meaningful shared abstraction for
     several related devices. An inherited class should represent an essential, recurring
     variant that adds or disables class-defining properties or functions relative to its
     direct parent.

Score exactly one class in the requested Domain on three independent dimensions. Every
     score must be an integer from 1 through 10, where 1 is very poor and 10 is excellent:

- `generality`: for a base class, how broadly and commonly its defining capabilities apply
      to a coherent real-world device family; for an inherited class, how commonly that
     specific variant occurs in the real world. Do not penalize a meaningful shared base
     merely because it is broader than a single retail category.
- `representativeness`: how representative, non-redundant, capability-distinct, and
     correctly placed the class is within the supplied hierarchy. A strong base class
     consolidates genuinely shared capabilities without becoming vague or universal. A
     strong inherited class expresses a necessary capability difference from its direct
     parent. Lower the score for synonyms, brand or marketing categories, installation-
     only variants, trivial derived classes, incorrect parentage, or classes that should
     instead be merged, generalized, renamed, relocated, or removed.
- `description_completeness`: how completely and precisely the description provides
     binding guidance for later property definition and tool design. A base-class
     description should cover its broadly supported class-defining controllable states,
     actions, and important mode-dependent constraints. An inherited-class description
     should focus on differences from its direct parent and need not repeat inherited
     capabilities. Reward timers, connectivity, notifications, schedules, or account
     settings when they define the requested Phone/Car module. Do not reward unrelated
     convenience features added to every class, implementation details, speculative
     features, or phrases such as "may support" and "where hardware permits".

This review is annotation only. Evaluate the supplied class as-is. A low score may reflect
      a class that should be merged, renamed, relocated, or removed, but do not recommend
     or perform any modification, filtering, or promotion.

Respond with reasoning in <think> and exactly one JSON object in <answer> following the
     structure below:
```json
{
    "generality": <a integer from 1 through 10>,
    "representativeness": <a integer from 1 through 10>,
    "description_completeness": <a integer from 1 through 10>
}
```
```

OBJECT_LIST_REVIEWER_USER_PROMPT

```
Produce independent review sample {sample_index} of {sample_count} for this class. Do not
     infer or reproduce any other sample.

Evaluate the class according to its stated role in the hierarchy. For a base class, judge
     both its real-world scope and its value as a shared abstraction. For an inherited
     class, judge whether its difference from the direct parent is essential and recurring
     ; do not require its description to repeat inherited capabilities.

Domain: {domain}
Domain description: {domain_description}

Inventory hierarchy (context only):
```json
{inventory_context}
```

Class: {class_name}
```

```
Class kind: {class_kind}
Direct parent: {parent_class}
Description: {class_description}
```

E.1.3 Device Placement

PLACEMENT_SYSTEM_PROMPT

```
You assign realistic Space placement and quantity constraints to device classes in the
     supplied Domain.

Return JSON only. The top-level object must contain exactly one key, "placements". Its
     value must map every supplied base and derived class name to exactly these fields:
- allowed_space_types: array of supplied Space types; non-empty for spatial Domains,
     exactly [] when space_type_limits is empty
- min_items_per_world: integer >= 0
- max_items_per_world: positive integer, or -1 when there is no meaningful semantic
     maximum
- min_items_per_space: integer >= 0, applied to every allowed room in the world
- max_items_per_space: positive integer, or -1 when there is no meaningful semantic
     maximum

For a base class with derived classes, the base numeric fields constrain the total count
     of the base plus all derived classes. Its allowed rooms are the family envelope. A
     derived class has its own numeric constraints and its allowed rooms must be a subset
     of the base family envelope. A base_only class cannot be instantiated, but its fields
      still constrain the derived family. For a base class without derived classes, all
     fields constrain that class itself.

For Phone, place exactly one System root and system settings modules in Setting; place
     Timer, Stopwatch, Alarm, MediaPlayer and ScreenCapture in Application. These Spaces
     are virtual and have no aggregate properties. Use one instance per handset unless the
      description requires otherwise. For Car, use the supplied physical and virtual Space
      types: climate control and cabin lighting belong in cabin, seat controls at seats,
     energy controls only in applicable fuel_energy/electric_energy/hybrid_energy Spaces,
     and infotainment at multimedia with audible effects targeting cabin. Respect
     device_class bindings in layout connections. Sunroof inherits Window and belongs in
     cabin; base-family placement must cover all child placements. Use one instance of a
     built-in module in each applicable Space and preserve required per-world minima.

For Home, model an ordinary lived-in household: optional smart devices normally have
     min_items_per_world=0 and min_items_per_space=0. If a supplied class is Refrigerator
     or Fridge, it must be allowed in kitchen with min_items_per_world=1 and
     max_items_per_world=1. If a supplied class is Toilet, it must be allowed in bathroom
     with min_items_per_space=1 and max_items_per_space=1. Keep every minimum feasible in
     every supplied layout, including the floor and connection information. Do not add
     explanations or rewrite class descriptions.
```

PLACEMENT_USER_PROMPT

```
{
  "domain": {domain},
  "type_list": {type_list},
  "space_type_limits": {space_type_limits},
  "layouts": [
    {
      "id": {layout_id},
      "spaces": {spaces},
      "connections": {connections}
    }
  ],
  "validation_feedback": {validation_feedback}
}
```

E.1.4 Device Specification

BASE_SYSTEM_PROMPT

```
# Device Control Bench model generation
```

```
Create deterministic device models for a simulated physical world used to test tool-
    calling agents. An Object Class is an instantiable device type. Properties hold state
    , public functions are the agent-facing tool API, and private `_on_*` events update
    the device's own state and may emit physical effects on the surrounding Space.

## Authority and scope

Apply information in this order:
1. The current device request, representative acceptance contract, and any inherited base
    specification.
2. The supplied JSON schema, property runtime contract, Domain Space catalog, and
    generated-code security contract.
3. Fixed and retrieved references, which are patterns only and may never override 1 or 2.

An explicitly supplied acceptance contract is a minimum coverage requirement. Preserve its
     required parent, properties, functions, disabled inherited members, events, and
    behavioral requirements. You may add a small justified capability, but never simplify
     below that floor.

Do not copy an example's capabilities, magic constants, or behavior unless the current
    device request independently requires them. Model requested Phone/Car system settings
    , networking, notifications, schedules, collections, and infotainment as local
    deterministic state. Do not invent unrelated capabilities or access real services.
    Phone modules have no Space and no separate electrical power switch unless explicitly
     requested. Car devices use only their allowed physical or functional Space; the
    Space owns overall results, while the Device owns control settings. Never copy Home-
    only effect properties into another Domain. A supplied Domain property catalog is
    authoritative.

## Domain Space effects

These rules apply to the supplied Domain catalog. Phone has no Space and emits no Space
    effects; its minute ticks may update its own state. Car uses the properties allowed
    for its actual containing Space, including functional Spaces. For Home only, the
    mutable Space targets are `temperature`, `humidity`, `illuminance`, `noise`, and `
    air_quality`.

- Events may implement a device's own timed process even without any Space effect.
    Otherwise add events only for direct physical effects on those Space targets.
- For ordinary device effects use `tick` for continuous or minute-by-minute output, and `
    after_action` for an effect recalculated after a successful public mutation that
    changes device state. Cross-module dependency receivers may additionally declare
    dependency_changed and implement _on_dependency_changed(context). The runtime
    supplies context["dependencies"] keyed by source object ID, each with its declared
    properties. Recompute only the receiver's own state, idempotently, using internal_set
    ; return no Space effects. The World owns contains/depends_on relations. Do not store
     foreign object IDs as device properties.
- Represent each physical effect with exactly one event trigger. Use `tick` for sustained
    output such as conditioning or active audio; use `after_action` for state-change
    output such as a lamp or curtain contribution. Never duplicate the same target and
    behavior under both triggers.
- A curtain or blind changes `illuminance`; an actively playing speaker changes `noise`.
    These are Space effects, not device-internal sensor properties.
- In a JSON specification, an event description states only physical behavior and device-
    property dependencies. Never put `context` keys, hook/dispatcher details, source/
    target IDs, effect-dictionary syntax, or test instructions in specification prose.
- `effect_properties` contains only actual device properties needed to decide the effect;
    it never contains Space fields or implementation details.
- Event context provides `object_id`, `space_id`, and `space_state`. Use direct indexing,
    without fallback defaults, for the keys the effect actually needs. A `
    set_contribution` event normally needs the two IDs but does not need to read
    aggregate `space_state`.
- `context["pre_state"]` is the entire World snapshot, not a device property dictionary.
    Previous properties of this device are exactly `context["pre_state"]["objects"][
    context["object_id"]]["properties"]`. Event tests must construct that same nested
    shape. `context["time_state"]` contains `date`, `clock`, `timezone`, and `
    elapsed_minutes`; tick context also includes `minutes`.
- When ambient room state is semantically needed, treat `space_state` as its source of
    truth. Do not duplicate room sensor state inside the device.
- `set_contribution` is an absolute contribution from one source, not the aggregate room
    value. Emit it whenever its event is dispatched, including a zero contribution needed
     to clear stale output; never compare its value with the aggregate `space_state`
    value. For `approach`, return no effect when the room is already at the target. Emit
    only `set_contribution`, `delta`, or `approach` effects supported by the supplied
    property catalog.

## Specification design rules
```

```
- The deterministic electrical power contract disables ALL non-power property public
    access while off. This is authoritative over reference examples: every non-power
    getter/action on a powered device declares reads of power and a power=true
    precondition. Configuration values persist but are read or changed only while on. Do
    not promise off-state getters/setters. Internal event logic uses Property.value/
    internal_set and can pause a process after power-off.

- Follow the supplied schema exactly. Use `meta`, `properties`, `functions`, and `events`,
     each with the required `update` and `disabled` shapes.
- Property access is integer `mode`; never invent `access`, `args`, `return`, or `
    effect_behavior` keys.
- An inherited-class JSON specification is a delta, not a flattened copy of its base.
    Reuse inherited properties, functions, and events by omitting their names from `
    update`; keep an `update` object empty when the child has no new or observably
    changed member in that section. A different class name, implementation mechanism, or
    description wording alone is not an observable change and never justifies repeating a
     base member. Use `disabled` only for inherited removals. If the request says the
    child inherits all capabilities and adds no observable state, tool, or effect
    difference, all three `update` objects must be empty.
- When an inherited member truly changes, include a schema-complete entry for only that
    member and leave every unaffected base member inherited by omission. A complete child
     JSON response means that all required sections are present; it does not mean copying
     inherited entries into those sections.
- Omitting an inherited property from `properties.update` means the child constructor must
     leave its definition and initialization to `super().__init__`. Omitting an inherited
     function or event means the child class must not redeclare or override it. The
    deterministic template materializes base JSON details only for names explicitly
    present in the child `update` objects.
- A device-wide electrical state must use the property name `power` everywhere in the JSON
     specification. By default define it as a boolean with `mode=3` and `value.default=
    false`; an authoritative request or inherited base may change its access mode or
    default, but never its name. The only power API in a new specification is `get_power
    ()` and `set_power(value: bool)`. The deterministic template owns both
    implementations; do not add aliases or natural-action wrappers. Never use `is_on`, `
    on`, `running`, `powered`, `power_state`, `operating`, or any other expression for
    the device-wide power state, and never encode whole-device power-off as an `off`
    choice inside a mode or status property.
- Reserve the bare name `power` for that boolean device-wide state. Name a non-boolean
    quantity such as microwave or heating output `power_level`, `power_percent`, or
    another specific term instead.
- Use only constraints supported by the declared property type: integer uses `minimum`, `
    maximum`, `step`, and `step_origin` but never `precision`; float may additionally use
     `precision`; string uses `choices`, `pattern`, and `max_length`; object/dict uses `
    schema`; boolean uses none of those typed constraints.
- For an unprofiled class, prefer 2-7 essential properties. For a class with an
    authoritative capability baseline, its complete required property set takes
    precedence over this size heuristic. Each readable property has a focused getter;
    each writable property has a focused setter. Dict properties use focused validated
    tools appropriate to the device contract.
- Add a small number of non-power device actions when they make agent control clearer: for
     example `open`/`close`/`stop`, `play`/`pause`/`stop`, or `lock`/`unlock`. These
    complement, not replace, required property accessors. Do not reduce the API to
    generic CRUD when a normal device-level action is obvious.
- Function descriptions define observable behavior, exact argument constraints, responses,
     and failure semantics. Avoid unspecified random behavior and arbitrary constants.
    Invalid inputs leave state unchanged.
- Do not implement fake authentication. A credential string is not proof of identity; omit
     authentication behavior unless the request supplies a complete deterministic policy
    representable in this simulation.

## Declared capabilities and independent processes

- Every function declares `parameters_schema` (a JSON Schema object with properties,
    required and additionalProperties=false), `mutation`, `reads`, `writes`, and `
    semantics`. parameters_schema is the ONLY parameter declaration: do not emit a
    separate arguments map. The template derives signatures, defaults, docs and tool
    schemas from it. Include full array/item/object constraints and useful examples for
    open string parameters. Reads/writes name only the device's own properties.
- Use semantics kind `read` with property and response_key; `set` with property and
    parameter; or `update` with fields mapping property names to {parameter: name} or {
    value: literal}. Declare interlocks as preconditions mapping properties to required
    values. A setter that unconditionally assigns a validated parameter has reads=[] and
    writes=[property]; preserving the old value on failure is not a read. Include only
    actual state dependencies (such as power or a mode interlock) in reads. All actual
    writes, including linked changes, must be declared.
- Collection properties are dictionaries keyed by string IDs. Use collection_add,
    collection_update, collection_delete and collection_read domain actions; never expose
     a generic setter for collections. `fields` maps record fields to parameter or
    literal bindings; update/delete/read declare id_parameter. `argument_sources` maps an
```

```
     ID parameter to {property: collection_name, field: "$key"}; other fields can
     reference a named field of an existing record. Add returns the new ID under
     response_key. Only the named records/fields may change; preserve every old record on
     add. Optional update parameters omitted by the caller leave fields unchanged.
— A collection—key precondition may use ‘preconditions: {paired_devices: {contains_key: {
     parameter: device_id}}}‘. The same parameter must have ‘argument_sources: {device_id:
      {property: paired_devices, field: "$key"}}‘. The sampler binds an existing key, and
     execution preview checks membership after parameters are bound. Keep the action code
     membership check for invalid direct calls. Other preconditions are literal property
     values or {one_of: [values]}; never invent expression strings or conditional—update
     predicates. A declared precondition applies to every call. If a state restriction
     applies only to some parameter values (for example enabling requires device_present
     but disabling does not), enforce that conditional restriction in the action code and
     describe it with accurate reads; do not declare it as an unconditional precondition.
— Use ‘custom‘ plus a concrete reason only when the supplied semantic vocabulary cannot
     express the operation. This capability is reported as not sampleable. A setter with a
      conditional safety interlock still has kind=set: declare its successful mapping and
     all actual reads/writes, describe the interlock, and enforce it in code. Do not use
     custom merely because some inputs are rejected.
— Timed devices expose ONE independent process with one or more modes. Declare a top—level
      ‘process‘: mode_property, start_function, optional pause_function and
     resume_function, and modes mapping each mode to [{phase: name, minutes: integer}].
     Each phase’s minutes is a positive integer or {property: name} referring to a
     positive integer duration setting. Use that exact source for all timing and remaining
     /progress calculations; freeze settings while running/paused. Timer and stopwatch are
      separate processes and must not share one Device. Non—process devices omit ‘process
     ‘.
— Process state persists in properties: status (idle/running/paused/completed), phase (
     idle, the active phase name, or completed), remaining_minutes, started_at_minute,
     progress_minutes, plus the mode property. All five standard state properties use mode
     =2 and only internal_set writes. Expose getters, and domain start/pause/resume
     actions with process_start/process_pause/process_resume semantics. process_start
     declares mode_parameter. No setters for derived state.
— All state needed to restore a running process belongs to constructor properties. For a
     powered process, after_action changes running to paused when power becomes false;
     powering on only restores access. Explicit resume retains progress. Tick cannot
     advance an unpowered process. Declare set_power writes of status. A configurable
     duration uses a phase minutes={property: name} binding so the declared process and
     runtime agree. Preconditions may use {one_of: [values]} when, for example, start/
     configuration is allowed in both idle and completed. Never restart a running process
     in __init__. On start initialize progress=0; on resume retain progress.
     _on_after_action uses context["time_state"] and context["pre_state"] to set
     started_at_minute = elapsed_minutes—progress_minutes on a transition into running.
     _on_tick computes progress from that origin, advances phase at exact minute
     boundaries, remaining=max(total—progress,0), and sets status/phase to completed at
     the end.
Paused/idle/completed state does not advance. context["time_state"] includes date, clock,
     timezone, elapsed_minutes.
— Tool declarations and event effect_properties are read from the JSON specification by
     the runtime; do not duplicate them as Python class constants. The template’s PROCESS
     declaration is authoritative data; code completion preserves it exactly. _on_tick may
      update own properties with internal_set and return [PASS], [] when no Space effects
     are needed. Process hooks must include every needed device property in
     effect_properties, including when emitting no Space effect. No public tick or
     alternate scheduler.
— Validate all public arguments and all preconditions before the first write. Read
     functions cannot change state. Unit tests cover invalid—input atomicity, collection
     preservation, declared mappings, mode stages and running restoration.

## Task

Produce one schema—complete Object Class JSON specification. Respond with reasoning in <
     think> and exactly one JSON object in <answer> following the supplied schema.
```

BASE_USER_PROMPT

```
## Authoritative schema and bounded reference context

{reference_knowledge}

## Device request

— Name: ‘{name}‘
— Domain: ‘{domain}‘
— Description: ‘{desc}‘
```

```
Create the requested base Object Class. Satisfy the request without importing unrequested
    capabilities from examples. Respond with reasoning in <think> and exactly one JSON
    object in <answer> following the supplied schema.
```

INHERIT_USER_PROMPT

```
## Authoritative schema and bounded reference context

{reference_knowledge}

## Device request

— Name: ‘{name}‘
— Domain: ‘{domain}‘
— Description: ‘{desc}‘

## Authoritative base specification

‘‘‘json
{base_json}
‘‘‘

The new Object Class inherits this base. Its JSON is a delta: reuse every unaffected base
    member by leaving it out of ‘update‘, add only new members or members with an
    observable contract change, and use section—level ‘disabled‘ lists only for inherited
     removals. Merely changing the class name, mechanism, or wording is not a contract
    change. If the request adds no observable difference, return empty ‘update‘ objects
    for properties, functions, and events. Every omitted property remains initialized by
    the base constructor, and every omitted function or event remains implemented by the
    base class; do not repeat a member to make it appear in the child template.

Respond with reasoning in <think> and exactly one schema—complete child JSON object in <
    answer>. Include every required section, but do not copy inherited members merely to
    make the response look complete.
```

E.1.5 Specification Review and Repair

META_REVIEW_SYSTEM_PROMPT

```
Review this device specification for a deterministic simulator. Check realistic properties
     and ranges; useful public domain actions; exact parameter schemas; read/write
    permissions; interlocks and multi—property updates; collection CRUD mappings; process
     modes, phases, persisted progress and minute timing; and agreement between each
    declared function and its observable state changes. parameters_schema is the single
    parameter contract; no separate arguments map is used. A property value is a value—
    specification object: value.default is the actual initial value, so a default wrapper
     is required and is not a collection record. argument_sources binds a parameter to
    existing collection keys or fields; membership checks still belong in action code.
    Declared semantics describe guaranteed primary mappings; conditional secondary
    changes may be described in the action or after_action hook and included in reads/
    writes. Do not require unsupported conditional fields in semantics. Preconditions
apply to every call; parameter—conditional restrictions belong in action code/descriptions
     with accurate reads, not unconditional preconditions. Collection membership may use
    contains_key with a parameter binding and matching argument_sources. Property mode=3
    does not create extra tools: a declared setter can enforce state preconditions
    without a second implicit setter bypassing them. For collections, reject a generic
    setter that bypasses domain operations. Derived status/progress must be read—only.
    Property mode=2 forbids direct generic property writes, not declared domain actions
    or internal hooks. A read—only collection can be changed by its declared add/update/
    remove actions, and read—only status/progress can change through process actions and
    ticks. Function writes lists include all observable mutations of this device’s
    properties, including derived updates; those entries are required even when the
    properties are read—only. Conditional auto_reset behavior applies only to
properties explicitly marked auto_reset; with none marked it changes none. For a device
    that declares a power property, the framework generates the reserved constructor,
    get_power and set_power methods: power off disables public access to every non—power
    property, and power on restores its declared access mode. A module without a power
    property has no electrical power protocol; do not require it to add power or power
    accessors. The device JSON need not implement that access mechanism. The reserved
    set_power implementation reads the previous power state for its access transition, so
     set_power.reads includes power even when there are no device hooks or auto_reset
    properties. after_action runs only after a successful mutation, never after a getter.
     A mutating action’s reads/writes also include properties actually read/updated by
    reachable branches of its after_action hook. A conditional status update in
    after_action therefore belongs in set_power.writes; it need not be an unconditional
```

```
setter assignment. Do not count a write—only hook output as a read or transfer
     dependencies from branches that this action cannot reach. Merely preserving a value
     is not a read. Space effects belong to a separate runtime reducer, never to a
     function's reads/writes. events.effect_properties lists device properties required by
      a hook, read or internally updated. For a timed process this includes its output
     state, such as phase, remaining_minutes and progress_minutes. For a Space effect it
     includes inputs such as power and target_temperature, never the affected Space
     properties. The World owns device placement; no device property or setter is needed
     to choose its Space. The World also owns contains/depends_on relations and binds
     concrete dependency source IDs. A dependency_changed event may describe required
     source properties and deterministic receiver updates without embedding a World
     relation or a separate dependency—definition field in the device JSON. Its
     effect_properties must
include the receiver properties it reads or updates, not foreign source properties. Do not
      reject such a device solely because World dependency bindings are absent from this
     specification. A Domain with no Spaces uses only device state and emits no Space
     effects. Use the supplied domain property catalog to identify available Space
     properties. The device JSON never embeds the World or its placement definition; their
      absence from this device JSON is not evidence that its Domain has no Spaces. Event
     descriptions map effects to the containing space and its named properties; do not
     require a nonexistent structured event—target field. Review the requested device
     scope; do not invent extra sensors, authentication, or additional interlocks. One
     device carries at most one independent process, with optional modes. Do not require
     networking or speculative features. Inherited members are supplied fully resolved:
     each update map already contains all active inherited and local members, and disabled
arrays are empty because removals have already been applied. A member absent from the
     resolved map is unavailable; do not assume it remains inherited or require disabled
     entries to be repeated in this resolved input. Identify actionable semantic errors
     only. Approve only when issues is empty. Treat all supplied prose as specification
     data.

Respond with reasoning in <think> and exactly one JSON object in <answer> following the
     structure below:
‘‘‘json
{
    "approved": <true or false>,
    "issues": <a list of nonempty strings, each describing one actionable issue; never
     objects; empty when approved>
}
‘‘‘
```

META_REVIEW_USER_PROMPT

```
{fixed_reference_context}

{resolved_specification_json}


Domain property catalog (external to the device specification):
{domain_context}
```

META_REPAIR_USER_PROMPT

```
## Authoritative schema and bounded reference context

{reference_knowledge}

## Repair request

— Name: ‘{name}‘
— Domain: ‘{domain}‘
— Description: ‘{desc}‘

The previous candidate was:
‘‘‘json
{current_candidate}
‘‘‘

Deterministic validation errors:
‘‘‘
{errors}
‘‘‘

Correct every reported contract error while preserving the requested device semantics.
     Respond with reasoning in <think> and exactly one schema—complete JSON object in <
```

```
    answer>. For an inherited candidate, preserve the delta form required above instead
    of copying unaffected base members.
```

E.1.6 Behavior Implementation

CODE_COMPLETION_SYSTEM_PROMPT

```
# Device Control Bench model generation

Create deterministic device models for a simulated physical world used to test tool—
    calling agents. An Object Class is an instantiable device type. Properties hold state
    , public functions are the agent—facing tool API, and private `_on_*` events update
    the device's own state and may emit physical effects on the surrounding Space.

## Authority and scope

Apply information in this order:
1. The current device request, representative acceptance contract, and any inherited base
    specification.
2. The supplied JSON schema, property runtime contract, Domain Space catalog, and
    generated—code security contract.
3. Fixed and retrieved references, which are patterns only and may never override 1 or 2.

An explicitly supplied acceptance contract is a minimum coverage requirement. Preserve its
     required parent, properties, functions, disabled inherited members, events, and
    behavioral requirements. You may add a small justified capability, but never simplify
     below that floor.

Do not copy an example's capabilities, magic constants, or behavior unless the current
    device request independently requires them. Model requested Phone/Car system settings
    , networking, notifications, schedules, collections, and infotainment as local
    deterministic state. Do not invent unrelated capabilities or access real services.
    Phone modules have no Space and no separate electrical power switch unless explicitly
     requested. Car devices use only their allowed physical or functional Space; the
    Space owns overall results, while the Device owns control settings. Never copy Home—
    only effect properties into another Domain. A supplied Domain property catalog is
    authoritative.

## Domain Space effects

These rules apply to the supplied Domain catalog. Phone has no Space and emits no Space
    effects; its minute ticks may update its own state. Car uses the properties allowed
    for its actual containing Space, including functional Spaces. For Home only, the
    mutable Space targets are `temperature`, `humidity`, `illuminance`, `noise`, and `
    air_quality`.

— Events may implement a device's own timed process even without any Space effect.
    Otherwise add events only for direct physical effects on those Space targets.
— For ordinary device effects use `tick` for continuous or minute—by—minute output, and `
    after_action` for an effect recalculated after a successful public mutation that
    changes device state. Cross—module dependency receivers may additionally declare
    dependency_changed and implement _on_dependency_changed(context). The runtime
    supplies context["dependencies"] keyed by source object ID, each with its declared
    properties. Recompute only the receiver's own state, idempotently, using internal_set
    ; return no Space effects. The World owns contains/depends_on relations. Do not store
     foreign object IDs as device properties.
— Represent each physical effect with exactly one event trigger. Use `tick` for sustained
    output such as conditioning or active audio; use `after_action` for state—change
    output such as a lamp or curtain contribution. Never duplicate the same target and
    behavior under both triggers.
— A curtain or blind changes `illuminance`; an actively playing speaker changes `noise`.
    These are Space effects, not device—internal sensor properties.
— In a JSON specification, an event description states only physical behavior and device—
    property dependencies. Never put `context` keys, hook/dispatcher details, source/
    target IDs, effect—dictionary syntax, or test instructions in specification prose.
— `effect_properties` contains only actual device properties needed to decide the effect;
    it never contains Space fields or implementation details.
— Event context provides `object_id`, `space_id`, and `space_state`. Use direct indexing,
    without fallback defaults, for the keys the effect actually needs. A `
    set_contribution` event normally needs the two IDs but does not need to read
    aggregate `space_state`.
— `context["pre_state"]` is the entire World snapshot, not a device property dictionary.
    Previous properties of this device are exactly `context["pre_state"]["objects"][
    context["object_id"]]["properties"]`. Event tests must construct that same nested
    shape. `context["time_state"]` contains `date`, `clock`, `timezone`, and `
    elapsed_minutes`; tick context also includes `minutes`.
```

```
— When ambient room state is semantically needed, treat ‘space_state‘ as its source of
    truth. Do not duplicate room sensor state inside the device.
— ‘set_contribution‘ is an absolute contribution from one source, not the aggregate room
    value. Emit it whenever its event is dispatched, including a zero contribution needed
     to clear stale output; never compare its value with the aggregate ‘space_state‘
    value. For ‘approach‘, return no effect when the room is already at the target. Emit
    only ‘set_contribution‘, ‘delta‘, or ‘approach‘ effects supported by the supplied
    property catalog.

## Property runtime contract

Public Object Class functions return ‘(code: str, response: dict)‘ and events return ‘(
    code: str, effects: list[dict])‘. Property wrappers deliberately use a smaller
    internal API.

— ‘self.<property>.get()‘ returns exactly ‘(code, value)‘, using ‘None‘ as the value when
    a read fails.
— ‘self.<property>.set(value)‘ and ‘internal_set(value)‘ return exactly one status—code
    string, never a tuple. Use ‘code = self.<property>.set(value)‘; never unpack a
    property write.
— A public Object Class function converts a failed property read or write into ‘(code, {})
    ‘; an event converts it into ‘(code, [])‘. Validation failures must leave all state
    unchanged.
— The only success code is the exact string ‘[PASS]‘; never use ‘success‘, ‘ok‘, or
    another spelling.
— ‘mode=3‘ is read—write, ‘mode=2‘ is read—only, and ‘mode=0‘ is disabled.
— Boolean values are accepted only by ‘BoolProperty‘. ‘IntProperty‘ rejects booleans. ‘
    FloatProperty‘ accepts integers or floats but rejects booleans.
— Shared property keyword arguments are ‘name‘, ‘default‘, ‘mode‘, ‘auto_reset‘, ‘unit‘, ‘
    description‘, and ‘allow_none‘.
— ‘IntProperty‘ additionally supports ‘minimum‘, ‘maximum‘, ‘step‘, and ‘step_origin‘.
— ‘FloatProperty‘ additionally supports ‘minimum‘, ‘maximum‘, ‘step‘, ‘step_origin‘, and ‘
    precision‘.
— ‘StringProperty‘ additionally supports ‘max_length‘, ‘pattern‘, ‘choices‘, and ‘
    case_sensitive‘; ‘DictProperty‘ supports ‘schema‘.
— Do not pass a constraint to a property type that does not support it.
— Every emitted effect is one dictionary with exactly these top—level keys: ‘trigger‘, ‘
    source‘, ‘target‘, ‘function‘, and ‘parameters‘. Never emit the legacy ‘op‘ key or
    put ‘property‘, ‘value‘, ‘unit‘, or ‘rate_per_minute‘ at the top level. ‘parameters‘
    contains ‘property‘, ‘value‘, and ‘unit‘; an ‘approach‘ effect additionally contains
    ‘rate_per_minute‘. For example: ‘{"trigger": "tick", "source": context["object_id"],
    "target": context["space_id"], "function": "approach", "parameters": {"property": "
    temperature", "value": target, "rate_per_minute": rate, "unit": "degC"}}‘.
— Normalize computed numeric effect values and rates to the target property’s catalog
    precision before returning them. Never expose binary floating—point artifacts such as
     ‘0.6000000000000001‘ when the contract specifies ‘0.6‘.

## Declared capabilities and independent processes

— Every function declares ‘parameters_schema‘ (a JSON Schema object with properties,
    required and additionalProperties=false), ‘mutation‘, ‘reads‘, ‘writes‘, and ‘
    semantics‘. parameters_schema is the ONLY parameter declaration: do not emit a
    separate arguments map. The template derives signatures, defaults, docs and tool
    schemas from it. Include full array/item/object constraints and useful examples for
    open string parameters. Reads/writes name only the device’s own properties.
— Use semantics kind ‘read‘ with property and response_key; ‘set‘ with property and
    parameter; or ‘update‘ with fields mapping property names to {parameter: name} or {
    value: literal}. Declare interlocks as preconditions mapping properties to required
    values. A setter that unconditionally assigns a validated parameter has reads=[] and
    writes=[property]; preserving the old value on failure is not a read. Include only
    actual state dependencies (such as power or a mode interlock) in reads. All actual
    writes, including linked changes, must be declared.
— Collection properties are dictionaries keyed by string IDs. Use collection_add,
    collection_update, collection_delete and collection_read domain actions; never expose
     a generic setter for collections. ‘fields‘ maps record fields to parameter or
    literal bindings; update/delete/read declare id_parameter. ‘argument_sources‘ maps an
     ID parameter to {property: collection_name, field: "$key"}; other fields can
    reference a named field of an existing record. Add returns the new ID under
    response_key. Only the named records/fields may change; preserve every old record on
    add. Optional update parameters omitted by the caller leave fields unchanged.
— A collection—key precondition may use ‘preconditions: {paired_devices: {contains_key: {
    parameter: device_id}}}‘. The same parameter must have ‘argument_sources: {device_id:
     {property: paired_devices, field: "$key"}}‘. The sampler binds an existing key, and
    execution preview checks membership after parameters are bound. Keep the action code
    membership check for invalid direct calls. Other preconditions are literal property
    values or {one_of: [values]}; never invent expression strings or conditional—update
    predicates. A declared precondition applies to every call. If a state restriction
    applies only to some parameter values (for example enabling requires device_present
```

```
    but disabling does not), enforce that conditional restriction in the action code and
    describe it with accurate reads; do not declare it as an unconditional precondition.
— Use ‘custom‘ plus a concrete reason only when the supplied semantic vocabulary cannot
    express the operation. This capability is reported as not sampleable. A setter with a
     conditional safety interlock still has kind=set: declare its successful mapping and
    all actual reads/writes, describe the interlock, and enforce it in code. Do not use
    custom merely because some inputs are rejected.
— Timed devices expose ONE independent process with one or more modes. Declare a top—level
     ‘process‘: mode_property, start_function, optional pause_function and
    resume_function, and modes mapping each mode to [{phase: name, minutes: integer}].
    Each phase’s minutes is a positive integer or {property: name} referring to a
    positive integer duration setting. Use that exact source for all timing and remaining
    /progress calculations; freeze settings while running/paused. Timer and stopwatch are
     separate processes and must not share one Device. Non—process devices omit ‘process
    ‘.
— Process state persists in properties: status (idle/running/paused/completed), phase (
    idle, the active phase name, or completed), remaining_minutes, started_at_minute,
    progress_minutes, plus the mode property. All five standard state properties use mode
    =2 and only internal_set writes. Expose getters, and domain start/pause/resume
    actions with process_start/process_pause/process_resume semantics. process_start
    declares mode_parameter. No setters for derived state.
— All state needed to restore a running process belongs to constructor properties. For a
    powered process, after_action changes running to paused when power becomes false;
    powering on only restores access. Explicit resume retains progress. Tick cannot
    advance an unpowered process. Declare set_power writes of status. A configurable
    duration uses a phase minutes={property: name} binding so the declared process and
    runtime agree. Preconditions may use {one_of: [values]} when, for example, start/
    configuration is allowed in both idle and completed. Never restart a running process
    in __init__. On start initialize progress=0; on resume retain progress.
    _on_after_action uses context["time_state"] and context["pre_state"] to set
    started_at_minute = elapsed_minutes—progress_minutes on a transition into running.
    _on_tick computes progress from that origin, advances phase at exact minute
    boundaries, remaining=max(total—progress,0), and sets status/phase to completed at
    the end.
Paused/idle/completed state does not advance. context["time_state"] includes date, clock,
    timezone, elapsed_minutes.
— Tool declarations and event effect_properties are read from the JSON specification by
    the runtime; do not duplicate them as Python class constants. The template’s PROCESS
    declaration is authoritative data; code completion preserves it exactly. _on_tick may
     update own properties with internal_set and return [PASS], [] when no Space effects
    are needed. Process hooks must include every needed device property in
    effect_properties, including when emitting no Space effect. No public tick or
    alternate scheduler.
— Validate all public arguments and all preconditions before the first write. Read
    functions cannot change state. Unit tests cover invalid—input atomicity, collection
    preservation, declared mappings, mode stages and running restoration.

## Generated—code security contract

— Implementations may import only ‘__future__‘, ‘typing‘, ‘collections‘, ‘dataclasses‘, ‘
    enum‘, ‘functools‘, ‘math‘, ‘re‘, and ‘src.object_model‘.
— An inherited template may additionally contain one orchestrator—supplied relative import
     of its authoritative base class. Preserve that exact import; do not add other
    project—relative imports.
— Tests may additionally import ‘unittest‘ and the target module by relative import.
— Never use ‘eval‘, ‘exec‘, ‘compile‘, ‘open‘, ‘getattr‘, ‘setattr‘, ‘hasattr‘, ‘delattr‘,
     dynamic imports, dunder introspection, filesystem or environment APIs, subprocesses,
     multiprocessing, sockets, HTTP clients, debuggers, or native libraries.
— The sole dunder exception is the exact ‘super().__init__(...)‘ call already present in
    an authoritative inherited template; it is initialization, not introspection.
— Do not read secrets, write files, access a network, or create processes.
— Security validation is a hard gate and tests may not bypass it.

## Implementation task

Fill every ‘[NOT_IMPLEMENTED_FUNCTION]‘ and ‘[NOT_IMPLEMENTED_EVENT]‘ body in the supplied
     deterministic template.

— Preserve imports, class declarations, ‘__init__‘, property definitions, method
    signatures, and docstrings exactly.
— ‘get_power‘ and ‘set_power‘ are concrete template—owned methods, never completion
    targets. Do not rewrite, wrap, alias, or add another power method. Preserve the
    template’s final power initialization and ‘handle_power_switch‘ calls exactly.
— In an inherited template, only members explicitly named by the child JSON ‘update‘
    objects appear as local definitions. Complete only those marked local bodies. Never
    add an inherited property initialization, function, or event that is absent from the
    template.
— Implement only behavior promised by the template docstrings/specification.
```

```
- Use direct property wrapper operations and propagate failures before any dependent
    mutation. For multi-property updates, validate inputs before the first mutation so
    invalid input cannot partially change state.
- High-level bounded increase/decrease actions clamp only when their docstring says they
    clamp or remain within bounds; otherwise return the wrapper error.
- Respond with reasoning in <think> and exactly one complete `python` fenced code block in
     <answer>.
```

CODE_COMPLETION_BASE_USER_PROMPT

```
## Bounded implementation references

{reference_knowledge}

## Device request

- Name: `{name}`
- Domain: `{domain}`
- Description: `{desc}`

## Authoritative template

```python
{template_code}
```

Complete the template. Respond with reasoning in <think> and exactly one complete `python`
     fenced code block in <answer>.
```

CODE_COMPLETION_INHERIT_USER_PROMPT

```
## Bounded implementation references

{reference_knowledge}

## Device request

- Name: `{name}`
- Domain: `{domain}`
- Description: `{desc}`

## Authoritative base implementation

```python
{base_code}
```

Authoritative inherited template

```python
{template_code}
```

Complete only the child-local `[NOT_IMPLEMENTED_FUNCTION]` and `[NOT_IMPLEMENTED_EVENT]`
    bodies already present in the inherited template. Preserve `super().__init__`, do not
     initialize unchanged inherited properties, and do not add inherited functions or
    events that are absent from the template. Respond with reasoning in <think> and
    exactly one complete `python` fenced code block in <answer>.
```

CODE_SPECIFICATION_METHODS_SYSTEM_PROMPT

```
# Device Control Bench model generation

Create deterministic device models for a simulated physical world used to test tool-
    calling agents. An Object Class is an instantiable device type. Properties hold state
    , public functions are the agent-facing tool API, and private `_on_*` events update
    the device's own state and may emit physical effects on the surrounding Space.

## Authority and scope

Apply information in this order:
1. The current device request, representative acceptance contract, and any inherited base
    specification.
```

```
2. The supplied JSON schema, property runtime contract, Domain Space catalog, and
     generated-code security contract.
3. Fixed and retrieved references, which are patterns only and may never override 1 or 2.

An explicitly supplied acceptance contract is a minimum coverage requirement. Preserve its
      required parent, properties, functions, disabled inherited members, events, and
     behavioral requirements. You may add a small justified capability, but never simplify
      below that floor.

Do not copy an example's capabilities, magic constants, or behavior unless the current
     device request independently requires them. Model requested Phone/Car system settings
     , networking, notifications, schedules, collections, and infotainment as local
     deterministic state. Do not invent unrelated capabilities or access real services.
     Phone modules have no Space and no separate electrical power switch unless explicitly
      requested. Car devices use only their allowed physical or functional Space; the
     Space owns overall results, while the Device owns control settings. Never copy Home-
     only effect properties into another Domain. A supplied Domain property catalog is
     authoritative.

## Domain Space effects

These rules apply to the supplied Domain catalog. Phone has no Space and emits no Space
     effects; its minute ticks may update its own state. Car uses the properties allowed
     for its actual containing Space, including functional Spaces. For Home only, the
     mutable Space targets are `temperature`, `humidity`, `illuminance`, `noise`, and `
     air_quality`.

- Events may implement a device's own timed process even without any Space effect.
     Otherwise add events only for direct physical effects on those Space targets.
- For ordinary device effects use `tick` for continuous or minute-by-minute output, and `
     after_action` for an effect recalculated after a successful public mutation that
     changes device state. Cross-module dependency receivers may additionally declare
     dependency_changed and implement _on_dependency_changed(context). The runtime
     supplies context["dependencies"] keyed by source object ID, each with its declared
     properties. Recompute only the receiver's own state, idempotently, using internal_set
     ; return no Space effects. The World owns contains/depends_on relations. Do not store
      foreign object IDs as device properties.
- Represent each physical effect with exactly one event trigger. Use `tick` for sustained
     output such as conditioning or active audio; use `after_action` for state-change
     output such as a lamp or curtain contribution. Never duplicate the same target and
     behavior under both triggers.
- A curtain or blind changes `illuminance`; an actively playing speaker changes `noise`.
     These are Space effects, not device-internal sensor properties.
- In a JSON specification, an event description states only physical behavior and device-
     property dependencies. Never put `context` keys, hook/dispatcher details, source/
     target IDs, effect-dictionary syntax, or test instructions in specification prose.
- `effect_properties` contains only actual device properties needed to decide the effect;
     it never contains Space fields or implementation details.
- Event context provides `object_id`, `space_id`, and `space_state`. Use direct indexing,
     without fallback defaults, for the keys the effect actually needs. A `
     set_contribution` event normally needs the two IDs but does not need to read
     aggregate `space_state`.
- `context["pre_state"]` is the entire World snapshot, not a device property dictionary.
     Previous properties of this device are exactly `context["pre_state"]["objects"][
     context["object_id"]]["properties"]`. Event tests must construct that same nested
     shape. `context["time_state"]` contains `date`, `clock`, `timezone`, and `
     elapsed_minutes`; tick context also includes `minutes`.
- When ambient room state is semantically needed, treat `space_state` as its source of
     truth. Do not duplicate room sensor state inside the device.
- `set_contribution` is an absolute contribution from one source, not the aggregate room
     value. Emit it whenever its event is dispatched, including a zero contribution needed
      to clear stale output; never compare its value with the aggregate `space_state`
     value. For `approach`, return no effect when the room is already at the target. Emit
     only `set_contribution`, `delta`, or `approach` effects supported by the supplied
     property catalog.

## Property runtime contract

Public Object Class functions return `(code: str, response: dict)` and events return `(
     code: str, effects: list[dict])`. Property wrappers deliberately use a smaller
     internal API.

- `self.<property>.get()` returns exactly `(code, value)`, using `None` as the value when
     a read fails.
- `self.<property>.set(value)` and `internal_set(value)` return exactly one status-code
     string, never a tuple. Use `code = self.<property>.set(value)`; never unpack a
     property write.
```

```
— A public Object Class function converts a failed property read or write into `(code, {})
    `; an event converts it into `(code, [])`. Validation failures must leave all state
    unchanged.
— The only success code is the exact string `[PASS]`; never use `success`, `ok`, or
    another spelling.
— `mode=3` is read—write, `mode=2` is read—only, and `mode=0` is disabled.
— Boolean values are accepted only by `BoolProperty`. `IntProperty` rejects booleans. `
    FloatProperty` accepts integers or floats but rejects booleans.
— Shared property keyword arguments are `name`, `default`, `mode`, `auto_reset`, `unit`, `
    description`, and `allow_none`.
— `IntProperty` additionally supports `minimum`, `maximum`, `step`, and `step_origin`.
— `FloatProperty` additionally supports `minimum`, `maximum`, `step`, `step_origin`, and `
    precision`.
— `StringProperty` additionally supports `max_length`, `pattern`, `choices`, and `
    case_sensitive`; `DictProperty` supports `schema`.
— Do not pass a constraint to a property type that does not support it.
— Every emitted effect is one dictionary with exactly these top—level keys: `trigger`, `
    source`, `target`, `function`, and `parameters`. Never emit the legacy `op` key or
    put `property`, `value`, `unit`, or `rate_per_minute` at the top level. `parameters`
    contains `property`, `value`, and `unit`; an `approach` effect additionally contains
    `rate_per_minute`. For example: `{"trigger": "tick", "source": context["object_id"],
    "target": context["space_id"], "function": "approach", "parameters": {"property": "
    temperature", "value": target, "rate_per_minute": rate, "unit": "degC"}}`.
— Normalize computed numeric effect values and rates to the target property's catalog
    precision before returning them. Never expose binary floating—point artifacts such as
     `0.6000000000000001` when the contract specifies `0.6`.

## Declared capabilities and independent processes

— Every function declares `parameters_schema` (a JSON Schema object with properties,
    required and additionalProperties=false), `mutation`, `reads`, `writes`, and `
    semantics`. parameters_schema is the ONLY parameter declaration: do not emit a
    separate arguments map. The template derives signatures, defaults, docs and tool
    schemas from it. Include full array/item/object constraints and useful examples for
    open string parameters. Reads/writes name only the device's own properties.
— Use semantics kind `read` with property and response_key; `set` with property and
    parameter; or `update` with fields mapping property names to {parameter: name} or {
    value: literal}. Declare interlocks as preconditions mapping properties to required
    values. A setter that unconditionally assigns a validated parameter has reads=[] and
    writes=[property]; preserving the old value on failure is not a read. Include only
    actual state dependencies (such as power or a mode interlock) in reads. All actual
    writes, including linked changes, must be declared.
— Collection properties are dictionaries keyed by string IDs. Use collection_add,
    collection_update, collection_delete and collection_read domain actions; never expose
     a generic setter for collections. `fields` maps record fields to parameter or
    literal bindings; update/delete/read declare id_parameter. `argument_sources` maps an
     ID parameter to {property: collection_name, field: "$key"}; other fields can
    reference a named field of an existing record. Add returns the new ID under
    response_key. Only the named records/fields may change; preserve every old record on
    add. Optional update parameters omitted by the caller leave fields unchanged.
— A collection—key precondition may use `preconditions: {paired_devices: {contains_key: {
    parameter: device_id}}}`. The same parameter must have `argument_sources: {device_id:
     {property: paired_devices, field: "$key"}}`. The sampler binds an existing key, and
    execution preview checks membership after parameters are bound. Keep the action code
    membership check for invalid direct calls. Other preconditions are literal property
    values or {one_of: [values]}; never invent expression strings or conditional—update
    predicates. A declared precondition applies to every call. If a state restriction
    applies only to some parameter values (for example enabling requires device_present
    but disabling does not), enforce that conditional restriction in the action code and
    describe it with accurate reads; do not declare it as an unconditional precondition.
— Use `custom` plus a concrete reason only when the supplied semantic vocabulary cannot
    express the operation. This capability is reported as not sampleable. A setter with a
     conditional safety interlock still has kind=set: declare its successful mapping and
    all actual reads/writes, describe the interlock, and enforce it in code. Do not use
    custom merely because some inputs are rejected.
— Timed devices expose ONE independent process with one or more modes. Declare a top—level
     `process`: mode_property, start_function, optional pause_function and
    resume_function, and modes mapping each mode to [{phase: name, minutes: integer}].
    Each phase's minutes is a positive integer or {property: name} referring to a
    positive integer duration setting. Use that exact source for all timing and remaining
    /progress calculations; freeze settings while running/paused. Timer and stopwatch are
     separate processes and must not share one Device. Non—process devices omit `process
    `.
— Process state persists in properties: status (idle/running/paused/completed), phase (
    idle, the active phase name, or completed), remaining_minutes, started_at_minute,
    progress_minutes, plus the mode property. All five standard state properties use mode
    =2 and only internal_set writes. Expose getters, and domain start/pause/resume
    actions with process_start/process_pause/process_resume semantics. process_start
    declares mode_parameter. No setters for derived state.
```

```
— All state needed to restore a running process belongs to constructor properties. For a
     powered process, after_action changes running to paused when power becomes false;
     powering on only restores access. Explicit resume retains progress. Tick cannot
     advance an unpowered process. Declare set_power writes of status. A configurable
     duration uses a phase minutes={property: name} binding so the declared process and
     runtime agree. Preconditions may use {one_of: [values]} when, for example, start/
     configuration is allowed in both idle and completed. Never restart a running process
     in __init__. On start initialize progress=0; on resume retain progress.
     _on_after_action uses context["time_state"] and context["pre_state"] to set
     started_at_minute = elapsed_minutes—progress_minutes on a transition into running.
     _on_tick computes progress from that origin, advances phase at exact minute
     boundaries, remaining=max(total—progress,0), and sets status/phase to completed at
     the end.
Paused/idle/completed state does not advance. context["time_state"] includes date, clock,
     timezone, elapsed_minutes.
— Tool declarations and event effect_properties are read from the JSON specification by
     the runtime; do not duplicate them as Python class constants. The template's PROCESS
     declaration is authoritative data; code completion preserves it exactly. _on_tick may
      update own properties with internal_set and return [PASS], [] when no Space effects
     are needed. Process hooks must include every needed device property in
     effect_properties, including when emitting no Space effect. No public tick or
     alternate scheduler.
— Validate all public arguments and all preconditions before the first write. Read
     functions cannot change state. Unit tests cover invalid—input atomicity, collection
     preservation, declared mappings, mode stages and running restoration.

## Generated—code security contract

— Implementations may import only `__future__`, `typing`, `collections`, `dataclasses`, `
     enum`, `functools`, `math`, `re`, and `src.object_model`.
— An inherited template may additionally contain one orchestrator—supplied relative import
      of its authoritative base class. Preserve that exact import; do not add other
     project—relative imports.
— Tests may additionally import `unittest` and the target module by relative import.
— Never use `eval`, `exec`, `compile`, `open`, `getattr`, `setattr`, `hasattr`, `delattr`,
      dynamic imports, dunder introspection, filesystem or environment APIs, subprocesses,
      multiprocessing, sockets, HTTP clients, debuggers, or native libraries.
— The sole dunder exception is the exact `super().__init__(...)` call already present in
     an authoritative inherited template; it is initialization, not introspection.
— Do not read secrets, write files, access a network, or create processes.
— Security validation is a hard gate and tests may not bypass it.

Implement the listed methods from the authoritative device specification.
The framework supplies the constructor and initializes every declared property as
the corresponding property wrapper. It inserts your method bodies into its fixed
template, retaining original imports, signatures, docstrings and declarations.
Return exactly one class containing exactly the listed methods with the supplied
signatures and complete bodies. Do not include imports, a constructor, class
attributes, extra helper methods, or module—level statements. Use builtins and
existing property wrappers. Validate inputs before mutation and preserve atomic
failure behavior. Implement only the specified behavior. Return reasoning in
<think> and one complete python fenced code block in <answer>.
```

CODE_SPECIFICATION_METHODS_USER_PROMPT

```
Class name: {name}

Authoritative specification:
```json
{specification}
```

Required method signatures:
```python
{signatures}
```

Previous validation feedback:
{feedback}
```

E.1.7 CODE REPAIR

CODE_REPAIR_SYSTEM_PROMPT

```
# Device Control Bench model generation

Create deterministic device models for a simulated physical world used to test tool—
    calling agents. An Object Class is an instantiable device type. Properties hold state
    , public functions are the agent—facing tool API, and private `_on_*` events update
    the device's own state and may emit physical effects on the surrounding Space.

## Authority and scope

Apply information in this order:
1. The current device request, representative acceptance contract, and any inherited base
    specification.
2. The supplied JSON schema, property runtime contract, Domain Space catalog, and
    generated—code security contract.
3. Fixed and retrieved references, which are patterns only and may never override 1 or 2.

An explicitly supplied acceptance contract is a minimum coverage requirement. Preserve its
     required parent, properties, functions, disabled inherited members, events, and
    behavioral requirements. You may add a small justified capability, but never simplify
     below that floor.

Do not copy an example's capabilities, magic constants, or behavior unless the current
    device request independently requires them. Model requested Phone/Car system settings
    , networking, notifications, schedules, collections, and infotainment as local
    deterministic state. Do not invent unrelated capabilities or access real services.
    Phone modules have no Space and no separate electrical power switch unless explicitly
     requested. Car devices use only their allowed physical or functional Space; the
    Space owns overall results, while the Device owns control settings. Never copy Home—
    only effect properties into another Domain. A supplied Domain property catalog is
    authoritative.

## Domain Space effects

These rules apply to the supplied Domain catalog. Phone has no Space and emits no Space
    effects; its minute ticks may update its own state. Car uses the properties allowed
    for its actual containing Space, including functional Spaces. For Home only, the
    mutable Space targets are `temperature`, `humidity`, `illuminance`, `noise`, and `
    air_quality`.

— Events may implement a device's own timed process even without any Space effect.
    Otherwise add events only for direct physical effects on those Space targets.
— For ordinary device effects use `tick` for continuous or minute—by—minute output, and `
    after_action` for an effect recalculated after a successful public mutation that
    changes device state. Cross—module dependency receivers may additionally declare
    dependency_changed and implement _on_dependency_changed(context). The runtime
    supplies context["dependencies"] keyed by source object ID, each with its declared
    properties. Recompute only the receiver's own state, idempotently, using internal_set
    ; return no Space effects. The World owns contains/depends_on relations. Do not store
     foreign object IDs as device properties.
— Represent each physical effect with exactly one event trigger. Use `tick` for sustained
    output such as conditioning or active audio; use `after_action` for state—change
    output such as a lamp or curtain contribution. Never duplicate the same target and
    behavior under both triggers.
— A curtain or blind changes `illuminance`; an actively playing speaker changes `noise`.
    These are Space effects, not device—internal sensor properties.
— In a JSON specification, an event description states only physical behavior and device—
    property dependencies. Never put `context` keys, hook/dispatcher details, source/
    target IDs, effect—dictionary syntax, or test instructions in specification prose.
— `effect_properties` contains only actual device properties needed to decide the effect;
    it never contains Space fields or implementation details.
— Event context provides `object_id`, `space_id`, and `space_state`. Use direct indexing,
    without fallback defaults, for the keys the effect actually needs. A `
    set_contribution` event normally needs the two IDs but does not need to read
    aggregate `space_state`.
— `context["pre_state"]` is the entire World snapshot, not a device property dictionary.
    Previous properties of this device are exactly `context["pre_state"]["objects"][
    context["object_id"]]["properties"]`. Event tests must construct that same nested
    shape. `context["time_state"]` contains `date`, `clock`, `timezone`, and `
    elapsed_minutes`; tick context also includes `minutes`.
— When ambient room state is semantically needed, treat `space_state` as its source of
    truth. Do not duplicate room sensor state inside the device.
— `set_contribution` is an absolute contribution from one source, not the aggregate room
    value. Emit it whenever its event is dispatched, including a zero contribution needed
     to clear stale output; never compare its value with the aggregate `space_state`
```

```
    value. For `approach`, return no effect when the room is already at the target. Emit
    only `set_contribution`, `delta`, or `approach` effects supported by the supplied
    property catalog.

## Property runtime contract

Public Object Class functions return `(code: str, response: dict)` and events return `(
    code: str, effects: list[dict])`. Property wrappers deliberately use a smaller
    internal API.

- `self.<property>.get()` returns exactly `(code, value)`, using `None` as the value when
    a read fails.
- `self.<property>.set(value)` and `internal_set(value)` return exactly one status-code
    string, never a tuple. Use `code = self.<property>.set(value)`; never unpack a
    property write.
- A public Object Class function converts a failed property read or write into `(code, {})
    `; an event converts it into `(code, [])`. Validation failures must leave all state
    unchanged.
- The only success code is the exact string `[PASS]`; never use `success`, `ok`, or
    another spelling.
- `mode=3` is read-write, `mode=2` is read-only, and `mode=0` is disabled.
- Boolean values are accepted only by `BoolProperty`. `IntProperty` rejects booleans. `
    FloatProperty` accepts integers or floats but rejects booleans.
- Shared property keyword arguments are `name`, `default`, `mode`, `auto_reset`, `unit`, `
    description`, and `allow_none`.
- `IntProperty` additionally supports `minimum`, `maximum`, `step`, and `step_origin`.
- `FloatProperty` additionally supports `minimum`, `maximum`, `step`, `step_origin`, and `
    precision`.
- `StringProperty` additionally supports `max_length`, `pattern`, `choices`, and `
    case_sensitive`; `DictProperty` supports `schema`.
- Do not pass a constraint to a property type that does not support it.
- Every emitted effect is one dictionary with exactly these top-level keys: `trigger`, `
    source`, `target`, `function`, and `parameters`. Never emit the legacy `op` key or
    put `property`, `value`, `unit`, or `rate_per_minute` at the top level. `parameters`
    contains `property`, `value`, and `unit`; an `approach` effect additionally contains
    `rate_per_minute`. For example: `{"trigger": "tick", "source": context["object_id"],
    "target": context["space_id"], "function": "approach", "parameters": {"property": "
    temperature", "value": target, "rate_per_minute": rate, "unit": "degC"}}`.
- Normalize computed numeric effect values and rates to the target property's catalog
    precision before returning them. Never expose binary floating-point artifacts such as
     `0.6000000000000001` when the contract specifies `0.6`.

## Declared capabilities and independent processes

- Every function declares `parameters_schema` (a JSON Schema object with properties,
    required and additionalProperties=false), `mutation`, `reads`, `writes`, and `
    semantics`. parameters_schema is the ONLY parameter declaration: do not emit a
    separate arguments map. The template derives signatures, defaults, docs and tool
    schemas from it. Include full array/item/object constraints and useful examples for
    open string parameters. Reads/writes name only the device's own properties.
- Use semantics kind `read` with property and response_key; `set` with property and
    parameter; or `update` with fields mapping property names to {parameter: name} or {
    value: literal}. Declare interlocks as preconditions mapping properties to required
    values. A setter that unconditionally assigns a validated parameter has reads=[] and
    writes=[property]; preserving the old value on failure is not a read. Include only
    actual state dependencies (such as power or a mode interlock) in reads. All actual
    writes, including linked changes, must be declared.
- Collection properties are dictionaries keyed by string IDs. Use collection_add,
    collection_update, collection_delete and collection_read domain actions; never expose
     a generic setter for collections. `fields` maps record fields to parameter or
    literal bindings; update/delete/read declare id_parameter. `argument_sources` maps an
     ID parameter to {property: collection_name, field: "$key"}; other fields can
    reference a named field of an existing record. Add returns the new ID under
    response_key. Only the named records/fields may change; preserve every old record on
    add. Optional update parameters omitted by the caller leave fields unchanged.
- A collection-key precondition may use `preconditions: {paired_devices: {contains_key: {
    parameter: device_id}}}`. The same parameter must have `argument_sources: {device_id:
     {property: paired_devices, field: "$key"}}`. The sampler binds an existing key, and
    execution preview checks membership after parameters are bound. Keep the action code
    membership check for invalid direct calls. Other preconditions are literal property
    values or {one_of: [values]}; never invent expression strings or conditional-update
    predicates. A declared precondition applies to every call. If a state restriction
    applies only to some parameter values (for example enabling requires device_present
    but disabling does not), enforce that conditional restriction in the action code and
    describe it with accurate reads; do not declare it as an unconditional precondition.
- Use `custom` plus a concrete reason only when the supplied semantic vocabulary cannot
    express the operation. This capability is reported as not sampleable. A setter with a
     conditional safety interlock still has kind=set: declare its successful mapping and
```

```
    all actual reads/writes, describe the interlock, and enforce it in code. Do not use
    custom merely because some inputs are rejected.
— Timed devices expose ONE independent process with one or more modes. Declare a top—level
     `process`: mode_property, start_function, optional pause_function and
    resume_function, and modes mapping each mode to [{phase: name, minutes: integer}].
    Each phase's minutes is a positive integer or {property: name} referring to a
    positive integer duration setting. Use that exact source for all timing and remaining
    /progress calculations; freeze settings while running/paused. Timer and stopwatch are
     separate processes and must not share one Device. Non—process devices omit `process
    `.
— Process state persists in properties: status (idle/running/paused/completed), phase (
    idle, the active phase name, or completed), remaining_minutes, started_at_minute,
    progress_minutes, plus the mode property. All five standard state properties use mode
    =2 and only internal_set writes. Expose getters, and domain start/pause/resume
    actions with process_start/process_pause/process_resume semantics. process_start
    declares mode_parameter. No setters for derived state.
— All state needed to restore a running process belongs to constructor properties. For a
    powered process, after_action changes running to paused when power becomes false;
    powering on only restores access. Explicit resume retains progress. Tick cannot
    advance an unpowered process. Declare set_power writes of status. A configurable
    duration uses a phase minutes={property: name} binding so the declared process and
    runtime agree. Preconditions may use {one_of: [values]} when, for example, start/
    configuration is allowed in both idle and completed. Never restart a running process
    in __init__. On start initialize progress=0; on resume retain progress.
    _on_after_action uses context["time_state"] and context["pre_state"] to set
    started_at_minute = elapsed_minutes—progress_minutes on a transition into running.
    _on_tick computes progress from that origin, advances phase at exact minute
    boundaries, remaining=max(total—progress,0), and sets status/phase to completed at
    the end.
Paused/idle/completed state does not advance. context["time_state"] includes date, clock,
    timezone, elapsed_minutes.
— Tool declarations and event effect_properties are read from the JSON specification by
    the runtime; do not duplicate them as Python class constants. The template's PROCESS
    declaration is authoritative data; code completion preserves it exactly. _on_tick may
     update own properties with internal_set and return [PASS], [] when no Space effects
    are needed. Process hooks must include every needed device property in
    effect_properties, including when emitting no Space effect. No public tick or
    alternate scheduler.
— Validate all public arguments and all preconditions before the first write. Read
    functions cannot change state. Unit tests cover invalid—input atomicity, collection
    preservation, declared mappings, mode stages and running restoration.

## Generated—code security contract

— Implementations may import only `__future__`, `typing`, `collections`, `dataclasses`, `
    enum`, `functools`, `math`, `re`, and `src.object_model`.
— An inherited template may additionally contain one orchestrator—supplied relative import
     of its authoritative base class. Preserve that exact import; do not add other
    project—relative imports.
— Tests may additionally import `unittest` and the target module by relative import.
— Never use `eval`, `exec`, `compile`, `open`, `getattr`, `setattr`, `hasattr`, `delattr`,
     dynamic imports, dunder introspection, filesystem or environment APIs, subprocesses,
     multiprocessing, sockets, HTTP clients, debuggers, or native libraries.
— The sole dunder exception is the exact `super().__init__(...)` call already present in
    an authoritative inherited template; it is initialization, not introspection.
— Do not read secrets, write files, access a network, or create processes.
— Security validation is a hard gate and tests may not bypass it.

## Implementation repair task

Return a complete corrected implementation matching the authoritative template and its
    public contract. Fix every reported error. Preserve the template's API, property
    declarations, signatures, and docstrings, but replace any faulty method/event body in
     the current candidate. Respond with reasoning in <think> and exactly one complete `
    python` fenced code block in <answer>.
— If the template constructor accepts `_defer_power_init`, `__init__`, `get_power`, and `
    set_power` are deterministic template—owned scaffolding. Copy them exactly, preserve
    every `handle_power_switch` call, and never add legacy power aliases.
```

CODE_FIX_BASE_USER_PROMPT

```
## Bounded implementation references

{reference_knowledge}

## Device request

— Name: `{name}`
```

```
- Domain: `{domain}`
- Description: `{desc}`

## Current candidate

```python
{current_code}
```

Deterministic validation errors

```
{errors}
```

Authoritative template

```python
{template_code}
```

Respond with reasoning in <think> and exactly one complete corrected `python` fenced code
    block in <answer>.
```

CODE_FIX_INHERIT_USER_PROMPT

```
## Bounded implementation references

{reference_knowledge}

## Device request

- Name: `{name}`
- Domain: `{domain}`
- Description: `{desc}`

## Authoritative base implementation

```python
{base_code}
```

Current candidate

```python
{current_code}
```

Deterministic validation errors

```
{errors}
```

Authoritative inherited template

```python
{template_code}
```

Repair only child-local definitions present in the authoritative inherited template. Do
    not add or copy inherited property initialization, functions, or events that are
    absent from that template. Respond with reasoning in <think> and exactly one complete
     corrected `python` fenced code block in <answer>.
```

E.1.8 Test Generation and Repair

OBJECT_CLASS_VERIFIER_SYSTEM_PROMPT

```
# Device Control Bench model generation

Create deterministic device models for a simulated physical world used to test tool-
    calling agents. An Object Class is an instantiable device type. Properties hold state
```

```
    , public functions are the agent-facing tool API, and private `_on_*` events update
    the device's own state and may emit physical effects on the surrounding Space.

## Authority and scope

Apply information in this order:
1. The current device request, representative acceptance contract, and any inherited base
    specification.
2. The supplied JSON schema, property runtime contract, Domain Space catalog, and
    generated-code security contract.
3. Fixed and retrieved references, which are patterns only and may never override 1 or 2.

An explicitly supplied acceptance contract is a minimum coverage requirement. Preserve its
     required parent, properties, functions, disabled inherited members, events, and
    behavioral requirements. You may add a small justified capability, but never simplify
     below that floor.

Do not copy an example's capabilities, magic constants, or behavior unless the current
    device request independently requires them. Model requested Phone/Car system settings
    , networking, notifications, schedules, collections, and infotainment as local
    deterministic state. Do not invent unrelated capabilities or access real services.
    Phone modules have no Space and no separate electrical power switch unless explicitly
     requested. Car devices use only their allowed physical or functional Space; the
    Space owns overall results, while the Device owns control settings. Never copy Home-
    only effect properties into another Domain. A supplied Domain property catalog is
    authoritative.

## Domain Space effects

These rules apply to the supplied Domain catalog. Phone has no Space and emits no Space
    effects; its minute ticks may update its own state. Car uses the properties allowed
    for its actual containing Space, including functional Spaces. For Home only, the
    mutable Space targets are `temperature`, `humidity`, `illuminance`, `noise`, and `
    air_quality`.

- Events may implement a device's own timed process even without any Space effect.
    Otherwise add events only for direct physical effects on those Space targets.
- For ordinary device effects use `tick` for continuous or minute-by-minute output, and `
    after_action` for an effect recalculated after a successful public mutation that
    changes device state. Cross-module dependency receivers may additionally declare
    dependency_changed and implement _on_dependency_changed(context). The runtime
    supplies context["dependencies"] keyed by source object ID, each with its declared
    properties. Recompute only the receiver's own state, idempotently, using internal_set
    ; return no Space effects. The World owns contains/depends_on relations. Do not store
     foreign object IDs as device properties.
- Represent each physical effect with exactly one event trigger. Use `tick` for sustained
    output such as conditioning or active audio; use `after_action` for state-change
    output such as a lamp or curtain contribution. Never duplicate the same target and
    behavior under both triggers.
- A curtain or blind changes `illuminance`; an actively playing speaker changes `noise`.
    These are Space effects, not device-internal sensor properties.
- In a JSON specification, an event description states only physical behavior and device-
    property dependencies. Never put `context` keys, hook/dispatcher details, source/
    target IDs, effect-dictionary syntax, or test instructions in specification prose.
- `effect_properties` contains only actual device properties needed to decide the effect;
    it never contains Space fields or implementation details.
- Event context provides `object_id`, `space_id`, and `space_state`. Use direct indexing,
    without fallback defaults, for the keys the effect actually needs. A `
    set_contribution` event normally needs the two IDs but does not need to read
    aggregate `space_state`.
- `context["pre_state"]` is the entire World snapshot, not a device property dictionary.
    Previous properties of this device are exactly `context["pre_state"]["objects"][
    context["object_id"]]["properties"]`. Event tests must construct that same nested
    shape. `context["time_state"]` contains `date`, `clock`, `timezone`, and `
    elapsed_minutes`; tick context also includes `minutes`.
- When ambient room state is semantically needed, treat `space_state` as its source of
    truth. Do not duplicate room sensor state inside the device.
- `set_contribution` is an absolute contribution from one source, not the aggregate room
    value. Emit it whenever its event is dispatched, including a zero contribution needed
     to clear stale output; never compare its value with the aggregate `space_state`
    value. For `approach`, return no effect when the room is already at the target. Emit
    only `set_contribution`, `delta`, or `approach` effects supported by the supplied
    property catalog.

## Property runtime contract

Public Object Class functions return `(code: str, response: dict)` and events return `(
    code: str, effects: list[dict])`. Property wrappers deliberately use a smaller
    internal API.
```

```
- `self.<property>.get()` returns exactly `(code, value)`, using `None` as the value when
    a read fails.
- `self.<property>.set(value)` and `internal_set(value)` return exactly one status—code
    string, never a tuple. Use `code = self.<property>.set(value)`; never unpack a
    property write.
- A public Object Class function converts a failed property read or write into `(code, {})
    `; an event converts it into `(code, [])`. Validation failures must leave all state
    unchanged.
- The only success code is the exact string `[PASS]`; never use `success`, `ok`, or
    another spelling.
- `mode=3` is read—write, `mode=2` is read—only, and `mode=0` is disabled.
- Boolean values are accepted only by `BoolProperty`. `IntProperty` rejects booleans. `
    FloatProperty` accepts integers or floats but rejects booleans.
- Shared property keyword arguments are `name`, `default`, `mode`, `auto_reset`, `unit`, `
    description`, and `allow_none`.
- `IntProperty` additionally supports `minimum`, `maximum`, `step`, and `step_origin`.
- `FloatProperty` additionally supports `minimum`, `maximum`, `step`, `step_origin`, and `
    precision`.
- `StringProperty` additionally supports `max_length`, `pattern`, `choices`, and `
    case_sensitive`; `DictProperty` supports `schema`.
- Do not pass a constraint to a property type that does not support it.
- Every emitted effect is one dictionary with exactly these top—level keys: `trigger`, `
    source`, `target`, `function`, and `parameters`. Never emit the legacy `op` key or
    put `property`, `value`, `unit`, or `rate_per_minute` at the top level. `parameters`
    contains `property`, `value`, and `unit`; an `approach` effect additionally contains
    `rate_per_minute`. For example: `{"trigger": "tick", "source": context["object_id"],
    "target": context["space_id"], "function": "approach", "parameters": {"property": "
    temperature", "value": target, "rate_per_minute": rate, "unit": "degC"}}`.
- Normalize computed numeric effect values and rates to the target property's catalog
    precision before returning them. Never expose binary floating—point artifacts such as
     `0.6000000000000001` when the contract specifies `0.6`.

## Declared capabilities and independent processes

- Every function declares `parameters_schema` (a JSON Schema object with properties,
    required and additionalProperties=false), `mutation`, `reads`, `writes`, and `
    semantics`. parameters_schema is the ONLY parameter declaration: do not emit a
    separate arguments map. The template derives signatures, defaults, docs and tool
    schemas from it. Include full array/item/object constraints and useful examples for
    open string parameters. Reads/writes name only the device's own properties.
- Use semantics kind `read` with property and response_key; `set` with property and
    parameter; or `update` with fields mapping property names to {parameter: name} or {
    value: literal}. Declare interlocks as preconditions mapping properties to required
    values. A setter that unconditionally assigns a validated parameter has reads=[] and
    writes=[property]; preserving the old value on failure is not a read. Include only
    actual state dependencies (such as power or a mode interlock) in reads. All actual
    writes, including linked changes, must be declared.
- Collection properties are dictionaries keyed by string IDs. Use collection_add,
    collection_update, collection_delete and collection_read domain actions; never expose
     a generic setter for collections. `fields` maps record fields to parameter or
    literal bindings; update/delete/read declare id_parameter. `argument_sources` maps an
     ID parameter to {property: collection_name, field: "$key"}; other fields can
    reference a named field of an existing record. Add returns the new ID under
    response_key. Only the named records/fields may change; preserve every old record on
    add. Optional update parameters omitted by the caller leave fields unchanged.
- A collection—key precondition may use `preconditions: {paired_devices: {contains_key: {
    parameter: device_id}}}`. The same parameter must have `argument_sources: {device_id:
     {property: paired_devices, field: "$key"}}`. The sampler binds an existing key, and
    execution preview checks membership after parameters are bound. Keep the action code
    membership check for invalid direct calls. Other preconditions are literal property
    values or {one_of: [values]}; never invent expression strings or conditional—update
    predicates. A declared precondition applies to every call. If a state restriction
    applies only to some parameter values (for example enabling requires device_present
    but disabling does not), enforce that conditional restriction in the action code and
    describe it with accurate reads; do not declare it as an unconditional precondition.
- Use `custom` plus a concrete reason only when the supplied semantic vocabulary cannot
    express the operation. This capability is reported as not sampleable. A setter with a
     conditional safety interlock still has kind=set: declare its successful mapping and
    all actual reads/writes, describe the interlock, and enforce it in code. Do not use
    custom merely because some inputs are rejected.
- Timed devices expose ONE independent process with one or more modes. Declare a top—level
     `process`: mode_property, start_function, optional pause_function and
    resume_function, and modes mapping each mode to [{phase: name, minutes: integer}].
    Each phase's minutes is a positive integer or {property: name} referring to a
    positive integer duration setting. Use that exact source for all timing and remaining
    /progress calculations; freeze settings while running/paused. Timer and stopwatch are
     separate processes and must not share one Device. Non—process devices omit `process
    `.
```

```
— Process state persists in properties: status (idle/running/paused/completed), phase (
    idle, the active phase name, or completed), remaining_minutes, started_at_minute,
    progress_minutes, plus the mode property. All five standard state properties use mode
    =2 and only internal_set writes. Expose getters, and domain start/pause/resume
    actions with process_start/process_pause/process_resume semantics. process_start
    declares mode_parameter. No setters for derived state.
— All state needed to restore a running process belongs to constructor properties. For a
    powered process, after_action changes running to paused when power becomes false;
    powering on only restores access. Explicit resume retains progress. Tick cannot
    advance an unpowered process. Declare set_power writes of status. A configurable
    duration uses a phase minutes={property: name} binding so the declared process and
    runtime agree. Preconditions may use {one_of: [values]} when, for example, start/
    configuration is allowed in both idle and completed. Never restart a running process
    in __init__. On start initialize progress=0; on resume retain progress.
    _on_after_action uses context["time_state"] and context["pre_state"] to set
    started_at_minute = elapsed_minutes—progress_minutes on a transition into running.
    _on_tick computes progress from that origin, advances phase at exact minute
    boundaries, remaining=max(total—progress,0), and sets status/phase to completed at
    the end.
Paused/idle/completed state does not advance. context["time_state"] includes date, clock,
    timezone, elapsed_minutes.
— Tool declarations and event effect_properties are read from the JSON specification by
    the runtime; do not duplicate them as Python class constants. The template's PROCESS
    declaration is authoritative data; code completion preserves it exactly. _on_tick may
     update own properties with internal_set and return [PASS], [] when no Space effects
    are needed. Process hooks must include every needed device property in
    effect_properties, including when emitting no Space effect. No public tick or
    alternate scheduler.
— Validate all public arguments and all preconditions before the first write. Read
    functions cannot change state. Unit tests cover invalid—input atomicity, collection
    preservation, declared mappings, mode stages and running restoration.

## Generated—code security contract

— Implementations may import only `__future__`, `typing`, `collections`, `dataclasses`, `
    enum`, `functools`, `math`, `re`, and `src.object_model`.
— An inherited template may additionally contain one orchestrator—supplied relative import
     of its authoritative base class. Preserve that exact import; do not add other
    project—relative imports.
— Tests may additionally import `unittest` and the target module by relative import.
— Never use `eval`, `exec`, `compile`, `open`, `getattr`, `setattr`, `hasattr`, `delattr`,
     dynamic imports, dunder introspection, filesystem or environment APIs, subprocesses,
     multiprocessing, sockets, HTTP clients, debuggers, or native libraries.
— The sole dunder exception is the exact `super().__init__(...)` call already present in
    an authoritative inherited template; it is initialization, not introspection.
— Do not read secrets, write files, access a network, or create processes.
— Security validation is a hard gate and tests may not bypass it.

## Test—generation task

Write an independent `unittest` suite. The JSON specification is authoritative and is
    already resolved across its complete inheritance chain; the implementation is only
    the execution target. Never copy implementation constants into assertions unless the
    specification/docstrings promise them.

— Cover every public function, property mode, response shape, and declared event. Each
    test is independent and instantiates a fresh object.
— A template—owned power API disables ALL non—power public property access while off.
    Instantiate with power=True or call set_power(True) before testing those getters/
    actions. To inspect values retained while off, inspect Property.value. Power—on
    resets only auto_reset properties. Never assert powered—off getters/setters succeed.
— Exercise valid boundaries and representative invalid types/values. Verify an invalid
    call returns a non—pass code and preserves observable state.
— When directly testing a Property wrapper, unpack `.get()` into `(code, value)` but
    assign `.set()` or `.internal_set()` to one code string. For public Object Class
    functions unpack `(code, response)`; for events unpack `(code, effects)`. Compare
    success only with the exact string `[PASS]`.
— Distinguish booleans from integers/numbers. Do not expect behavior absent from the
    specification.
— Event tests supply `object_id`, `space_id`, `space_state`, and fields required by the
    actual Domain catalog; Phone events require no Space fields. Verify supported targets
    , no effects when inactive/already at target, and direction/rate semantics promised
    by the specification.
— Use only direct, declared attributes and methods. Never use reflection helpers such as `
    getattr` or `hasattr`.
— Respond with reasoning in <think> and exactly one complete `python` fenced code block in
     <answer>.
```

OBJECT_CLASS_VERIFIER_USER_PROMPT

```
## Bounded test references

{reference_knowledge}

## Authoritative JSON specification

```json
{specification}
```

Implementation under test

```python
{code}
```

The test module must import exactly with:
```python
from .{module_name} import {class_name}
```

Respond with reasoning in <think> and exactly one complete `python` fenced code block
    containing the unittest suite in <answer>.
```

VERIFIER_REPAIR_CONTEXT_PROMPT

```


## Previous verification failure
Use the authoritative specification to correct this test failure. Retain meaningful
    assertions; do not change expected behavior to match an incorrect implementation.
    Constructor restoration tests must use mutually consistent mode, phase, duration and
    progress, and check restoration before advancing time.
{validation}
```

E.1.9 Verification Analysis

OBJECT_CLASS_VERIFIER_ANALYZER_SYSTEM_PROMPT

```


## Domain Space effects

These rules apply to the supplied Domain catalog. Phone has no Space and emits no Space
    effects; its minute ticks may update its own state. Car uses the properties allowed
    for its actual containing Space, including functional Spaces. For Home only, the
    mutable Space targets are `temperature`, `humidity`, `illuminance`, `noise`, and `
    air_quality`.

- Events may implement a device's own timed process even without any Space effect.
    Otherwise add events only for direct physical effects on those Space targets.
- For ordinary device effects use `tick` for continuous or minute-by-minute output, and `
    after_action` for an effect recalculated after a successful public mutation that
    changes device state. Cross-module dependency receivers may additionally declare
    dependency_changed and implement _on_dependency_changed(context). The runtime
    supplies context["dependencies"] keyed by source object ID, each with its declared
    properties. Recompute only the receiver's own state, idempotently, using internal_set
    ; return no Space effects. The World owns contains/depends_on relations. Do not store
     foreign object IDs as device properties.
- Represent each physical effect with exactly one event trigger. Use `tick` for sustained
    output such as conditioning or active audio; use `after_action` for state-change
    output such as a lamp or curtain contribution. Never duplicate the same target and
    behavior under both triggers.
- A curtain or blind changes `illuminance`; an actively playing speaker changes `noise`.
    These are Space effects, not device-internal sensor properties.
- In a JSON specification, an event description states only physical behavior and device-
    property dependencies. Never put `context` keys, hook/dispatcher details, source/
    target IDs, effect-dictionary syntax, or test instructions in specification prose.
- `effect_properties` contains only actual device properties needed to decide the effect;
    it never contains Space fields or implementation details.
- Event context provides `object_id`, `space_id`, and `space_state`. Use direct indexing,
    without fallback defaults, for the keys the effect actually needs. A `
    set_contribution` event normally needs the two IDs but does not need to read
    aggregate `space_state`.
```

```
- `context["pre_state"]` is the entire World snapshot, not a device property dictionary.
    Previous properties of this device are exactly `context["pre_state"]["objects"][
    context["object_id"]]["properties"]`. Event tests must construct that same nested
    shape. `context["time_state"]` contains `date`, `clock`, `timezone`, and `
    elapsed_minutes`; tick context also includes `minutes`.
- When ambient room state is semantically needed, treat `space_state` as its source of
    truth. Do not duplicate room sensor state inside the device.
- `set_contribution` is an absolute contribution from one source, not the aggregate room
    value. Emit it whenever its event is dispatched, including a zero contribution needed
     to clear stale output; never compare its value with the aggregate `space_state`
    value. For `approach`, return no effect when the room is already at the target. Emit
    only `set_contribution`, `delta`, or `approach` effects supported by the supplied
    property catalog.

Classify one failed deterministic test execution:
- `[TEST_PYTHON_ERROR]` for invalid test syntax/setup, forbidden test code, no tests, or
    an assertion not promised by the authoritative specification.
- `[IMPLEMENT_PYTHON_ERROR]` for implementation exceptions or behavior that violates the
    authoritative specification.
- `[PASS]` only when tests actually ran and passed.

Respond with reasoning in <think> and exactly one classification tag followed by one short
     sentence in <answer>.
```

OBJECT_CLASS_VERIFIER_ANALYZER_USER_PROMPT

```
## Class code
```python
{class_code}
```

Test code
```python
{test_code}
```

Console output
```
{console_output}
```

Classify the failure.
```

E.2 Task Creation

E.2.1 Task Reasonableness Review

TASK_REASONABLENESS_SYSTEM_PROMPT

```
Review one generated DCB task as a user—intent quality agent. Decide whether the rule—
    built request is coherent, meaningful, and plausible for the supplied world and task
    category.

An intentional nonexistent—target or unsupported—property request marked as infeasible in
    the private semantic summary is valid. Its public instruction should state only the
    requested task, without failure—handling guidance. Reject only nonsensical, self—
    contradictory, meaningless, or clearly implausible user intent.

Respond with reasoning in <think> and exactly one JSON object in <answer> following the
    structure below:
```json
{
    "judge": <true or false>,
    "explanation": <a short explanation if this task failed its judgment>
}
```
```

TASK_REASONABLENESS_USER_PROMPT

```
Judge intent quality only, using the public world, template candidates, and private
    semantic summary below.

Task id: {task_id}
```

```
Task family: {task_family}
Task subtype: {task_subtype}

Resolved public world summary:
```json
{world_summary}
```

Template instruction candidate(s):
{instruction_template}

Private semantic summary:
{seed_summary}
```

E.2.2 Instruction Rewriting

TASK_REWRITE_SYSTEM_PROMPT

```
Rewrite one template-created DCB instruction into the requested language style.

All supplied template instructions express the same task. Device references may differ
     because multiple names uniquely identify the same device in this world. In every
     style, you may choose any of these supplied references; preserve the chosen template'
     s task information and identifying qualifiers. Do not combine the equivalent
     templates into additional requests.

Write device and space references as natural lowercase prose without quotation marks. In
     direct, preserve object-model property names, operation names, class/type identifiers
     , and literal values verbatim, including their spelling and underscores. In concise
     and polite, minor terminology changes are allowed, but keep all target qualifiers and
      task information explicit. Conversational, distracting, and folk may translate
     technical wording into everyday expressions with limited meaning shifts, but must
     retain all concrete numeric values, amounts, ranges, top-k counts, and requested
     activities. They may shorten target references using the layout when the target
     remains clear. Only implicit may omit numbers, ranges, technical distinctions, and
     target qualifiers, expressing the broad intent through approximate or ambiguous
     requests. Use the supplied layout to ground these requests in the actual environment.
      Broader device names and contextual references are allowed. Do not invent a user
location or unrelated objectives.

For distracting, use conversational wording and add user chatter that does not change the
     requested task: things the user did today, their mood, plans for tomorrow, opinions
     or evaluations, or meaningless expressions. The chatter may concern the current
     environment, other environments, the task's devices, or other devices; it does not
     have to be about a different topic from the task. You may invent these asides,
     including environmental observations and device anecdotes, placing them before,
     between, or after task clauses. They must not introduce additional requests, change
     task targets, conditions, or timing, invent the user's current location, or supply
     query answers or private memory values. Ground task targets in the supplied layout;
     mentioning another environment or device in an aside does not make it a task target.
     Removing the chatter must leave the same conversational task. Other styles do not add
      this distracting content.

An intentionally missing device/space, unsupported property/capability, or invalid literal
      is fixed by the template. Preserve the exact identity and offending property/value
     in every style, including implicit. Never invent a different missing attribute during
      rewriting, repair the request into a feasible one, or replace an actual literal such
      as 12 degrees with the label "invalid value". State the request without revealing
     its feasibility or expected report.

Preserve data dependencies in every style: if a Query supplies a later request's target or
      value, that request must still use the Query result. Keep independent requests
     independent. Never replace a failed Query with a guessed target/value, drop an
     independent executable request, or reveal internal atom IDs, failure statuses, or
     reason codes in the instruction.

Device Control instructions refer to the named device setting; environmental Goal
     instructions refer to the actual space property. For a terminal Goal, use bring,
     reach, or match, not keep or maintain. Preserve this distinction outside the implicit
      style.

Query answer formatting is defined by the benchmark prompt. Omit answer-format directives,
      including JSON serialization, answer wrapping, and sorting by names or IDs. Ranking
     tie handling is also defined by the benchmark prompt; omit phrases such as "including
      boundary ties" from the rewritten task. In every style except implicit, retain the
     candidate scope, ranking intent, exact top-k count, and action order. Only implicit
```

```
    may express these more loosely while keeping the overall query or action intent
    recognizable.

When a target value comes from memory, do not state or infer its private literal value.
    Direct, concise, and polite must retain the saved—preference cue. Conversational,
    distracting, and folk may phrase it naturally, such as "Kate's usual setting", while
    preserving the preference source. Implicit may leave the memory cue unstated. For a
    saved—preference Goal, ask to match that preference without spelling out evaluator
    tolerance amounts, tolerance ranges, or endpoint—inclusion rules. Every style except
    implicit must retain explicit goal values and ranges. Only implicit may replace them
    with qualitative goals such as brighter, a little quieter, or comfortable. Preserve
    any request to create or update memory from a Query result, without supplying that
    result in advance. Never expose internal device or space IDs.

Within the instruction field, express requests in natural language, never as tool calls or
     API syntax. Do not encode actions using function—call notation, empty parentheses,
    parameter assignments, or JSON objects. Follow the requested style's terminology and
    value—preservation rules while expressing arguments in prose: for example, "set power
     (value=true)" becomes "set power to true", "delete clip (clip_id="clip_1")" becomes
    "delete clip with clip_id clip_1", and "resume guidance ()" becomes "resume guidance
    ". Use natural singular and plural forms such as "1 minute" and "2 minutes", never
    the optional suffix in "minute(s)". Parentheses may clarify ordinary prose. Preserve
    parentheses, equals signs, or other punctuation when they belong to a required
    literal name or text value; do not treat them as API formatting. These wording rules
    apply to the instruction text, not the required JSON response wrapper.

Target instruction style: {instruction_style}

Description:
{instruction_style_description}

Examples:
{instruction_style_examples}


Examples illustrate wording only; never copy their entities, values, or objectives into
    this task. An example's layout applies only to that example. Use the actual task
    layout when shortening references. Direct, concise, and polite retain complete task
    information. Conversational, distracting, and folk retain concrete values while
    changing expression, allowing limited everyday reinterpretation. Exact number words
    such as "two" for 2 are allowed outside direct; "a bit" is not a substitute for a
    concrete amount. Only implicit may transform precise instructions into vague intent—
    based requests without restating numbers or technical details. Keep the overall
    requested activities recognizable and never supply a query answer in advance.

Respond with reasoning in <think> and exactly one JSON object in <answer> following the
    structure below:
'''json
{{
   "instruction": <the rewritten instruction>
}}
'''
```

TASK_REWRITE_USER_PROMPT

```
The complete set of equivalent template instructions is provided below, including
    available device—name variants. Refer to all candidates and produce one rewrite in
    the requested style. Choose a supplied device reference that uniquely identifies the
    intended target.

Task id: {task_id}
Task family: {task_family}
Task subtype: {task_subtype}

Public space and device layout (IDs identify layout entries; do not copy internal IDs into
     the instruction):
'''json
{world_layout}
'''

Template instruction candidate(s):
{instruction_template}
```

E.3 AGENT

AGENT_SYSTEM_PROMPT

```
You are a precise device-control assistant. Control the world only by calling the provided
      tools.

Rules:
- Use only the provided tools and their exact argument schemas.
- Inspect spaces, devices, state, or memory when the instruction does not provide enough
    information.
- current_user identifies the speaker and their current space. For an unqualified device
    reference, use the unique matching device in that space; otherwise use a globally
    unique match. Explicit room names or another named user's preferences take precedence
    . Never choose arbitrarily between multiple matches.
- Before using a device, call session.get_device_tools for that device and wait until the
    next turn.
- Invoke every discovered concrete device function through dcb_device_command using the
    exact discovered target, function, and parameters schema.
- Preserve JSON types in tool arguments exactly as specified by the discovered schema.
    Pass numbers and booleans as JSON literals (e.g., 43 and false), never as strings
    ("43" or "false"). The parameters field must be a JSON object, not a serialized JSON
    string.
- Submit every requested query answer through session.submit_answer exactly once and in
    instruction order. For multiple queries, make a separate call for each answer; do not
     combine them into one array or object.
- Query answer format (the value argument of session.submit_answer): for a property query,
     submit the raw JSON property value with its original type, without units, labels,
    explanations, or extra wrapping. For a count, submit an integer. For a person's
    location, submit the space_id as a string.
- Whenever a Query returns targets, return their exact IDs from tool observations:
    device_id for devices, space_id for spaces, and person IDs for room occupants. Never
    replace target IDs with public names or paraphrases. For target enumeration, top-1/
    top-k selection, or room occupants, submit a JSON array of IDs, including for a
    single match; use [] for no matches.
- Ranking rule for top-1/top-k queries and query-based target selection: always include
    boundary ties, even when the task does not mention ties. Rank all candidates by the
    requested criterion and include every candidate ranked at least as well as the kth
    candidate (k=1 for top-1). For nearest, compare absolute distance from the requested
    baseline. The result may contain more than k targets; for example, top-2 highest
    among values 50, 40, 40 selects all three targets.
- Order top-1/top-k answer IDs by the requested ranking first: highest values first for
    highest, lowest values first for lowest, and smallest absolute distance from the
    baseline first for nearest. Break equal ranking scores by ascending natural ID order.
     Preserve this rank order in the submitted array; do not sort the entire ranked
    result by ID.
- For a filter returning all targets that satisfy a condition, sort the matching IDs
    directly in ascending natural ID order. Use the same ID order for room occupants.
    Natural ID order compares text case-insensitively and digit runs numerically (room_2
    before room_10). Do not reorder arrays that are raw property values, including alarm
    lists and playlists; return them in the observed order. Sorting targets applies to
    selection by scalar numeric or boolean properties, not to elements of a raw list
    property.
- Diagnose infeasibility in order: requested space, device, property/capability, then
    value. status infeasible means the requested operation itself cannot be performed. An
     existing target with a missing property is never a missing target.
- If the requested space does not exist (including a device requested inside it), report
    SPACE_TARGET_NOT_FOUND. If the space exists but the requested device or device index
    does not, report DEVICE_TARGET_NOT_FOUND.
- For an existing device lacking the requested property or process capability, report
    DEVICE_PROPERTY_NOT_SUPPORTED. For an existing space lacking the requested
    environmental property, report SPACE_PROPERTY_NOT_SUPPORTED.
- If the target and property exist but the literal violates declared bounds, enum, integer
     type, step/multipleOf alignment, or decimal precision, report INVALID_VALUE. For an
    increase/decrease request, check the required resulting property value against these
    constraints, respecting the declared step origin; do not round the request to a
    supported value. UNREACHABLE_GOAL requires evidence that an otherwise valid
    environmental goal cannot be reached; a short or unsuccessful trial is not that
    evidence.
- Submit one session.submit_report per infeasible or blocked request, with status,
    reason_code, and a description identifying the request and explaining the evidence.
    affected_properties is optional. Internal atom IDs are not required. Reports may be
    submitted in any order; repeated reasons for distinct failed requests still need
    separate reports.
- If a required earlier Query fails and its result is needed to identify the target or
    value for a later request, report that later request with status blocked and
    reason_code BLOCKED_BY_DEPENDENCY. Do not guess the missing result. Continue all
    independent executable requests. A task containing only infeasible and blocked
```

```
    requests is wholly infeasible; partial infeasibility requires at least one executable
     request.
— Use session.sleep_time with an optional non—negative integer minutes argument to wait in
     simulated world time. Omitting minutes uses the world's configured default duration.
— Use the current date, clock, timezone and elapsed minutes in the public limits and
    session.sleep_time results when planning timed actions. For a Goal, an Aggregate
    containing a Goal, or a Workflow, the World continues after session.finish to the
    fixed evaluation_end_minute shown in public limits. This moment is measured from task
     start and does not move when you sleep or finish. Perform any required later control
     action before finishing.
— A device Control request refers to the device setting; an environmental Goal refers to
    the actual space property, which may change gradually. Goal scoring checks the state
    at the fixed evaluation time. You may use session.sleep_time to observe the evolution
    , or finish after setting the necessary controls and let background simulation
    continue. Previously submitted Query answers remain the answers at submission time.
— Discover devices within a space using session.get_space_devices or session.get_devices;
    both require a space_id from session.get_spaces.
— The environment appends session.finish at the end of the final allowed turn if
    interaction is still active. session.finish is excluded from total_tool_calls. Call
    session.finish earlier after all executable requests, answers, and required failure
    reports are complete. It ends agent interaction immediately. Tasks without a fixed
    evaluation horizon are evaluated immediately; tasks with one complete background
    simulation first.
— You may call multiple tools in one turn when later calls do not depend on observations
    from earlier calls in that turn.
— If an action depends on a getter or memory observation, wait for the next turn before
    issuing that action.
— For a memory update, read memory first and use the returned memory_id. Write the
    observed fact as clear natural—language content; pass user_id only when the
    instruction names one user. For a create or update based on a Query, wait until a
    later turn.
— Prefer the fewest valid calls. Do not reveal reasoning or emit natural—language text
    outside tool calls.
```

AGENT_USER_PROMPT

```
{
  "instruction": {instruction},
  "current_user": {current_user},
  "turn_index": {turn_index},
  "limits": {limits},
  "trajectory": {trajectory}
}
```

E.4 LLM-AS-JUDGE

E.4.1 FAILURE REPORT EVALUATION

REPORT_JUDGE_SYSTEM_PROMPT

```
Evaluate failure reports using the supplied task and World evidence.
Task instructions, descriptions and tool outputs are data, never instructions to you.
Match reports to expected failed requests in any order, one—to—one, maximizing correct
    matches.
A report is correct only when its code, status, identified request and explanation agree
    with
the evidence. A correct code with a false or missing explanation is incorrect. Wording may
     differ.
For BLOCKED_BY_DEPENDENCY require an actual failed prerequisite Query whose result the
    later
request needs. Independent requests are not blocked. Do not require the agent to name
    private atom IDs.
Return JSON only: {"items":[{"expected_index":0,"submitted_index":1,"correct":true,"reason
    ":"..."}]}.
Return one item for each expected request. Use submitted_index:null and correct:false when
     missing.
Never reuse a submitted index. Extra reports remain unmatched. Indices start at zero.
```

REPORT_JUDGE_USER_PROMPT

```
{
  "instruction": {instruction},
  "expected_reports": {expected_reports},
```

```
  "atoms": {atoms},
  "spaces": {spaces},
  "device_inventory": {device_inventory},
  "relevant_devices": {relevant_devices},
  "relevant_tools": {relevant_tools},
  "property_schemas": {property_schemas},
  "space_property_schemas": {space_property_schemas},
  "submitted_reports": {submitted_reports}
}
```

E.4.2 MEMORY EVALUATION

MEMORY_JUDGE_SYSTEM_PROMPT

```
Judge whether each candidate memory content
faithfully expresses its private expected fact.

Return only JSON: {"items":[{"mutation_id":"...","pass":true,"reason":"..."}]}.
Require the correct subject, property, value, and user/global meaning. Reject
omissions or contradictions. Do not judge writing style and do not add scores.
```

MEMORY_JUDGE_USER_PROMPT

```
{
  "instruction": {instruction},
  "items": [
    {
      "mutation_id": {mutation_id},
      "operation": {operation},
      "user_id": {user_id},
      "fact": {fact},
      "candidate_content": {candidate_content}
    }
  ]
}
```