BEYOND ARTICULATION: INTRINSIC OPERATIONAL LOGIC FOR INTERACTIVE 3D ASSETS

Anonymous authors
Paper under double-blind review

ABSTRACT

Simulating everyday object use requires operational behavior: rules governing how objects respond to manipulation over time. Authoring these rules is costly, and simulator-specific implementations limit reuse. We introduce Intrinsic Operational Logic (IOL), a declarative representation for portable behavior execution and reliable authoring by large language model (LLM) agents. An LLM specifies state predicates, temporal rules, and motion effects; automatic synthesis constructs the controller, and lightweight adapters connect the controller to different simulators. The controller also guides interaction-trace-based test creation; execution is compared with the LLM's expectations to diagnose and repair errors. On AuthorBench, our benchmark of 50 objects, IOL improves behavioral accuracy over procedural code; trace-based repair further improves over standard agentic self-testing. IOL-equipped objects also enable OperateBench, our benchmark of 25 manipulation tasks. GPT-6 Astra solves 92% of tasks in a privileged-state workflow and 56% in a separate workflow using simulated RGB-D observations, highlighting the challenge of complex object operations from visual observations.

1 INTRODUCTION

To train and evaluate robots on everyday object use, simulated assets must reproduce how objects respond to manipulation. Formats such as URDF describe how an object's parts are connected and how each joint can move ROS 2 Documentation (2026). For a toaster, this data specifies how the lever slides and the timer knob rotates, but not that the lever stays down during a timed cycle and rises when it ends. We call such rules *operational behavior*: how an object's motion depends on its controls, state, and time. Operational behavior often lives in simulator-specific scripts or task code Srivastava et al. (2022); Li et al. (2023). Moving an asset to another simulator then requires reimplementing and testing its behavior. A portable specification would let this behavior be authored once and reused with the asset.

Authoring costs also make acquiring operational behavior at scale challenging. Recent work uses large language models (LLMs) to construct articulated object models by generating code Zhao et al. (2025), with agents inspecting simulated outputs and iteratively correcting errors Le et al. (2025). These write–test–revise loops suggest an automated route to acquiring operational behavior. Alongside simulator portability, we seek an operational behavior representation that LLM agents can reliably author.

We introduce *Intrinsic Operational Logic* (IOL), a declarative representation designed around these goals (Figure 1). Implementing behavior rules directly in procedural code requires an LLM to coordinate conditions, persistent states, and timing in monolithic, highly branching logic. IOL cleanly separates the task into three parts: Boolean predicates that the LLM implements to detect conditions in the joint state; temporal rules relating those conditions to operating modes; and numerical effects that the LLM implements to change joint motion when a mode is active. For the toaster, a predicate detects timer expiry, a rule specifies when the cycle is running, and an effect holds the lever down while it runs. We restrict the rules to a subset of linear temporal logic Pnueli (1977); Bloem et al. (2012), allowing us to automatically synthesize a deterministic finite-state controller for each realizable specification. The controller, predicates, and effects use asset state rather than simulator APIs; lightweight adapters execute the same behavior in simulators such as SAPIEN Xiang et al. (2020) or Isaac Sim NVIDIA (n.d.).

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.

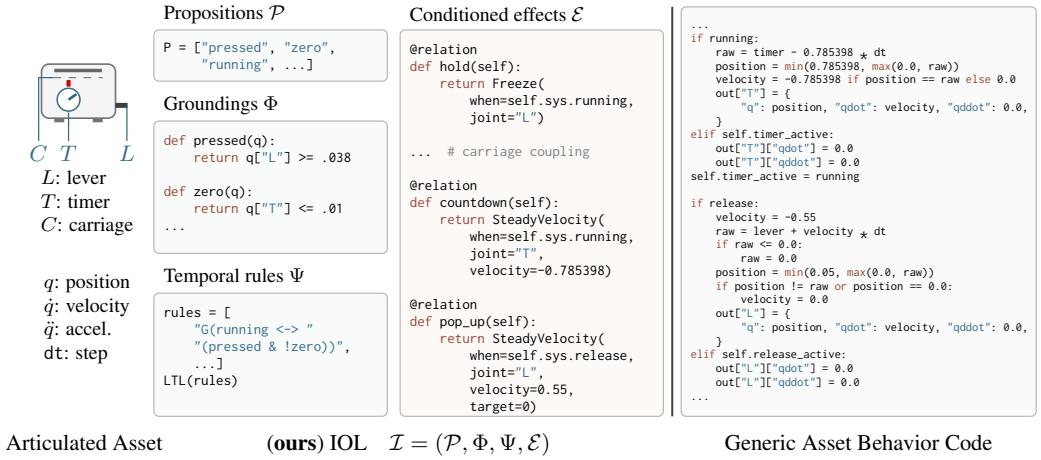


Figure 1: **IOL is a better authoring target for LLM agents.** The agent specifies predicates, temporal rules, and conditioned effects; an automated synthesizer generates a corresponding controller from the temporal rules. In contrast, direct implementation often involves authoring hard-to-debug branching code. IOL makes the intended behavior explicit and supports trace-based testing and correction, helping the agent refine its specification. Simplified Python excerpts are shown; ellipses mark omitted code.

IOL also structures how LLM agents test and correct their behavioral rules. Like unit tests for code, *interaction traces* check concrete behavioral expectations: each trace proposes a sequence of joint motions from an initial object state. The compiled controller guides the construction of traces that systematically probe the authored rules alongside the agent's self-written tests. The LLM judges which traces should be allowed, using asset evidence and its understanding of object semantics. It then executes them using the compiled controller and compares its responses with those expectations. For example, the LLM may expect a toaster lever to stay down during a cycle, while execution releases it early. Disagreements and per-step joint responses help the agent locate errors in its predicates, rules, or effects. It revises the specification and replays the traces to check the repair.

We introduce *AuthorBench* to evaluate whether IOL helps LLMs acquire correct operational behavior. It provides human-authored reference behaviors for 50 objects, against which we compare generated behavior on held-out interaction traces. We compare IOL with procedural Python code that either uses simulator-specific APIs or directly computes joint-state updates through a simulator-independent function. IOL recovers reference behavior more accurately than either baseline. In our main Sol/ultra comparison of end-to-end authoring workflows, trace-based correction achieves higher trajectory match than agentic self-testing.

Objects with IOL specifications also enable *OperateBench*: 25 simulated manipulation tasks that require agents to respect operational behavior rules while using objects (e.g. to heat bread in a toaster, an agent must coordinate bread placement with the timed cycle). Agents write policies as code Fu et al. (2026), combining manipulation skills with conditional decisions and feedback. GPT-6 Astra solves 23 of 25 tasks (92%) in a privileged-state workflow and 14 (56%) in a separate RGB-D-based workflow. These results motivate further work on integrating perception, operational reasoning, and physical control for reliable object use.

Our contributions are:

1. **Intrinsic Operational Logic**, a portable representation of object behavior with automatic controller synthesis and trace-based tools for LLM testing and repair.
2. **AuthorBench**, evaluating LLM acquisition of operational behavior against reference behaviors for 50 objects.
3. **OperateBench**, evaluating embodied object use on 25 manipulation tasks governed by IOL specifications.

2 Related Work

Operational behavior and embodied benchmarks. Household and manipulation benchmarks cover diverse object interactions, including appliance use and articulated-object manipulation (Li et al., 2023; 2022; James et al., 2020; Gu et al., 2023; Nasiriany et al., 2024; 2026; Shridhar et al., 2020; Gong et al., 2023; Padmakumar et al., 2022). BEHAVIOR-1K/OmniGibson already models operational dependencies through object states and conditional transition rules, including prerequisites for appliance use (BEHAVIOR, 2026). We study how such behavior can be represented for reuse across simulators and effectively authored by LLM agents. IOL combines a portable asset-level specification with automatic controller synthesis and trace-based testing tools. OperateBench uses these assets for a focused evaluation of operational object use. Appendix F.6 compares the scope and representations of additional systems.

Acquiring articulated and functional assets. Real2Code reconstructs articulated objects through code generation (Zhao et al., 2025), while Articulate-Anything uses an actor–critic loop to propose and refine articulation models (Le et al., 2025). These methods motivate LLM-based asset acquisition; we study acquiring operational rules beyond joint structure. EgoFun3D infers executable receptor–effector templates from egocentric video (Peng et al., 2026). Its implemented templates model individual part relationships, without synthesizing controllers from temporal constraints across controls and operating modes. IOL supports these coordinated rules—such as retaining a toaster lever during a countdown and releasing it at expiry—and uses the synthesized controller to guide trace-based testing and repair. We evaluate operational-rule authoring for existing articulated assets against procedural implementations with matched inputs; EgoFun3D targets reconstruction and template inference from video.

Execution-guided LLM authoring and repair. LLMs can improve generated programs by inspecting execution feedback and revising their code. Self-Debugging combines execution with explanations of the generated program (Chen et al., 2024); Reflexion uses feedback and verbal reflection to improve subsequent attempts (Shinn et al., 2023). We build on iterative authoring and ask how the representation and testing tools affect recovery of object behavior. IOL automatically generates interaction traces whose execution can be compared with the LLM's expectations from prior knowledge. Disagreements provide concrete tests for revising the specification.

Declarative specifications and reactive synthesis. Formal methods let software engineers state behavioral requirements explicitly and check software designs against them (Newcombe et al., 2015). Linear temporal logic (LTL) describes how conditions relate over time (Pnueli, 1977), and GR(1) synthesis constructs reactive controllers satisfying specifications in a restricted subset of LTL (Bloem et al., 2012). These techniques have also been used to synthesize robot mission and motion controllers (Kress-Gazit et al., 2009). IOL applies established synthesis techniques to an object's operational rules represented as LTL formulas. The synthesized controller selects effects on simulator-proposed joint updates, rather than selecting robot actions. This declarative structure supports behavior execution and automatic test construction for LLM-based revision.

3 Method

We aim to equip articulated assets with operational behavior authorable by LLM agents and reusable across simulators. We first describe how this behavior mediates simulator-proposed joint updates, then introduce IOL as a declarative specification of that mediator. Automatic controller synthesis makes the specification executable and structures trace-based testing and repair (Figure 2). Finally, we introduce benchmarks for evaluating behavior acquisition and embodied object use.

3.1 Mediating Simulator Updates

In our framework, the simulator proposes joint-state updates at each time step. An object-local *update mediator* takes the current state and this proposal, then determines the next state under the object's operational rules. For a toaster, it keeps the lever down during a timed cycle while advancing the countdown. This reuses the simulator's physics engine for contact forces, collisions, and robot dynamics while the mediator enforces operational rules.

The mediator uses asset joint names, coordinates, and elapsed time rather than simulator APIs. Lightweight adapters read the simulator state and apply the resulting update. We implement adapters for SAPIEN and Isaac Sim, reusing the same behavior specification. The interface also supports numerical tests without robot rollouts. Execution and adapter details are in Appendices A.1 and A.2.

3.2 Intrinsic Operational Logic

An LLM could implement the mediator directly in procedural code, but would need to coordinate conditions, persistent states, and timing through branching logic. As timing and interlocks interact, these branches become harder to inspect and revise. We draw on formal methods, where engineers check software designs against explicit behavioral specifications Newcombe et al. (2015). IOL separates this task into Boolean predicates that detect joint-state conditions, temporal rules relating those conditions to operating modes, and numerical effects that change joint motion. The LLM writes the predicates and effects in Python; temporal rules are automatically compiled into a controller that decides when the effects apply. An IOL specification (Figure 1) is

$$\mathcal{I} = (\mathcal{P}, \Phi, \Psi, \mathcal{E}), \tag{1}$$

combining grounded *atomic propositions* $(\mathcal{P}, \Phi)$, *temporal rules* Ψ, and *conditioned effects* $\mathcal{E}$.

The vocabulary $\mathcal{P}$ names observed conditions and internal operating modes. Predicates Φ ground observed propositions as Boolean tests of the current joint state. For the toaster, $\mathsf{pressed}$ tests whether the lever is down, and $\mathsf{timerZero}$ tests whether the timer knob has reached its expiry threshold. Temporal rules Ψ specify how operating modes depend on these observations and, when needed, past interactions. For example, the controller's internal mode $\mathsf{running}$ is constrained by

$$\mathbf{G}\big(\mathsf{running} \leftrightarrow (\mathsf{pressed} \wedge \neg\mathsf{timerZero})\big) \tag{2}$$

where $\mathbf{G}$ is the temporal logic operator for "globally" or "always". At every step, the toaster runs exactly when the lever is down and the timer has not expired. A separate rule enables lever release when the cycle ends.

We restrict Ψ to a subset of linear temporal logic (LTL) Pnueli (1977) supported by GR(1) synthesis Bloem et al. (2012). For each realizable specification, synthesis automatically constructs a deterministic finite-state controller, or *Mealy machine* $\mathcal{A}_\Psi$, that satisfies the rules. The author specifies the required behavior without manually enumerating condition combinations and their transitions. At each step, the controller reads the joint-state predicates and uses its stored state to select operating modes and update its memory. Changes to the object's controls therefore change the observed predicates and, through the controller, which effects are active. Its transition structure also guides test construction. Execution semantics and LTL authoring instructions appear in Appendices A.1 and K.2.1, respectively.

Conditioned effects $\mathcal{E}$ are numerical joint-update functions activated by operating modes. For the toaster, the predicates detect that the lever is down and the timer has not expired. The controller activates $\mathsf{running}$, enabling effects that hold the lever down and advance the countdown at a specified rate. At expiry, the changed timer predicate causes the controller to end this mode and enable an effect that returns the lever toward its resting position. Effects can also clamp motion or couple joints. These effects modify the simulator's proposal; the controller determines when they apply. Together, the predicates, synthesized controller, and effects implement the update mediator.

3.3 Agentic Acquisition with Trace-Based Correction

The authoring LLM receives the asset's joint structure and limits, semantic annotations, any supplied operational facts, and rendered views when available. It uses this evidence and its knowledge of the object to write the predicates, rules, and effects, without a reference program.

Source checks and controller synthesis expose construction errors, but an executable specification may still encode the wrong behavior. The LLM may correctly anticipate how an object should respond to an interaction yet implement that response incorrectly. Trace-based correction makes this discrepancy explicit. Alongside the agent's self-written tests, IOL provides controller-guided traces and per-step feedback for comparing the authored behavior with the LLM's expectations (Figure 2).

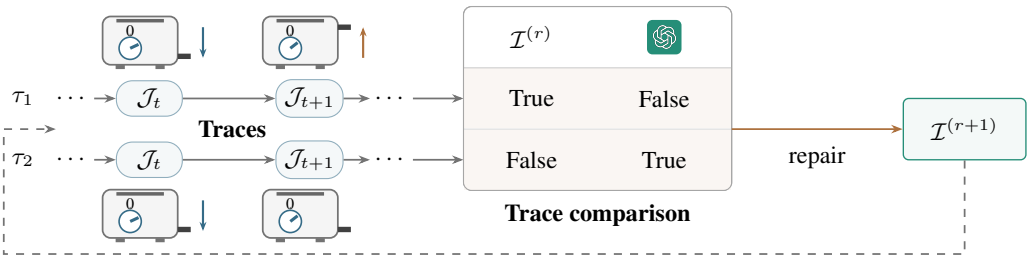


Figure 2: **An illustrative example of trace-based correction.** Trace τ_1 requests release of the toaster lever before the timer expires, while τ_2 requests that it remain pressed. Each $\mathcal{J}_t$ is a requested joint state. Replaying both through the synthesized controller and effects of the current specification $\mathcal{I}^{(r)}$ yields True if all requested positions are preserved, and False otherwise. Disagreements with the LLM's expectations, together with per-step responses, guide revision to $\mathcal{I}^{(r+1)}$. The agent repeats the loop until it chooses to stop or exhausts its budget. See Appendix B for the full procedure.

Constructing interaction traces. A trace specifies a sequence of requested joint states from the asset's rest state. It is *Valid* (True) if replay preserves every requested joint position at every step, including requests to hold still; otherwise, it is *Invalid* (False). Figure 2 shows two examples: after setting the timer and pressing the lever, τ_1 requests an early lever release, while τ_2 requests that it remain down.

We convert the Boolean values of observed APs along sampled controller paths into numerical joint requests. Using the predicate definitions and joint limits, a constraint solver finds joint positions matching the path's required AP values. We supplement these traces with requests to move joints individually or together. The sampler returns numerical sequences without validity labels or reference behavior.

Comparing intended and executed behavior. For each selected trace, the same authoring LLM records an expected label and rationale before the corresponding replay, using asset evidence and its knowledge of the mechanism. In the toaster example, it expects early release to be rejected (τ_1: False) and remaining pressed to be permitted (τ_2: True). An Invalid label can therefore be the correct response to a prohibited motion.

We replay requests through the synthesized controller and numerical effects. The comparison table pairs the resulting labels with the LLM's expectations. This *trace feedback* also includes recorded rationales and per-step joint responses, showing where execution departs from a request.

Repairing the specification. In Figure 2, the faulty candidate drives the lever upward while the timer still runs, accepting the early-release trace and rejecting the stay-down trace. When its expectation remains supported, the LLM uses the feedback and program source to revise the responsible predicate, rule, or effect. Here, disabling premature release yields $\mathcal{I}^{(r+1)}$, which agrees with both expectations. Expectations may also be reconsidered if their supporting evidence was misread or insufficient, but not merely to match the candidate.

After an edit, the agent recompiles the specification and replays retained traces as regression tests. Additional traces can probe the revised behavior; the agent decides when to stop within its session budget. The protocol, including replay semantics and revision rules, is in Appendix B.2.

3.4 AuthorBench and OperateBench

Our two benchmarks have distinct goals: AuthorBench evaluates acquired behavior fidelity, while OperateBench evaluates embodied agents' ability to use objects governed by such behavior.

AuthorBench. This benchmark tests whether IOL and trace-based correction improve authoring of operational behavior over direct procedural code with agentic testing and revision. We manually author reference specifications for 50 objects with coupled motions, timing, sequencing, and interlocks. A frozen set of numerical traces is replayed through the references and generated models.

Comparing validity labels and resolved joint trajectories on identical inputs measures behavioral fidelity across representations. Trajectory agreement distinguishes models that reject the same requests but produce different joint responses. Reference programs, traces, and outputs are withheld during authoring and repair. Asset sources and evaluation details appear in Appendices H and E.1.

OperateBench. Successful object use requires more than reaching a joint configuration: an agent must also respect prerequisites, coordinate controls, and account for time. OperateBench tests these abilities in 25 simulated manipulation tasks. Each task supplies an initial state, goal, and manipulation-skill interface, with fixed IOL specifications governing the objects' responses. These specifications are part of the benchmark environment; the evaluated agent authors interaction policies, not the object specifications. For example, bread in the toaster scene heats only while it is inside the slot and the toaster is running, requiring coordination of placement and cycle control. We build on CaP-X Fu et al. (2026) for policy authoring: code lets agents combine skills with conditional decisions and feedback. We evaluate policy authoring with privileged simulator object state and, separately, with only RGB-D perception, studying object use with and without direct state access.

4 EXPERIMENTS

On AuthorBench, we ask how well LLMs author operational behavior with IOL compared with direct code, and what trace-based correction adds to agentic testing and revision. On OperateBench, we ask how well embodied agents operate objects with these behaviors.

4.1 EXPERIMENTAL SETUP

AuthorBench protocol. LLM agents receive URDF-derived asset information, a rendered view, shared public guidance, and seven generic mechanism examples expressed in the target representation. Our main comparison uses GPT-5.6 Sol at ultra reasoning effort, with a 45-minute deadline per authoring session. Additional models and input ablations appear in Appendix F.

The *SAPIEN Native* condition expresses behavior through simulator-specific Python APIs. *Update Function* instead uses a stateful, simulator-independent Python function to mediate joint updates. We compare both with IOL under *agentic self-testing*: the author can write and run tests, inspect their outcomes, and revise its code. For IOL, we additionally evaluate one-pass authoring without execution feedback and *trace-based correction* using the tools described in Method. Interfaces and feedback are detailed in Appendices A.3 and C.

At evaluation, we replay the same frozen joint-motion proposal traces through each candidate and reference using a common numerical interface. The suite combines traces from reference-controller paths with asset-specific tests. For *SAPIEN Native*, a source adapter executes the authored callbacks through a numerical joint-state API; this evaluation does not run full physics simulation.

Our primary metric, *trajectory match*, is the percentage of traces matching reference joint positions at every step, within tolerance. *Validity F1* measures agreement with reference Valid/Invalid labels: a trace is Valid if all requested joint positions are preserved. Correctly blocking a prohibited motion therefore makes a trace Invalid, not an error in the model. Scores are averaged equally over all 50 objects, with incomplete evaluations contributing zero. Table 2 also reports *Completed evaluations*: the number of assets whose candidates cover all required reference joints and replay the full suite without execution failure, out of all 50 assets. Completion does not imply behavioral correctness. Precision/recall definitions, numerical tolerances, and replay details are in Appendix E.1.

OperateBench protocol. We use GPT-6 Astra at ultra reasoning effort; the perception condition uses calibrated RGB-D observations and SAM 3.1 segmentation (Meta AI, 2026). Policies control a floating gripper through bounded physical motions. Each scored run allows up to 25 simulator attempts: the agent tests a policy, receives feedback, and can revise it before the next reset episode. *Tasks solved within n trials* counts tasks with a successful policy found within n attempts and confirmed in a separate final replay. Table 3 summarizes checkpoints at $n \in \{5, 10, 15, 20, 25\}$ for Astra/ultra with and without privileged object state. Additional privileged-state results for Astra, Sol, and Terra at low and ultra reasoning effort and Luna at low and max effort, along with full protocols, are in Appendices D and E.2.

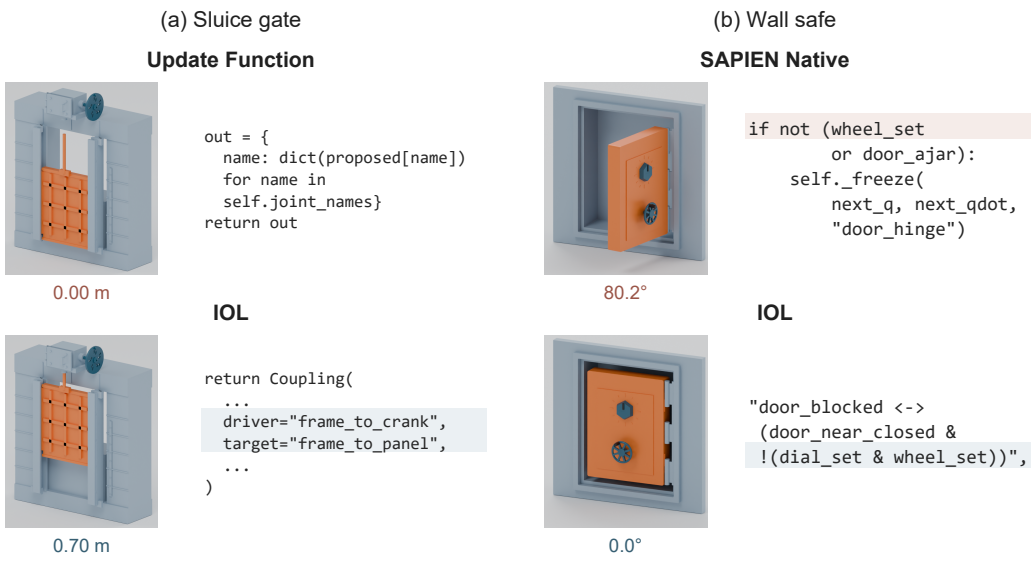


Figure 3: **Operational behavior under identical action sequences.** (a) *Update Function* omits crank–panel coupling: three crank turns leave the panel stationary, whereas IOL raises it by 0.70 m. (b) *SAPIEN Native* omits the dial prerequisite, incorrectly permitting door opening after the dial is reset; *IOL* keeps the door closed. All programs are authored with Sol-ultra.

Table 1: **IOL improves trajectory match even with agentic self-testing.** Sol/ultra on 50 objects. Scores are percentages, averaged over all 50 objects, with incomplete evaluations contributing zero. Valid is the positive class for precision, recall, and F1.

Authoring target	Correction	Precision ↑	Recall ↑	Validity F1 ↑	Trajectory match ↑
SAPIEN Native	Self-testing	89.3	85.7	81.0	75.3
Update Function	Self-testing	84.1	91.6	82.9	74.5
IOL (ours)	Self-testing	91.2	87.2	86.2	80.9
IOL (ours)	Trace-based	**91.3**	**91.9**	**89.0**	**83.3**

4.2 Authoring Operational Behavior

With agentic self-testing for all three representations, IOL reaches 80.9% trajectory match, compared with 74.5% for Update Function and 75.3% for SAPIEN Native (Table 1). Validity F1 also improves. Thus, IOL already improves accuracy when the LLM writes its own tests, without supplied trace-based correction tools. The 6.4-point gain over Update Function, also simulator-independent, shows that portability alone does not explain the authoring accuracy gap. Figure 3 shows qualitative examples of IOL's improved performance, and Appendix F reports comparisons across models, including per-asset analyses by reference controller size.

4.2.1 The Role of Trace-Based Correction

Trace-based correction further improves Sol/ultra authoring (Table 2). Holding IOL fixed, agentic self-testing raises trajectory match from 75.2% without correction to 80.9%; trace-based correction adds another 2.4 points, reaching 83.3%. Validity F1 and evaluation completion also improve. These independent runs compare end-to-end authoring workflows: for Sol/ultra, supplied trace-based tests and feedback improve on agentic self-testing. Gains depend on the model and repair setting; Appendix F reports broader comparisons, including repair from fixed initial programs.

Table 2: **Trace-based correction further improves Sol/ultra authoring.** Sol/ultra with the same public inputs and session deadline. Each row uses independent authoring runs. Completed evaluations: completed/total assets. Metrics and aggregation follow Table 1.

Authoring workflow	Completed evaluations	Precision ↑	Recall ↑	Validity F1 ↑	Trajectory match ↑
One-pass authoring	47/50	86.2	83.8	81.8	75.2
Agentic self-testing	49/50	91.2	87.2	86.2	80.9
Trace-based correction	50/50	**91.3**	**91.9**	**89.0**	**83.3**

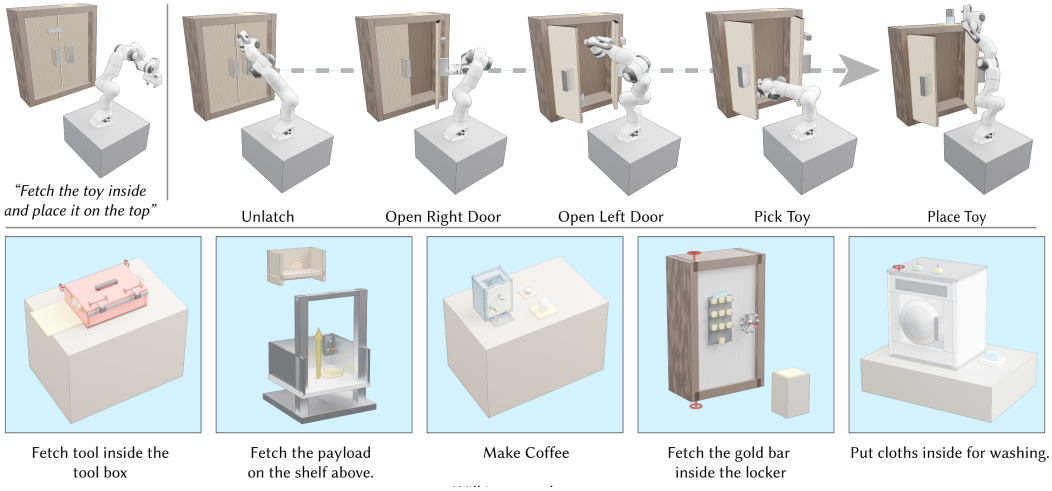


Figure 4: **OperateBench task settings.** Operational rules make object use depend on control sequences and interactions among parts. The top row illustrates a sequence of interactions; the bottom row shows additional task scenes. This illustration depicts the task settings, not the floating-gripper controller used for the reported RGB-D evaluation.

4.3 Operating Objects

OperateBench tests whether an agent can satisfy an object's operational prerequisites while manipulating its parts and controls. Figure 4 illustrates the task settings. Table 3 reports tasks Astra/ultra solves within each trial budget, with and without privileged object state. These checkpoints track the same adaptive authoring runs, rather than independent attempts.

With RGB-D observations and no privileged state, Astra/ultra solves one task within five trials and five within ten. It solves 14 of 25 tasks within 25 trials, so most solved tasks require more than ten trials. Even with ultra reasoning effort and repeated policy revision, 11 of 25 tasks exhaust the allowance without success.

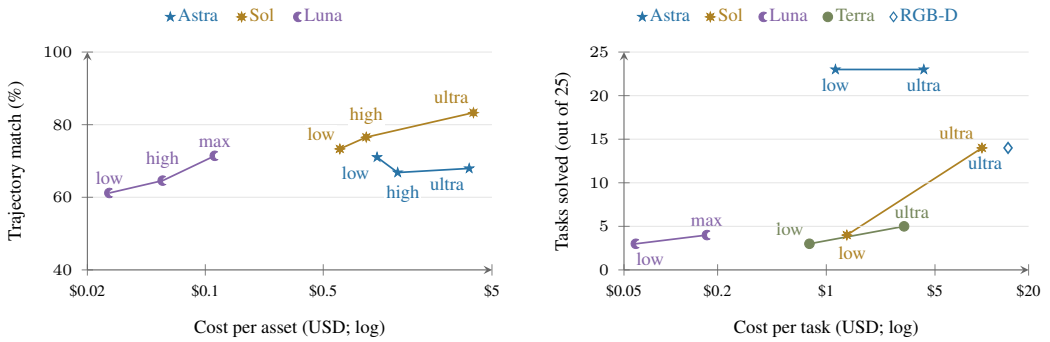
With privileged object state, Astra/ultra solves 23 of 25 tasks within 25 trials. This gives a performance reference with richer inputs, although workflow differences prevent definitively attributing the gap to perception alone. Additional model/reasoning results, protocol details, and infrastructure-replacement accounting are in Appendix E.2. All settings use one simulator seed.

4.4 Cost and Time

Figure 5(a) compares AuthorBench trajectory match and estimated authoring cost across three models and reasoning-effort settings. Trajectory match increases with spending for Sol and Luna, but not Astra. For Sol/ultra, IOL with trace-based correction averages $3.98 in estimated API-equivalent cost and 10.42 minutes per object, compared with $3.43 and 7.56 minutes for IOL with self-testing. The 2.4-point gain in trajectory match accompanies a 16% increase in estimated cost. Update Func-

Table 3: **OperateBench with and without privileged information.** Astra/ultra; number of tasks solved out of 25 within n adaptive simulator trials, with success confirmed by final replay. Both workflows omit initial explicit IOL specifications.

	Tasks solved (out of 25) within n trials ↑				
Information access	$n = 5$	$n = 10$	$n = 15$	$n = 20$	$n = 25$
Without privileged information	1	5	8	12	14
With privileged information	12	18	22	23	23



(a) AuthorBench

(b) OperateBench

Figure 5: **The gains from additional reasoning depend on the model.** (a) AuthorBench with IOL, trace-based correction, and image input. (b) OperateBench with privileged state and no initial IOL specification; the open diamond is the separate Astra/ultra RGB-D workflow. Labels denote reasoning effort; lines connect settings within each model and workflow. Costs average selected authoring runs, including failures and helpers, over all 50 assets or 25 tasks. Accounting details are in Appendix G.

tion and SAPIEN Native cost $2.56 and $2.99 per object, with mean times of 7.20 and 9.01 minutes. These observed workflow costs include recorded retries and failures; comparisons are not at equal cost. Accounting details are in Appendix G.1.

For OperateBench, Figure 5(b) reports tasks solved versus mean estimated API cost per task, averaged over all 25 tasks including failures. The RGB-D workflow is shown separately from privileged-state runs. Scored perception runs use 471 simulator attempts across 25 tasks, with a six-hour authoring deadline per run. Attempts measure interaction effort, while the deadline specifies the time budget; measured elapsed time remains unavailable for the model/reasoning sweep.

5 Conclusion

We introduced IOL, a portable representation of operational behavior that supports LLM authoring through trace-based correction. On AuthorBench, IOL with trace-based correction improves behavioral fidelity over procedural code baselines that also support agentic testing and revision. On OperateBench, the RGB-D-based agent solves 14 of 25 tasks despite repeated policy testing and revision, highlighting the remaining challenge of reliably operating objects from perception.

Limitations. Asset geometry and annotations can leave intended behavior ambiguous, requiring assumptions by the authoring LLM. The LLM can make the same mistake in its specification and its trace expectations, while sampled traces may miss other errors. Agreement after repair therefore does not guarantee correct behavior. Controller synthesis can also become costly with more interacting operating conditions; Appendix I discusses these limitations and possible extensions. Finally, OperateBench evaluates simulated interaction with one simulator seed; its perception setting uses rendered RGB-D observations and a floating gripper. These results neither establish real-robot performance nor isolate failures to perception, physical control, or operational reasoning.

AI USE STATEMENT

We used generative AI tools, including ChatGPT and Codex, for feedback on research methodology and experimental design, assistance with software implementation and debugging, and manuscript drafting and revision. The authors reviewed AI-generated suggestions and content and revised them where appropriate. They made the final research decisions and take responsibility for the manuscript and accompanying artifacts, including AI-assisted content.

LLMs also serve as components of the evaluated systems for operational-program authoring and manipulation-policy generation; these uses are documented in the method, experiments, and supplementary material.

REPRODUCIBILITY STATEMENT

The IOL representation, execution semantics, and baseline interfaces are described in Appendix A. The authoring and trace-based correction workflows are detailed in Appendix B, with AuthorBench inference settings reported in Appendix C and selected prompt instructions in Appendix K. Dataset sources and asset coverage are documented in Appendices H and J. OperateBench inference settings are reported in Appendix D, and the evaluation protocols for both benchmarks are described in Appendix E.

The supplementary AuthorBench artifacts include authored operational programs, archived prompts and model inputs, frozen reference programs, numerical evaluation traces with reference outputs, saved evaluation reports, and scripts for recomputing aggregate metrics from the saved evaluation counts. Supplementary videos provide qualitative evidence for the OperateBench experiments.

The current supplementary bundle does not include the inference pipeline, execution runtime, or evaluation backend. Upon acceptance, we will publicly release the complete codebase for this work, including these components.

REFERENCES

BEHAVIOR. Omnigibson transition rules documentation. https://behavior.stanford.edu/omnigibson/transition_rules.html, 2026. Accessed: 2026-09-25.

Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012. doi: 10.1016/j.jcss.2011.08.007.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/2460396f2d0d421885997dd1612ac56b-Abstract-Conference.html.

Matt Deitke, Eli VanderBilt, Alvaro Herrasti, Luca Weihs, Kiana Ehsani, Jordi Salvador, Winson Han, Eric Kolve, Aniruddha Kembhavi, and Roozbeh Mottaghi. Procthor: Large-scale embodied ai using procedural generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/27c546ab1e4f1d7d638e6a8dfbad9a07-Abstract-Conference.html.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Dantong Niu, Fei-Fei Li, Guanya Shi, Jiajun Wu, Shankar Sastry, Yuke Zhu, Ken Goldberg, and Linxi Fan. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. In *Proceedings of the International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2603.22435.

Chuang Gan, Jeremy Schwartz, Seth Alter, Damian Mrowca, Martin Schrimpf, James Traer, Julian De Freitas, Jonas Kubilius, Abhishek Bhandwaldar, Nick Haber, Megumi Sano, Kuno Kim, Elias Wang, Michael Lingelbach, Aidan Curtis, Kevin Feigelis, Daniel Bear, Dan Gutfreund, David Cox, Antonio Torralba, James J DiCarlo, Josh Tenenbaum, Josh McDermott, and Dan Yamins. Threedworld: A platform for interactive multi-modal physical simulation. In J. Vanschoren and S. Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/file/735b90b4568125ed6c3f678819b6e058-Paper-round1.pdf.

Ran Gong, Jiangyong Huang, Yizhou Zhao, Haoran Geng, Xiaofeng Gao, Qingyang Wu, Wensi Ai, Ziheng Zhou, Demetri Terzopoulos, Song-Chun Zhu, Baoxiong Jia, and Siyuan Huang. ARNOLD: A benchmark for language-grounded task learning with continuous states in realistic 3D scenes. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023. URL https://arnold-benchmark.github.io/.

Jiayuan Gu, Fanbo Xiang, Xuanlin Li, Zhan Ling, Xiqiang Liu, Tongzhou Mu, Yihe Tang, Stone Tao, Xinyue Wei, Yunchao Yao, Xiaodi Yuan, Pengwei Xie, Zhiao Huang, Rui Chen, and Hao Su. Maniskill2: A unified benchmark for generalizable manipulation skills. *arXiv preprint arXiv:2302.04659*, 2023. doi: 10.48550/arXiv.2302.04659. URL https://arxiv.org/abs/2302.04659.

Stephen James, Zicong Ma, David Rovick Arrojo, and Andrew J. Davison. Rlbench: The robot learning benchmark and learning environment. *IEEE Robotics and Automation Letters*, 5(2):3019–3026, 2020. doi: 10.1109/LRA.2020.2974707.

Abhishek Joshi, Beining Han, Jack Nugent, Max Gonzalez Saez-Diez, Yiming Zuo, Jonathan Liu, Hongyu Wen, Stamatis Alexandropoulos, Karhan Kayan, Anna Calveri, Tao Sun, Gaowen Liu, Yi Shao, Alexander Raistrick, and Jia Deng. Procedural generation of articulated simulation-ready assets. *arXiv preprint arXiv:2505.10755*, 2025. URL https://arxiv.org/abs/2505.10755.

Eric Kolve, Roozbeh Mottaghi, Winson Han, Eli VanderBilt, Luca Weihs, Alvaro Herrasti, Matt Deitke, Kiana Ehsani, Daniel Gordon, Yuke Zhu, Aniruddha Kembhavi, Abhinav Gupta, and Ali Farhadi. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*, 2017. doi: 10.48550/arXiv.1712.05474. URL https://arxiv.org/abs/1712.05474.

Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Temporal-logic-based reactive mission and motion planning. *IEEE Transactions on Robotics*, 25(6):1370–1381, 2009. doi: 10.1109/TRO.2009.2030225.

Long Le, Jason Xie, William Liang, Hung-Ju Wang, Yue Yang, Yecheng Jason Ma, Kyle Vedder, Arjun Krishna, Dinesh Jayaraman, and Eric Eaton. Articulate-anything: Automatic modeling of articulated objects via a vision-language foundation model. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/2c8047bf3ed8ef6905351608d641f02f-Abstract-Conference.html.

Chengshu Li, Fei Xia, Roberto Martín-Martín, Michael Lingelbach, Sanjana Srivastava, Bokui Shen, Kent Elliott Vainio, Cem Gokmen, Gokul Dharan, Tanish Jain, Andrey Kurenkov, C. Karen Liu, Hyowon Gweon, Jiajun Wu, Li Fei-Fei, and Silvio Savarese. igibson 2.0: Object-centric simulation for robot learning of everyday household tasks. *Proceedings of Machine Learning Research*, 164:455–465, 2022. URL https://proceedings.mlr.press/v164/li22b.html.

Chengshu Li, Ruohan Zhang, Josiah Wong, Cem Gokmen, Sanjana Srivastava, Roberto Martín-Martín, Chen Wang, Gabrael Levine, Michael Lingelbach, Jiankai Sun, Mona Anvari, Minjune Hwang, Manasi Sharma, Arman Aydin, Dhruva Bansal, Samuel Hunter, Kyu-Young Kim, Alan Lou, Caleb R. Matthews, Ivan Villa-Renteria, Jerry Huayang Tang, Claire Tang, Fei Xia, Silvio Savarese, Hyowon Gweon, C. Karen Liu, Jiajun Wu, and Li Fei-Fei. Behavior-1k: A benchmark for embodied ai with 1,000 everyday activities and realistic simulation. In *Proceedings of The 6th Conference on Robot Learning*, volume 205 of *Proceedings of Machine Learning Research*, pp. 80–93, 2023. URL https://proceedings.mlr.press/v205/li23a.html.

Meta AI. SAM 3.1: Faster and more accessible real-time video detection and tracking with multiplexing and global reasoning, 2026. URL https://ai.meta.com/blog/segment-anything-model-3/. Accessed September 26, 2026.

Soroush Nasiriany, Abhiram Maddukuri, Lance Zhang, Adeet Parikh, Aaron Lo, Abhishek Joshi, Ajay Mandlekar, and Yuke Zhu. RoboCasa: Large-scale simulation of everyday tasks for generalist robots. In *Robotics: Science and Systems (RSS)*, 2024. URL https://arxiv.org/abs/2406.02523.

Soroush Nasiriany, Sepehr Nasiriany, Abhiram Maddukuri, and Yuke Zhu. RoboCasa365: A large-scale simulation framework for training and benchmarking generalist robots. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2603.04356.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 2015. URL https://www.amazon.science/publications/how-amazon-web-services-uses-formal-methods.

NVIDIA. Isaac Sim. Robotics simulation software, n.d. URL https://docs.isaacsim.omniverse.nvidia.com/latest/index.html. Accessed September 24, 2026.

Aishwarya Padmakumar, Jesse Thomason, Ayush Shrivastava, Patrick Lange, Anjali Narayan-Chen, Spandana Gella, Robinson Piramuthu, Gokhan Tur, and Dilek Hakkani-Tur. TEACh: Task-driven embodied agents that chat. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2022. URL https://arxiv.org/abs/2110.00534.

Weikun Peng, Denys Iliash, and Manolis Savva. Egofun3d: Modeling interactive objects from egocentric videos using function templates. *arXiv preprint arXiv:2604.11038*, 2026. doi: 10.48550/arXiv.2604.11038. URL https://arxiv.org/abs/2604.11038.

Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 46–57, 1977. doi: 10.1109/SFCS.1977.32.

Xavier Puig, Kevin Ra, Marko Boben, Jiaman Li, Tingwu Wang, Sanja Fidler, and Antonio Torralba. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*

(CVPR), pp. 8494–8502, 2018. doi: 10.1109/CVPR.2018.00886. URL https://openaccess.thecvf.com/content_cvpr_2018/html/Puig_VirtualHome_Simulating_Household_CVPR_2018_paper.html.

Xiaowen Qiu, Jincheng Yang, Yian Wang, Zhehuan Chen, Yufei Wang, Tsun-Hsuan Wang, Zhou Xian, and Chuang Gan. Articulate anymesh: Open-vocabulary 3d articulated objects modeling. In *Proceedings of The 9th Conference on Robot Learning*, volume 305 of *Proceedings of Machine Learning Research*, pp. 4585–4606, 2025. URL https://proceedings.mlr.press/v305/qiu25b.html.

ROS 2 Documentation. urdfdom: Rolling 6.0.0 documentation. https://docs.ros.org/en/rolling/p/urdfdom/, 2026. Accessed: 2026-04-27.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, 2023. URL https://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. ALFRED: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. URL https://arxiv.org/abs/1912.01734.

Sanjana Srivastava, Chengshu Li, Michael Lingelbach, Roberto Martín-Martín, Fei Xia, Kent Elliott Vainio, Zheng Lian, Cem Gokmen, Shyamal Buch, C. Karen Liu, Silvio Savarese, Hyowon Gweon, Jiajun Wu, and Li Fei-Fei. Behavior: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *Proceedings of The 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pp. 477–490, 2022. URL https://proceedings.mlr.press/v164/srivastava22a.html.

Andrew Szot, Alexander Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John Turner, Noah Maestre, Mustafa Mukadam, Devendra Singh Chaplot, Oleksandr Maksymets, Aaron Gokaslan, Vladimir Vondrus, Sameer Dharur, Franziska Meier, Wojciech Galuba, Angel Chang, Zsolt Kira, Vladlen Koltun, Jitendra Malik, Manolis Savva, and Dhruv Batra. Habitat 2.0: Training home assistants to rearrange their habitat. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/hash/021bbc7ee20b71134d53e20206bd6feb-Abstract.html.

Fanbo Xiang, Yuzhe Qin, Kaichun Mo, Yikuan Xia, Hao Zhu, Fangchen Liu, Minghua Liu, Hanxiao Jiang, Yifu Yuan, He Wang, Li Yi, Angel X. Chang, Leonidas J. Guibas, and Hao Su. Sapien: A simulated part-based interactive environment. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11097–11107, 2020. URL https://openaccess.thecvf.com/content_CVPR_2020/html/Xiang_SAPIEN_A_SimulAted_Part-Based_Interactive_ENvironment_CVPR_2020_paper.html.

Mandi Zhao, Yijia Weng, Dominik Bauer, and Shuran Song. Real2code: Reconstruct articulated objects via code generation. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/028fcbcf85435d39a40c4d61b42c99a4-Abstract-Conference.html.

Matt Zhou, Ruining Li, Xiaoyang Lyu, Zhaomou Song, Zhening Huang, Chuanxia Zheng, Christian Rupprecht, Andrea Vedaldi, and Shangzhe Wu. Articraft: An agentic system for scalable articulated 3d asset generation. *arXiv preprint arXiv:2605.15187*, 2026. URL https://arxiv.org/abs/2605.15187.

APPENDIX

This appendix provides implementation details, evaluation protocols, additional analyses, and benchmark assets and prompts.

Anonymous supplementary materials:
https://anonymous.4open.science/r/beyond_articulation_supplementary/

CONTENTS

A IOL Specification and Execution 15

B LLM Authoring and Trace-Based Correction 16

C AuthorBench Inference Protocol 17

D OperateBench Inference Protocol 18

E Benchmark Evaluation Strategies 19

F Extended Authoring Results and Analyses 22

G Inference Cost 26

H Dataset Collection and Usage 27

I Limitations and Future Directions 27

J Asset Coverage and Manifests 28

K AuthorBench Prompts and Input Construction 30

L OperateBench Prompts 37



Figure 6: **Security-gate asset in two simulators.** The same asset rendered in Isaac Sim (left) and SAPIEN (right), each integrated with the shared IOL runtime through a backend-specific adapter.

A IOL Specification and Execution

A.1 Representation and Execution Semantics

Object-local update. Let x_t contain the object's joint positions, velocities, and accelerations, and let $\widehat{x}_{t+1}$ be a proposed next state. An IOL specification $\mathcal{I}$ defines a runtime

$$(x_{t+1}, h_{t+1}) = M_{\mathcal{I}}(x_t, \widehat{x}_{t+1}, h_t; \Delta t), \tag{3}$$

where x_{t+1} is the resolved state, h_t stores controller and effect-local memory, and Δt is the elapsed time. The same interface accepts simulator-proposed updates during embodied interaction and numerical proposals during trace-based authoring tests.

Representation and execution. The vocabulary $\mathcal{P} = \mathcal{P}^{\text{ext}} \cup \mathcal{P}^{\text{int}}$ contains observed and internal propositions. Groundings Φ evaluate the observed propositions through stateless Boolean predicates on the current state x_t. Temporal rules Ψ use a GR(1)-compatible safety fragment Pnueli (1977); Bloem et al. (2012); internal memory propositions have initial values and next-step rules when history is required. Synthesis produces the finite-state controller $\mathcal{A}_\Psi$, which maps the truth values of observed propositions and the current controller state to operating modes and the next controller state. Conditioned numerical effects $\mathcal{E}$ read these modes and modify the working proposal in declaration order. For example, a freeze effect preserves a joint's previous position, while a steady-velocity effect advances it using its rate and Δt. The resolved object state is observed on the following step. Trace initialization, position-preservation labels, and numerical tolerances are specified in Appendix B.2.

A.2 Backend Interface and Portability

Available simulator integrations. Our framework provides a shared IOL runtime with adapters and runnable asset examples for SAPIEN and Isaac Sim. Both integrations load an articulated asset and its IOL specification, allowing the same operational behavior to be used in either simulator. For example, the supplied timer demonstration uses the same asset and IOL specification in both backends. Users select the simulator integration while retaining the asset's proposition groundings, temporal rules, and conditioned effects. Figure 6 shows the security-gate asset in both simulators.

Reusable operational models. IOL expresses behavior in terms of named asset joints, their coordinates and limits, and elapsed time. These declarations are independent of simulator APIs: prerequisites, persistent modes, joint coupling, and timed behavior are defined once and executed by the shared runtime. At each interaction step, the simulator supplies a proposed object state, IOL resolves the operationally valid update, and the adapter applies it to the simulated asset. The simulator continues to provide robot dynamics, contact, collision, and rendering. This separation lets an operational model travel with its asset across simulation environments without rewriting its behavior as simulator-specific controller code.

Adapting to another simulator. Adding a backend requires an adapter that maps asset joint names to native simulator handles, exposes current and proposed joint states in consistent units and coordinate conventions, and applies the resolved positions and velocities. The integration also supplies the

elapsed time and resets the runtime's operational memory at episode boundaries. These responsibilities are localized to the simulator boundary; the IOL specification and shared reasoning and effect execution remain unchanged. The existing SAPIEN and Isaac Sim integrations provide examples of this integration pattern. Portability concerns the operational model; physical trajectories still depend on the backend's dynamics and constraint handling.

A.3 Baseline Representations

We compare IOL with two alternative representations of an object's operational behavior. All three use the same public asset evidence and are evaluated on the same held-out proposal traces through the common transition interface (Appendix E.1). They differ in how the author expresses and executes the behavior.

Update Function. An Update Function is a stateful, simulator-independent Python program that directly implements the object-local update interface of Eq. (3). The program implements prerequisites, coupling, timers, and persistent modes without a required decomposition into propositions, temporal rules, and conditioned effects. It can preserve, block, or modify the proposal and advance autonomous behavior using Δt. The supplied interface is an `ObjectMediator` class with `reset()` and `resolve(prev, proposed, dt)` methods. Here, h_t is the program's private operational memory, stored on the instance and reset between traces. The `resolve` method returns x_{t+1}, retaining the input joint names and position, velocity, and acceleration fields, $(q, \dot{q}, \ddot{q})$. Appendix K.2.2 provides the authoring prompt excerpts specifying the output format and joint-state interface.

SAPIEN Native. The SAPIEN Native authoring target expresses operational behavior as Python code using SAPIEN APIs. The author supplies a possibly multi-file program whose `entry.py` defines `create(articulation, *, dt)` and returns a `sapien.System`. The program may maintain episode-local memory, read the target articulation's joint positions, velocities, and accelerations, and act through native joint drives or direct position and velocity writes. Static joint metadata and the fixed timestep are available; the scene, global clock, intervention schedule, and evaluator labels are not exposed to the candidate. The authoring prompt and API excerpts in Appendix K.2.3 detail the program lifecycle, state access, and actuation conventions. AuthorBench evaluates the program through the reviewed numerical source adapter described in Appendix E.1.

B LLM Authoring and Trace-Based Correction

The authoring agent uses interaction sequences to test its interpretation of an object: what should happen when a part moves, and does the generated program respond that way? This section describes initial IOL authoring and the trace-based correction procedure. The experimental inputs, feedback conditions, and execution budgets are specified in Appendix C; selected prompt excerpts appear in Appendix K.

B.1 Initial Specification

Interpreting the mechanism. The agent uses knowledge of the named object together with the supplied evidence to identify prerequisites, dependencies, persistent states, timed motion, and coupling. It encodes these decisions as propositions, temporal rules, and effects. A movable part alone does not establish that a motor or lock exists: explicit asset evidence takes precedence over general assumptions, and unsupported details remain uncertain. The same interpretation guides what the agent expects during replay. Shared modeling conventions and executable examples support this translation (Section C); the agent selects the mechanisms, participating joints, and activation conditions for the current asset.

B.2 Trace-Based Correction

Selecting interaction sequences. A trace starts with the object at its public rest state and zero velocity and acceleration. It then specifies a sequence of requested positions, velocities, and accelerations for every joint. The sampler combines paths through the candidate controller with probes

of individual joints and joint combinations. Candidate replay also helps find short sequences that produce motion. The returned sequences contain no replay labels or reference behavior: the agent must judge what the object should do.

Predicting the response. For each selected sequence, the agent writes an expectation and explains its basis in the asset evidence or mechanism knowledge. It considers which controls have been operated, which prerequisites are met, and how earlier interactions affect later motion. Each row refers to an unchanged sampled sequence. Its explanation may focus on a particular step, but its scored expectation concerns the entire sequence: should every requested joint position be preserved, or should the object change at least one? Expectations are recorded before the corresponding table replay. Earlier feedback may be available, but candidate behavior does not determine intended behavior. Ambiguous sequences may be omitted; if none can be judged, the agent records why and the table provides no behavioral coverage.

Replaying the sequence. The harness resets the program's memory and starts from the trace's initial state. At each step, the controller reads the actual state reached so far and the effects resolve the next requested update. This result becomes the starting state for the following step, while the remaining requests stay fixed. A control moved during one step is therefore observed by the controller at the next step. Replay records the requested and resulting positions, including the first step and joint where they differ.

Trace comparison. We compare the agent's expectation with the replay label, which is computed from joint positions. For a trace τ, let $\widehat{q}_{t,j}$ be the requested position of joint j after step t and $q_{t,j}$ the position produced by the program. The observed label is

$$v(\tau) = \begin{cases} \text{Valid}, & \max_{t,j} |q_{t,j} - \widehat{q}_{t,j}| \le \epsilon, \\ \text{Invalid}, & \text{otherwise.} \end{cases} \tag{4}$$

Here t ranges over all executed steps, excluding the initial state, and j over every runtime joint, including joints requested to hold still. Replay uses 0.1-second steps and $\epsilon = 10^{-6}$ in each joint's native units. Velocities and accelerations affect execution but are not scored. One positional deviation makes the trace Invalid even if later steps match.

The comparison agrees when this label equals the agent's expectation. Invalid can be the expected result of correctly blocking a locked door, coupling parts, or moving a part automatically when the request leaves it still. It is not a judgment that the program is wrong. Execution failures are reported separately and receive no Valid/Invalid label. For successful replays, the trace feedback includes per-step results and position errors to help the agent interpret what happened beyond the binary comparison.

Deciding what to revise. The agent examines discrepancies using its expectation, numerical replay, and program source. If a door remains blocked after its latch is released, it can inspect the release predicate, the timing of the unlocked mode, and the blocking effect. When the expectation is still supported, it revises the relevant grounding, rule, or effect. An expectation can also change if the evidence was misread, but not merely to match execution. The accompanying analysis records the reasoning, uncertainty, and decision. Any program or expectation change requires a new replay. Retained sequences can be used to check later revisions, and the final replay must use the submitted program and table.

Agreement alone cannot establish that an intervention changed the right joint for the right reason. Shared misconceptions can also persist in both program and expectations. We assess behavioral quality separately using held-out evaluation (Section E.1).

C AuthorBench Inference Protocol

Inputs and guidance. All conditions receive the URDF and `asset_summary.json`, including joint/link structure, types, limits, kinematic metadata, and available `semantic_label` annotations. Image-enabled runs also receive a rendered contact sheet. A shared behavior contract provides modeling conventions and numerical defaults; seven generic mechanism examples use each

representation's API. Asset evidence and shared conventions are fixed within each comparison, except for rendered views in the visual-input ablation. Reference programs, held-out traces, private labels, and held-out evaluation results are withheld.

Feedback conditions. One-pass authoring (No-EC) prohibits candidate execution, tests, reviewer checks, and correction loops. Agentic self-testing (Naive-EC) permits optional self-authored checks and repair, without supplied measurement, sampling, or trace-table tools. Trace-based correction (TR-EC) additionally provides candidate-derived diagnostics, trace sampling, and replay against whole-trace expectations recorded before replay (Section B.2). Its optional diagnostic tool can also sample and replay without an expectation table. The comparison therefore evaluates the complete supplied workflow. The main trace-based correction runs use no separate critic.

Initialization. Representation, correction, and visual-input comparisons use independent fresh authoring sessions. The paired repair experiment instead compares free-form tests with trace-based repair from frozen one-pass programs, using Sol/ultra for both repair conditions (Section F.5). For each asset and source-model cohort, repair arms receive the same initial source in fresh sessions, without original conversations, check results, or other arms' feedback. Initial programs are not filtered by validation quality.

Budgets and stopping. Experiments use Codex CLI `0.153.4`, with Sol/ultra for the main comparisons. Additional model and reasoning-effort settings are specified alongside their results. Each authoring session has a 45-minute deadline. Managed IOL operations have a 640-second deadline; freely authored commands and native final checks retain their own limits. In trace-based correction, the agent requests 64 traces per call through the supplied sampling tool. Sampling uses $\Delta t = 0.1$ s, with no default horizon cap or cumulative sampling cap, subject to operation and session deadlines. Agents stop by submission, abstention, or deadline. Optional helpers follow the same input and feedback restrictions; their usage and model settings have not been separately audited. These deadlines do not impose equal token or monetary budgets.

Finalization. Final code, feedback, and execution records are preserved. Trace-based correction replays the final candidate and expectation table after edits; checks for one-pass authoring occur after authoring and provide no correction feedback.

D OperateBench Inference Protocol

Adaptive policy authoring. Each condition uses the same 25 tasks and simulator seed 1. A scored authoring run permits up to 25 serial simulator admissions, shared by the author and any helper agents. An attempt executes a candidate policy in a freshly reset episode, not an independent model sample. Admitted episodes are charged even if they subsequently crash. The agent receives feedback and can revise the policy after ordinary failures. Generation stops at the first verified success or at budget exhaustion; in the latter case, the last completed tested policy is selected. A separate replay evaluates the selected frozen policy outside the 25-attempt allowance. Generation has a six-hour deadline and each evaluation a 30-minute timeout.

State-based workflow. We evaluate GPT-6 Astra, GPT-5.6 Sol, and GPT-5.6 Terra at low and ultra reasoning effort, and Luna at low and max effort. The author receives goals, geometry, physical state, and approved mechanical information, but not the initial explicit IOL specification. The trusted simulator retains the original IOL mechanics, and feedback may contain IOL-derived labels. Thus, withholding the explicit specification does not remove all mechanical information. Agents produce policies in a restricted interaction language using file and shell tools, with optional helper agents sharing the simulator allowance. The conditions do not impose equal token, monetary, or model-call budgets.

Perception workflow. GPT-6 Astra at ultra reasoning effort receives the natural-language goal, camera observations, and bounded feedback, without IOL specifications or privileged object state. A fixed, calibrated external camera provides synchronized RGB and ideal rendered metric depth. SAM 3.1 segmentation and visible depth/mask pixels support geometry queries. The author can

Table 4: Frozen numerical evaluation suite. Transition counts exclude the initial state. Valid/Invalid counts are determined by reference replay with $\epsilon = 10^{-6}$.

Statistic	Value	Statistic	Value
Assets	50	Total traces	123,836
Total transitions	730,452	Trace cap per asset	50,000
Reference-valid traces	26,433	Reference-invalid traces	97,403

inspect rendered images; executed policies use opaque observation, detection, and geometry tokens. Hidden task definitions, URDF assets, named simulator joints, target coordinates, and ground-truth state queries are not exposed. Policies control a floating gripper through bounded physical motion; collision-aware path planning, automatic pre-grasp routing, and contact-carry assistance are disabled. This is simulated RGB-D perception-and-control evaluation, not RGB-only or real-robot performance.

Both workflows use one scored run per task, subject to the infrastructure replacements in Appendix E.2. Optional helpers share the attempt allowance. Selected-run cost estimates are reported in Appendix G.2; helper-use counts and complete per-task token and runtime records remain unavailable.

E Benchmark Evaluation Strategies

This section collects the evaluation protocols and metrics for AuthorBench and OperateBench. Asset coverage and dataset overviews are provided in Appendix J, with AuthorBench assets in Section J.1 and OperateBench assets in Section J.2.

E.1 AuthorBench Evaluation

AuthorBench measures how faithfully an authored operational model reproduces a human-authored reference's joint-level responses to held-out numerical interaction traces. A common numerical interface makes these responses comparable across authoring representations under the same inputs and observation domain.

Reference specifications and trace construction. Human annotators encode each asset's intended operational behavior in an executable reference IOL specification. The evaluation suite combines numerical joint-motion proposals derived from reference-controller paths with asset-specific interaction tests. Replaying each trace through the complete reference runtime yields its resolved joint positions and Valid/Invalid q-conformance label. The inputs and reference outputs are frozen, shared across methods, and withheld from candidate authoring and refinement. Table 4 summarizes the suite.

Common numerical interface. AuthorBench evaluates IOL, Update Function, and SAPIEN Native under a common numerical replay and scoring protocol. For each asset, all candidates receive the same initial state and frozen proposal traces at the same timestep. Their responses are evaluated against the same reference over the same joint domain, using identical tolerances, metrics, and aggregation rules. Representation-specific adapters expose a common joint-state transition interface with consistent joint names and coordinate units. IOL and Update Function implement the object-local mapping directly; SAPIEN Native uses a reviewed source adapter to execute the authored program through a numerical joint-state API.

SAPIEN source replay. The SAPIEN adapter invokes the authored callback once per trace step. It preserves explicit position outputs and integrates selected velocity-controlled updates through source annotations fixed before scoring. Registered ancillary drive calls are recorded without simulating their dynamics. Scores therefore measure operational behavior under this numerical interpretation on the shared joint domain, without full physics simulation.

Replay protocol. The candidate and reference reset their operational memory at the start of each trace and receive the same initial state and recorded proposals. Each maintains its own resolved states and memory during execution. The fixed time step $\Delta t = 0.1$ s is the simulated time between successive trace frames; it is passed to each state update to advance timers and time-dependent motion effects. We compare positions at every executed step and every joint in the reference observation domain, including holds and excluding the initial frame. Velocities and accelerations remain part of execution and numerical validation. Both metrics below use an absolute tolerance of $\epsilon = 10^{-6}$ in each joint's native coordinate units, without excursion normalization, to determine numerical agreement. Scoring requires no alignment of propositions, controller states, or program structure.

We evaluate one frozen candidate per asset and condition after authoring (Section C), with retries, replacements, and superseded attempts recorded. Each candidate–asset job has a 900-second timeout. Evaluation results are not returned to the authoring agent.

Metrics and aggregation. For a trace τ, let $\widehat{q}_{t,j}$ denote the requested position of joint j at step t, and $q^M_{t,j}$ the position resolved by model M, where $M = C$ denotes the candidate and $M = G$ the reference. Trajectory agreement compares candidate and reference outputs, while validity compares each model's outputs with the requests:

$$\begin{aligned} \text{Trajectory agreement}: &\quad |q^C_{t,j} - q^G_{t,j}| \le \epsilon \quad \text{for all } t, j, \\ \text{Valid under } M: &\quad |q^M_{t,j} - \widehat{q}_{t,j}| \le \epsilon \quad \text{for all } t, j. \end{aligned} \tag{5}$$

Trajectory match. For an asset's N test traces, let N_{match} count traces with trajectory agreement as defined above. We report the whole-trace match rate:

$$\text{TrajectoryMatch} = \frac{N_{\text{match}}}{N}. \tag{6}$$

The main paper reports this rate as a percentage. A single deviation beyond tolerance prevents a match, regardless of subsequent recovery. This metric distinguishes different resolved position histories even when both models label a trace invalid.

q-conformance label agreement (Valid/Invalid). Correct blocking, coupling, or autonomous motion can make a trace Invalid: this label describes preservation of the requests, not correctness of the operational model. We treat this comparison as binary classification, using the reference label as ground truth and *Valid* as the positive class. Each completed trace pair contributes once to the confusion counts: TP counts traces that both models label valid; TN counts traces that both label invalid; FP counts reference-invalid traces that the candidate labels valid; and FN counts reference-valid traces that the candidate labels invalid. For each asset, we report Precision, Recall, F1, and Accuracy:

$$\begin{aligned} \text{Precision} &= \frac{\text{TP}}{\text{TP} + \text{FP}}, & \text{Recall} &= \frac{\text{TP}}{\text{TP} + \text{FN}}, \\ \text{F1} &= \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}, & \text{Accuracy} &= \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{FN} + \text{TN}}. \end{aligned} \tag{7}$$

Precision measures how often a candidate-valid trace is reference-valid; recall measures how often the candidate preserves a reference-valid trace. F1 is their harmonic mean, and accuracy measures overall label agreement.

Aggregation and completion. We compute each metric per asset, then average it equally over the full 50-asset roster so that larger trace suites do not dominate the results. Each asset's F1 is computed before averaging. A zero denominator makes the corresponding asset metric undefined. Execution failures receive no semantic label or confusion-matrix count and make the asset evaluation incomplete. In the reported macro averages, incomplete assets contribute zero to every metric; complete assets contribute zero for any undefined metric. These macro scores therefore reflect both evaluation completion and behavioral agreement. The *Completed evaluations* column reports the number of assets whose candidates cover all required reference joints and replay the entire suite without execution failure, out of all 50 assets. A program can load successfully yet be incomplete because it omits a required joint; completion does not imply behavioral correctness.

Table 5: **OperateBench with privileged state, without initial explicit IOL specifications.** Cells report solved tasks / 25 and success percentage after adaptive policy authoring and final replay. The highest effort is max for Luna and ultra for the other models.

Model	Low reasoning	Ultra / Max reasoning
Astra	23/25 (92%)	23/25 (92%)
Sol	4/25 (16%)	14/25 (56%)
Terra	3/25 (12%)	5/25 (20%)
Luna	3/25 (12%)	4/25 (16%)

E.2 OperateBench Evaluation

Metric and scored runs. Task success is the number of tasks passed by the selected frozen policy in authoritative final replay, divided by 25. The policy is selected through the adaptive procedure in Appendix D; the score is neither first-attempt accuracy nor independent-sample pass@25. The trusted evaluator applies each task's success criterion while the simulator enforces the object mechanics. All settings use simulator seed 1, without repeated-authoring estimates of variability.

State-based results. Table 5 reports the eight model/effort conditions. Astra solves 23/25 tasks at both settings. Increasing reasoning effort raises Sol from 4/25 to 14/25, Terra from 3/25 to 5/25, and Luna from 3/25 to 4/25.

Perception results and comparison scope. The RGB-D workflow solves 14/25 tasks, as reported in Table 3. Successful tasks are 001, 002, 003, 004, 007, 009, 010, 015, 016, 017, 018, 020, 021, and 023; the other 11 tasks exhaust 25 attempts in their scored runs. The earlier state-based Astra/ultra result is contextual, not a matched perception ablation: equivalence of controllers, feedback, scoring, and recovery policies has not been established. The 56% versus 92% difference cannot be attributed to perception alone.

Recovery accounting. For the original six Astra/Sol/Terra conditions, the state-based sweep retains 141 validated outcomes and adds nine designated replacement sessions for excluded infrastructure or validation cases, yielding 72 successes across 150 model-setting/task pairs, not 150 distinct tasks. Validated benchmark failures were not rerun. The replacement policy was defined after inspecting the original results; eligible cells and their replacement mapping were fixed before replacement outcomes, and replacements were included whether they passed or failed. The nine original sessions used 171 admitted attempts (170 completed); replacements added 198.

The Luna runs additionally required four fresh replacements after infrastructure crashes; all four replacements ended in benchmark failure.

The perception evaluation retains 13 original outcomes and adds two replacements for infrastructure failures and ten first-time runs. Its scored runs use 471 admitted and completed generation attempts; including interrupted originals gives 514 admitted and 512 completed attempts, plus 25 final replays. Replacements receive fresh allowances, so total historical work can exceed 25 attempts per task. Original records remain preserved separately.

Table 6: Authoring representations across Sol and Luna on 50 AuthorBench assets. All conditions use agentic self-testing and IMG-on, with ultra reasoning effort for Sol and max for Luna. Completed evaluations: completed/total assets. Scores average all 50 assets, with incomplete evaluations contributing zero. Bold marks the highest displayed value of each metric within a model.

Representation	Completed evaluations	Precision	Recall	F1	Accuracy	Trajectory match
Sol / ultra						
IOL	49/50	**0.912**	0.872	**0.862**	**0.895**	**0.809**
Update Function	50/50	0.841	**0.916**	0.829	0.836	0.745
SAPIEN Native	50/50	0.893	0.857	0.810	0.869	0.753
Luna / max						
IOL	50/50	**0.874**	0.862	**0.814**	**0.863**	**0.754**
Update Function	50/50	0.756	**0.873**	0.730	0.763	0.651
SAPIEN Native	50/50	0.825	0.834	0.792	0.846	0.729

Table 7: Trajectory match (%) across reasoning-effort settings. The last column uses ultra for Sol and Astra, and max for Luna. Scores average all 50 assets, with incomplete evaluations contributing zero. Bold marks the highest score within each model.

Model	Low	High	Ultra / Max
Sol	73.32	76.55	**83.31**
Astra	**71.04**	66.82	67.95
Luna	61.19	64.59	**71.43**

F Extended Authoring Results and Analyses

The following analyses examine authoring representations, reasoning effort, controller size, visual input, and repair from frozen initial programs on the 50 AuthorBench assets. The tables on representation, visual input, and repair report Completed evaluations, Precision, Recall, F1, Accuracy, and Trajectory match, with quality metrics on a 0–1 scale. The reasoning-effort table reports trajectory match as a percentage. The *Completed evaluations* column reports completed/total assets, using the completion criterion in Appendix E.1. Completion does not imply behavioral correctness. Scores average all 50 assets, with incomplete evaluations contributing zero. In the tables, No-EC denotes *one-pass authoring*, Naive-EC denotes *agentic self-testing*, and TR-EC denotes *trace-based correction*.

F.1 Representation Comparison Across Models

We compare IOL, Update Function, and SAPIEN Native using Sol at ultra reasoning effort and Luna at max reasoning effort (Table 6). All conditions use agentic self-testing with rendered asset views (IMG-on). The Sol results repeat the agentic self-testing rows of Table 1; the Luna results extend the comparison to a second model configuration. Baseline interfaces are defined in Appendix A.3. Each representation is evaluated together with its supplied authoring interface and execution support. Reasoning effort is fixed within each model's representation comparison; comparisons across models also reflect their different reasoning-effort settings.

F.2 Reasoning Effort Across Models

Table 7 compares reasoning effort for IOL authoring with trace-based correction and rendered asset views (IMG-on). Trajectory match increases across the recorded settings for Sol and Luna; Astra scores highest at low effort. Low/high and highest-effort runs come from separate authoring batches, so the comparison is descriptive.

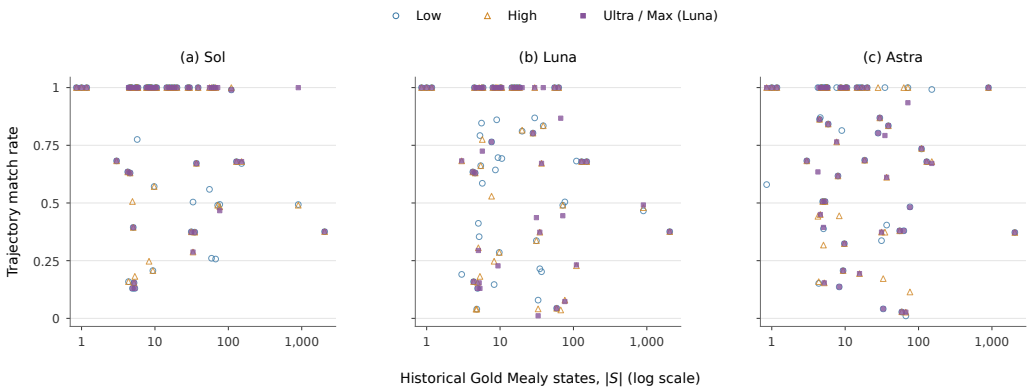


Figure 7: **Authoring performance versus reference controller size.** Each marker shows an asset's trajectory match rate at one reasoning-effort setting, plotted against its historical reference Mealy state count $|S|$ on a logarithmic scale. Panels show Sol, Luna, and Astra; colors and marker shapes distinguish low, high, and highest reasoning effort (ultra for Sol and Astra, max for Luna). The horizontal axis describes historical reference controllers, not the programs authored in these runs.

Table 8: Visual-input ablation for Sol IOL authoring. Completed evaluations: completed/total assets. Scores average all 50 assets, with incomplete evaluations contributing zero. Bold: better IMG-on/off score per metric within each authoring workflow.

Visual input	TR-EC	Naive-EC	No-EC	Completed evaluations	Precision	Recall	F1	Accuracy	Trajectory match
IMG-on	✓			50/50	0.913	**0.919**	**0.890**	**0.906**	**0.833**
IMG-off	✓			50/50	**0.923**	0.891	0.875	0.903	0.806
IMG-on		✓		49/50	**0.912**	**0.872**	**0.862**	**0.895**	**0.809**
IMG-off		✓		48/50	0.895	0.851	0.838	0.874	0.787
IMG-on			✓	47/50	**0.862**	0.838	**0.818**	0.842	**0.752**
IMG-off			✓	48/50	0.860	**0.848**	0.817	**0.853**	0.751

F.3 Authoring Performance versus Controller Size

Figure 7 compares per-asset trajectory match with the number of states in historical reference Mealy controllers. Scores vary substantially among assets with similar state counts, and high trajectory match also occurs for large controllers. Controller size alone therefore does not explain the observed variation in authoring performance. State count is a coarse descriptor of the compiled representation; this comparison does not measure compilation time or memory use.

F.4 Visual-Input Sensitivity

We compare IOL authoring with and without rendered asset views using Sol under each authoring workflow (Table 8). IMG-on includes rendered views, whereas IMG-off omits them. Within each pair, textual context, few-shot examples, reasoning effort, and authoring budget are held fixed. Both conditions use the same held-out trace suites.

F.5 Independent Repair from Frozen Initial Programs

We evaluate an independent repair stage applied to frozen initial IOL programs. Table 9 compares free-form test-based repair and trace-based repair on two cohorts of initial programs, generated by Sol at ultra reasoning effort and Luna at max reasoning effort.

Table 9: Independent repair from frozen initial IOL programs on 50 AuthorBench assets. Within each initial-program cohort, both repair methods independently revise copies of the same initial candidates using Sol/ultra, with IMG-on. Completed evaluations: completed/total assets. Scores average all 50 assets, with incomplete evaluations contributing zero. Bold marks the highest displayed value of each metric within a cohort.

Condition	Completed evaluations	Precision	Recall	F1	Accuracy	Trajectory match
Initial programs: Sol / ultra						
No repair	48/50	**0.868**	**0.889**	**0.854**	**0.871**	**0.786**
Free-form tests	49/50	**0.868**	0.869	0.828	0.860	0.766
Trace-based repair	49/50	0.843	0.887	0.821	0.852	0.762
Initial programs: Luna / max						
No repair	50/50	0.812	0.823	0.752	0.809	0.680
Free-form tests	50/50	**0.892**	0.871	0.835	**0.871**	**0.768**
Trace-based repair	50/50	0.891	**0.881**	**0.844**	**0.871**	0.766

For each asset within a cohort, both repair arms start in fresh sessions from copies of the same frozen one-pass program, without the original authoring conversation. All repairs use Sol at ultra reasoning effort. The repair agent both assesses and revises the program; no separate critic is used. Public asset context, few-shot examples, and visual input are held fixed within each cohort. These initial programs are generated separately from the visual-input ablation in Table 8.

One-pass authoring. The No-EC baseline generates code from scratch without compilation, execution, testing, or a check–repair loop during authoring. After submission, an external evaluator loads and replays the frozen program to determine evaluation completion and compute held-out metrics. Evaluation diagnostics are not returned to the authoring model.

Free-form unit tests. The Sol repair model writes and executes tests directly against the restricted-loader runtime, without a prescribed test format or common runner, and uses the feedback to revise the frozen candidate. The runtime's loading restrictions remain in force.

Trace-based repair. The Sol repair model samples candidate traces, assigns whole-trace expectations from public asset evidence, and replays them to guide revisions. Candidate-derived measurement diagnostics are also available. Final replay uses the final candidate and trace table. The comparison evaluates the complete repair workflows, including their additional inference, relative to the unmodified initial program.

F.6 Additional Related Work

Table 10: Selected approaches to interactive assets and object behavior. We compare their scope and representations with IOL's focus on portable operational rules and tools for LLM authoring.

Work	Nontrivial asset behavior	Where the logic lives	Relation to IOL
BEHAVIOR-1K / OmniGibson Li et al. (2023)	Object states, particles, fluids, slicing, dicing, melting, cooking, washing, drying, mixing, and toggled machines.	OmniGibson object-state APIs, simulator-side transition-rule classes, and recipe knowledge bases over synsets and systems.	Also uses conditional rules for object behavior. IOL focuses on portable asset-level specifications and controller-guided testing for LLM authoring.
iGibson 2.0 Li et al. (2022)	Household object states including temperature, wetness, cleanliness, toggled state, and sliced state.	Simulator object-state system.	Captures semantic state beyond geometry, but not as a portable operational specification for each asset.
AI2-THOR / ProcTHOR Kolve et al. (2017); Deitke et al. (2022)	Discrete interactions such as opening, closing, toggling, slicing, pickup, placement, and procedural scene variation.	Unity action APIs, object metadata, and task logic.	Provides useful discrete affordances, but the behavior is tied to simulator actions rather than asset-level update mediation.
VirtualHome Puig et al. (2018)	Household activity programs composed from atomic actions such as open, close, switch on, grab, put, and sit.	Symbolic programs executed in Unity.	Targets household activity programs; IOL specifies how an asset responds to attempted joint updates.
Habitat 2.0 / ReplicaCAD Szot et al. (2021)	Physics-enabled rearrangement with articulated furniture such as cabinets and drawers.	Physics simulation, articulated assets, and task predicates.	Strong articulation and simulation substrate; operational dependencies beyond joints are typically outside the asset.
SAPIEN / ManiSkill Xiang et al. (2020); Gu et al. (2023)	Large-scale articulated objects and manipulation tasks involving doors, drawers, tools, and appliances.	URDF-style articulation, physics simulation, and task-specific environment code.	Rich source of articulated assets and tasks; IOL targets state-dependent operation not captured by joints alone.
RLBench James et al. (2020)	Hand-designed manipulation tasks including buttons, drawers, doors, taps, ovens, and multi-stage procedures.	CoppeliaSim task scripts, waypoints, demonstrations, and success conditions.	Organizes scripted interactions around manipulation tasks; IOL packages operational rules with each asset.
ThreeDWorld Gan et al. (2021)	Rigid bodies, soft bodies, cloth, fluids, audio, deformables, and VR interaction.	Unity physics and controller APIs.	Broad physical simulation platform; object-specific operating procedures remain controller or simulator code.
EgoFun3D Peng et al. (2026)	Function templates for receptor–effector mappings, such as a knob controlling heat or a faucet handle controlling water.	Function templates inferred from egocentric video and compiled across simulation platforms.	Shares the goal of reusable functionality. IOL emphasizes temporal-rule synthesis, joint-update mediation, and trace-based authoring tools.
Real2Code / Articulate-Anything / Articulate AnyMesh Zhao et al. (2025); Le et al. (2025); Qiu et al. (2025)	Automatic reconstruction or generation of articulated digital twins from images, video, text, or multimodal input.	Generated articulation structures and simulator code.	Closely related acquisition goal, but primarily targets articulation rather than state-dependent operational behavior.

Table 11: **Authoring cost by representation.** Sol/ultra with agentic self-testing and IMG-on on 50 assets per condition. Token counts are per-asset means (M: million; K: thousand). Cache is the fraction of aggregate input tokens served from cache. Cost and time show per-asset mean / median.

Representation	Input (M)	Cache (%)	Output (K)	Cost (USD)	Time (min)
IOL	4.364	93.7	35.1	3.429 / 3.313	7.56 / 6.69
Update Function	2.810	93.0	36.6	2.563 / 2.483	7.20 / 6.22
SAPIEN Native	3.313	93.9	46.8	2.994 / 2.669	9.01 / 8.39

Table 12: **Authoring cost by workflow.** Sol/ultra with IOL and image input enabled on 50 assets per condition. Units and aggregation follow Table 11. The Naive-EC row reuses its IOL run from that table, matching the performance comparisons.

Authoring workflow	Input (M)	Cache (%)	Output (K)	Cost (USD)	Time (min)
No-EC	1.580	90.5	21.4	1.600 / 1.578	4.71 / 4.44
Naive-EC	4.364	93.7	35.1	3.429 / 3.313	7.56 / 6.69
TR-EC	4.628	92.8	46.4	3.978 / 4.002	10.42 / 9.61

G Inference Cost

G.1 AuthorBench

We report costs for the comparisons of representations and authoring workflows for Sol in Appendix F. Each condition contains the same 50 assets. Statistics include recorded usage from the author, its subagents, and retries, including failed attempts; the denominator remains 50 rather than the number of successful outputs. These are authoring costs, excluding held-out evaluation and downstream robot planning.

Costs are API-equivalent estimates using recorded usage and the pricing schedule in our September 21, 2026 snapshot, including prompt-cache discounts, rather than subscription charges. Input tokens include cached input; output tokens include reasoning tokens, without double counting. Time is recorded execution-attempt wall time summed over attempts for each asset, including controller overhead, tool execution, final validation, and waiting within each attempt. Gaps between attempts and concurrent subagent durations are not added separately.

Representation comparison. Table 11 reports the runs evaluated in Table 6 for Sol. All three representations use agentic self-testing and IMG-on. IOL costs an average of $3.429 per asset, including supplied execution support.

Authoring workflow comparison. Table 12 reports the Sol, IOL, IMG-on runs evaluated in Table 2. Costs include initial authoring and any subsequent repair. For Sol, trace-based correction increases mean total token usage by 6.3% and mean estimated cost by 16.0% over agentic self-testing (approximately $0.55 per asset). These are observed workflow costs, not comparisons under a matched budget.

G.2 OperateBench

Figure 5(b) uses estimated API costs for the selected 25-task sweeps whose outcomes determine the reported success rates. Each mean is the selected-sweep total divided by all 25 tasks, including unsolved tasks. Costs include failed attempts within those runs and child-agent calls, but exclude separate pilots, tests, and superseded runs.

The estimates use recorded billed charges and token usage matched by model and experiment timestamps. Charges for overlapping runs are split by author time, so the per-experiment allocation is estimated. The completed Luna supplement replaces the initial partial-export figures.

For privileged-state runs without initial IOL access, mean costs at low and highest effort are $1.1448 / $4.2372 for Astra, $1.3560 / $10.0240 for Sol, $0.7792 / $3.1632 for Terra, and $0.0596 / $0.1708 for Luna. Highest effort is max for Luna and ultra for the other models. The separate Astra/ultra RGB-D sweep costs $368.78 in total, or $14.7512 per task. The figure omits initial-IOL-access comparisons.

The handoff does not provide complete per-task token counts, cost distributions, or runtime records, nor a separate charge breakdown for final replays. Simulator admissions and deadlines are not measured inference costs or mean runtimes. The perception runs use 471 scored generation attempts, excluding the interrupted originals and final replays described in Appendix E.2.

H Dataset Collection and Usage

Asset sources. Across our experiments, the URDF assets are drawn primarily from three sources: PartNet-Mobility Xiang et al. (2020), Infinigen Joshi et al. (2025), and Articraft3D (Articraft) Zhou et al. (2026). These sources provide the underlying articulated objects, including their geometry, link structure, joints, and joint limits. We use this asset evidence as the basis for authoring object-level operational behavior.

Usage in the experiments. The collected assets support the two benchmarks introduced in Section 1. AuthorBench evaluates operational models on a fixed cohort of 50 assets using held-out interaction traces; OperateBench evaluates embodied agents on tasks involving operational objects. The source assets supply the geometry and articulation, while our benchmarks add the operational models, reference traces, and task specifications used in these evaluations. Within each authoring comparison, public asset evidence is held fixed across conditions, and reference programs, held-out traces, and evaluation results are excluded from the authoring context. Asset coverage is summarized in Appendix J.

I Limitations and Future Directions

Behavioral correctness. Trace-based correction compares execution with expectations supplied by the same LLM that writes the program. If the model misunderstands a mechanism, it may encode that misunderstanding in both the program and its expectations. Replay can then agree with an incorrect expectation without revealing the underlying error. Sampled traces may also miss critical interactions or run too briefly to expose timing errors. Agreement after repair therefore does not establish behavioral correctness. Future work could combine broader trace coverage with targeted human clarification of uncertain mechanisms or expected responses, using this input to revise expectations and guide further repair. The resulting accuracy gains and annotation cost remain to be evaluated.

Compilation cost. Our implementation compiles each object's temporal rules into a single controller over $N = |\mathcal{P}^{\mathrm{ext}}| + |\mathcal{P}^{\mathrm{int}}|$ Boolean propositions. Before applying specification constraints, their joint valuation space $\{0,1\}^N$ contains 2^N assignments. This count is not a compilation-time bound: symbolic synthesis need not enumerate every assignment. However, synthesis and subsequent controller enumeration can still become costly in time and memory as the number of propositions and their interactions grows. These costs occur during authoring and refinement; execution reuses the compiled controller without repeating synthesis at each simulation step.

Hierarchical Mealy machines. A possible extension is to nest Mealy machines: a state of an outer machine would be refined into an inner machine. For a washing machine, the outer machine could track idle, running, and finished modes. Entering running would activate an inner machine for the wash, rinse, and spin phases; its completion would trigger the outer transition to finished. Synthesizing each level separately could reduce compilation cost without flattening the hierarchy. Defining activation, completion, and reset semantics and validating correctness and compilation savings remain future work.

J Asset Coverage and Manifests

J.1 AuthorBench Assets

Coverage of the 50 Assets. AuthorBench comprises 50 assets across 23 subcategories. Table 13 summarizes their URDF joint types, while Table 14 reports the asset count and operational behaviors for each subcategory. The joint counts include 271 non-fixed joints after excluding 30 free-rolling chair wheel/caster joints.

Table 13: URDF joint types across the 50 AuthorBench assets.

Category	Revolute	Prismatic	Continuous	Total
Appliances	58	90	23	171
Furniture	2	8	3	13
Containers	15	18	7	40
Doors & Fixtures	25	17	3	45
Tools	0	1	1	2
Total	**100**	**134**	**37**	**271**

Revolute: bounded rotation; prismatic: bounded translation; continuous: unbounded rotation. Counts exclude fixed joints and 30 free-rolling chair wheel/caster joints (37 raw chair joints, 7 retained). The raw dataset total before wheel/caster exclusions is 301 non-fixed joints.

Table 14: Asset counts and operational behaviors for all 50 AuthorBench assets across 23 subcategories. Behaviors summarize the reference IOL specifications across variants.

Asset subcategory	Assets	Operational behavior
Dishwasher	4	Door-gated rack extension with reciprocal door-rack interlocks.
Washing machine	4	Selector or cycle gated door opening; timed tub rotation with a lid lock.
Microwave	3	Button-released door opening.
Safe[†]	4	Control or password gated door opening.
Toaster	4	Timer-gated carriage motion and timed knob return.
Door	4	Handle and lock gated opening of hinged door leaves.
Oven	3	Door-gated tray extension, with timer interlocks in some variants.
Security gate[†]	1	Password-triggered automatic sliding opening with STOP-button control.
Suitcase	4	Latch-released lid opening or button-released handle extension.
Refrigerator	4	Door-gated inner drawer extension with reciprocal door-drawer interlocks.
Bread maker	1	START/STOP button-controlled kneading-spindle rotation.
Chair	2	Lever-released seat-height adjustment.
Standing desk	1	Button-driven height adjustment with synchronized telescoping legs.
Toilet	2	Lid-gated seat lifting with a reciprocal lid-seat interlock.
Stand mixer	1	Lock-released head tilt; paddle rotation enabled by head and bowl alignment.
Toaster oven	1	Door-gated basket extension with a reciprocal door-basket interlock.
Spotlight	1	Independently lockable pan and tilt motion.
Tower fan	1	Speed and timer controlled blower rotation and body oscillation.
Box fan	1	Knob-controlled rotor rotation.
Sluice gate	1	Crank rotation coupled to vertical gate translation.
Trash bin	1	Pedal rotation coupled to lid opening and closing.
Bench vise	1	Handle rotation coupled to sliding-jaw translation.
Panini press	1	Free lid rotation and timed return of the timer dial.
Total	**50**	

[†]Object descriptions are empty except for the keypad safe (`r50_041`) and security gate (`r50_048`), which receive only the required keypad sequence and digit-to-joint mappings.

J.2 OPERATEBENCH ASSETS

Geometry and scene composition. OperateBench comprises 25 task-specific environments spanning lockable storage, household appliances, dispensers, crank-operated mechanisms, and adjustable furniture. Objects are represented as URDF models built primarily from boxes, cylinders, and spheres. Four tasks additionally use simple OBJ visual meshes for door panels or a drawer assembly; collision geometry uses primitives throughout. Scenes include supporting surfaces, movable props such as containers, cups, and household items, and target regions as needed. Manipulation uses a floating two-finger gripper derived from the Franka Panda hand.

Table 15 lists the task assets and their mechanisms and operational states. The inference and evaluation protocols are described in Appendices D and E.2, respectively.

Table 15: Asset coverage of the 25 OperateBench tasks. Each row lists the task's articulated asset, mechanisms, and operational states.

Task	Asset	Mechanisms and operational states
001	Latched door	Sliding latch, rotary handle, hinged door
002	Button-release cabinet	Release button and hinged door
003	Sliding-bolt cabinet	Sliding bolt, two hinged doors, rotating handles
004	Rotary-lock cabinet	Rotary lock tab and hinged door
005	Push-release drawer	Inward push release and sliding drawer
006	Dial safe	Rotary dial, handle, hinged door
007	Dual-latch suitcase	Two rotary latches and hinged lid
008	PIN-code locker	Ten keypad buttons, rotary wheel, hinged door
009	Child-lock container	Two sliding safety tabs and hinged lid
010	Microwave	Door-release button and hinged door
011	Toaster	Timer knob, lever, carriage, heating state
012	Washing machine	Door, start/release buttons, timer, cleaning state
013	Dishwasher	Door, sliding rack, release/start buttons, timer, cleaning state
014	Coffee machine	Pod loading, start button, brew timer, cup-fill state
015	Water dispenser	Press button and glass-fill state
016	Lever dispenser	Rotary lever, retracting shelf, collection box
017	Pedal trash bin	Pedal coupled to hinged lid
018	Crank window	Multi-turn crank coupled to sliding sash
019	Crank storage hatch	Crank transmission, two sliding bolts, sliding hatch
020	Adjustable desk	Up/down buttons and vertical lift
021	Powered cabinet lift	Power/up/down buttons and vertical platform
022	Fridge with drawer	Two hinged doors and inner sliding drawer
023	Height-adjustable chair	Adjustment lever and vertical seat travel
024	Toolbox	Ordered side slides, two front rotary latches, hinged lid
025	Medicine cabinet	Rotary guard tab, recessed safety button, hinged door

K AUTHORBENCH PROMPTS AND INPUT CONSTRUCTION

This section presents the authoring and refinement prompts used in our experiments.

Experimental inputs, model settings, and budgets are specified in Appendix C.

K.1 SHARED INPUTS AND GUIDANCE

The agent receives asset context, representation-specific authoring instructions and examples, and workflow instructions defining the available feedback. Asset-context fields and the optional description are specified in Appendix C; a description field or file can be present without containing supplied text. For image-enabled runs, a URDF multipose contact sheet provides additional visual context. The excerpt below shows how the agent is instructed to interpret this evidence.

A shared behavior contract provides modeling conventions. The visual-input ablation omits rendered asset views in the IMG-off condition.

Image guidance (selected instructions).

```
IMAGE CONTEXT:
- A neutral URDF multipose contact sheet is attached to this request.

- Because multiple joints may move together, use the sheet only for qualitative
visible geometry and motion plausibility. Do not attribute a change to one
joint unless textual or model evidence isolates it.
```

K.2 REPRESENTATION-SPECIFIC INSTRUCTIONS

Each representation receives seven executable examples alongside shared mechanism-inference guidance, modeling conventions, and public asset evidence. The representation comparison replaces the IOL output contract with the corresponding Update Function or SAPIEN Native interface (Appendix A.3). The following excerpts show the target artifact and state-access requirements from the representation-specific prompts. Each also receives the shared mechanism inference guidance and modeling conventions, public asset evidence, and its corresponding executable examples. The SAPIEN Native workspace additionally supplies an SDK reference; we include its lifecycle skeleton, state-access and actuation APIs, and execution semantics alongside the authoring brief. All three authoring targets are evaluated through the common backend transition interface for joint-level numerical evaluation (Appendix E.1).

K.2.1 IOL

The authoring prompt asks the agent to infer an object's operational mechanism from general knowledge and asset-specific evidence, then express it through propositions, temporal rules, and conditioned effects. The excerpts below retain the corresponding instructions. The seven executable examples provided in the prompt supply syntax and mechanism patterns; the prompt instructs the agent to adapt these patterns to the current asset rather than copy example-specific numerical constants. Interface details and shared numerical and observation conventions are included in the complete prompt.

Task and mechanism inference.

```
You are authoring a complete Pythonic IOL artifact for an articulated object.
The output must be directly executable by the Pythonic IOL backend.

- Use exact joint names from URDF_SUMMARY_JSON; do not invent joints.
- Infer probable operational behavior primarily from general world knowledge of
the named object and mechanism, grounded in the supplied asset evidence.
Explicit evidence overrides a conflicting general assumption.
```

```
- When several mechanisms remain plausible, choose the simplest conventional
behavior consistent with that evidence and record uncertainty in any requested
report. Missing descriptions alone do not rule out inferred behavior.
- Distinguish conditions observed in the current state from effects that
persist after those conditions cease. Represent required persistence within the
supplied state/history API; do not add persistence merely because a condition
was once true.

- Declare every prismatic, revolute, and continuous joint in URDF_SUMMARY_JSON
in Model, including joints with no AP or relation. Fixed joints need no Joint
declaration.
- Use APs only for joints whose state changes an operational precondition,
interlock, coupling, timer/cycle, persistent mode, or meaningful free object
state.
- Do not create APs merely because a joint is movable.
- Add observations or effects when the joint's state affects the inferred
behavior, regardless of whether it is a control, driven part, wheel, or
support. A declared joint with no applicable effect retains its proposed state.
- Prefer compact AP/logic/relation sets. If an AP is not used by logic or
relations and does not represent an independent meaningful state, omit it.
- Keep independent env APs minimal because controller synthesis branches over
their boolean combinations. Aggregate observations only when the mechanism does
not need to distinguish their identities or combinations. Preserve per-input
observations when logic needs their separate edges or history; edge detection
after aggregation can lose events.

- AP names should describe observed part roles and states when semantic
evidence is available.
- AP methods are stateless predicates over the current joint state. A predicate
may involve multiple joints when their combined state defines a relevant
observation.
- Do not encode memory, sequencing, blocking policy, or relation activation in
AP methods.
```

Propositions, temporal rules, and effects.

```
- Put observed propositions in @ap methods.
- Put operational control propositions and internal memory bits in sys_aps().
- Use memory sys APs when behavior depends on history, such as phases, input
sequences, or latched modes. Initialize them in sys_init and define their next
values with X(memory_ap) in sys_safety.
- Define current relation-facing outputs in sys_safety from observations and
current memory. Do not initialize observation-derived outputs as independent
memory.
- Use sys_init=[] when no controller memory is needed. Current guards read
current memory; X(...) updates become available on the next transition.

- Use Freeze when a joint must remain at its previous state while a blocking
condition is active; its guard means blocked.
- Use Coupling for mechanical driver-target projection.
- Use Timer or SteadyVelocity for autonomous time-like return/cycle behavior.
- Use Equation only when the relation cannot be expressed by the syntactic
sugar descriptors.
- If the asset only needs AP observation and no state modification, omit
@relation methods entirely.
```

K.2.2 UPDATE FUNCTION

The Update Function prompt specifies a stateful Python interface for mediating joint updates.

Update Function: output and state interface.

```
TASK: UPDATE_FUNCTION

You are authoring an Update Function for an articulated
URDF-style asset. Generate one Python implementation.

OUTPUT FORMAT:
- Output Python source only.
- Do not use markdown fences.
- Do not include prose outside Python.
- Define exactly one top-level class named ObjectMediator.
- Do not output YAML.

REQUIRED PYTHON INTERFACE:
class ObjectMediator:
    def __init__(self, metadata):
        self.joint_names = tuple(metadata["joint_names"])

    def reset(self):
        ...

    def resolve(self, prev, proposed, dt):
        ...
STATE API:
- prev is the current object state X_t.
- proposed is the caller-proposed next state after applying Delta X_t.
- dt is a finite non-negative float timestep and may be zero. Guard any
  division by dt.
- prev and proposed are dictionaries keyed by exact joint name:
  prev = {
      "<joint_name>": {"q": float, "qdot": float, "qddot": float},
      ...
  }
- resolve must return the final resolved next state with exactly the same joint
  names and q/qdot/qddot fields.
- Returned q/qdot/qddot values must be finite, non-Boolean real numbers; NaN
  and infinity are invalid.
- Start resolve with a fresh copy of proposed:
  out = {name: dict(proposed[name]) for name in self.joint_names}
- Do not reject otherwise finite numeric input solely because it lies outside
  a listed URDF joint limit.
```

K.2.3 SAPIEN NATIVE

The SAPIEN Native instructions specify the artifact lifecycle, state access, actuation APIs, and native execution semantics. Held-out scoring uses the numerical replay protocol in Appendix E.1.

SAPIEN Native: authoring brief and API.

```
# SAPIEN-native authoring brief

Author a multi-file Python artifact under `work/candidate/` using SAPIEN's
native programming model. `entry.py` must define `create(articulation, *, dt)`
and return a `sapien.System`; the controller registers that system and invokes
`step()` before each physics step.

Infer the intended whole-object behavior from the frozen object description,
asset summary, URDF, multipose image, static joint metadata, and general object
knowledge. The controller does not prescribe a behavior representation: choose
helper modules, internal episode-local state, and native actuation mechanisms
as needed. Author the behavior itself; do not reproduce an evaluator or a
visible input trajectory.

Dynamic simulator reads are limited to the target articulation's `q`, `qdot`,
and `qddot`. Static joint metadata and immutable `dt` may be read during
construction. The candidate does not receive the scene, global clock,
intervention schedule, evaluator predicates, or expected semantic end states.
Any target-articulation native actuation is allowed, including direct q/qdot
writes and drives. Do not implement an IOL spec or a repository-defined update
function.
import sapien

class CandidateSystem(sapien.System):
    def get_name(self):
        return "candidate_behavior"

    def step(self):
        ...

def create(articulation, *, dt: float):
    return CandidateSystem()
Tested target-articulation reads include `get_active_joints()`, `get_qpos()`,
`get_qvel()`, and `get_qacc()`. Joint names are available through
`joint.get_name()`. Allowed target-articulation actuation includes
`set_qpos(...)`, `set_qvel(...)`, and joint drive methods such as
`set_drive_properties(...)` and `set_drive_target(...)`.

Execution semantics:

- `CandidateSystem.step()` runs immediately before the simulator physics step.
- `set_qpos(...)` and `set_qvel(...)` write the current pre-physics state. They
do not define a persistent control law or guarantee `q_next = q + qdot * dt`;
constraints, drives, damping, contacts, and articulation dynamics may change
`q`, `qdot`, and `qddot` during the physics step.
- Joint drives are physics-mediated, while direct `q`/`qdot` writes are state
overrides. Both are allowed by this authoring treatment.
- Reason about rollout assumptions using actual SAPIEN physics semantics. A
tensor-only mock that manually integrates `q += qdot * dt` is not equivalent to
a SAPIEN rollout.

This reference describes only the public lifecycle, API shape, and execution
semantics. It contains no current-asset solution or expected rollout result.
```

K.3 AUTHORING WORKFLOWS

These excerpts specify the feedback available during authoring. No-EC requires a candidate without execution or diagnostic feedback. Naive-EC allows the agent to design and run local checks and decide whether to revise or stop, without a supplied semantic oracle or TR-EC feedback tool.

The main correction comparison varies these workflows in independent authoring sessions (Table 2); Section K.4 describes the additional instructions for repair from a supplied initial program.

K.3.1 ONE-PASS AUTHORING

One-pass authoring (No-EC) requires a candidate without execution or diagnostic feedback during authoring. The subsequent evaluation does not provide a correction opportunity.

No-EC (selected instructions).

```
Read the frozen inputs and API/examples, reason about the behavior, and write
one final candidate.

Do not execute, import, compile, test, measure, sample, or replay the candidate.
Do not run self-authored checks or ask a reviewer to check generated code.
Do not seek diagnostics or perform a generate/check/correct loop.

The controller checks the frozen output only after this session ends; those
results are never returned for authoring correction.
```

K.3.2 AGENTIC SELF-TESTING

Agentic self-testing (Naive-EC) allows the agent to design and run local checks and decide whether to revise or stop, without a supplied semantic oracle or trace-correction tool. Self-written checks are optional in this authoring workflow.

Naive-EC (selected instructions).

```
- You may freely write and run ordinary, self-authored local checks under work/.
- Decide whether to check, whether and how to repair, and when to stop.
- No experiment-derived semantic oracle or trace-EC feedback tool is available.
```

K.3.3 TRACE-BASED CORRECTION

Trace-based correction (TR-EC) asks the agent to record an evidence-grounded operational expectation for each selected trace, then compare it with replay evidence to decide whether to revise the candidate or reconsider its interpretation. Validity refers to conformance of the complete proposal sequence (Eq. 4).

The reported comparisons use this workflow without a separate critic.

Trace interpretation and expectations.

```
Returned traces are unlabeled candidate-runtime probes. Their selection and
construction do not establish intended semantics, correctness, or ground truth.

- hypothesized_outcome is valid or invalid for the complete trace. valid means
all resolved q match the original proposed q within absolute tolerance 1e-6 at
every step, including holds. qdot/qddot remain inputs but are not scored. One q
deviation makes the trace invalid; later recovery does not erase it. Describe
useful steps/joints in hypothesis/rationale.

- Before replay, set each row's hypothesis and hypothesized_outcome to your
best-supported operational judgment for that complete trace, not to a
```

```
prediction of the current candidate. Base the judgment primarily on general
world knowledge of the named object and mechanism, then ground it in supplied
input evidence. World knowledge is a fallible prior: explicit evidence
overrides a conflicting general assumption, and you must not fabricate precise
latent mechanisms or parameters unsupported by the available evidence.
- In rationale, cite the input cue or general mechanism assumption supporting
the judgment; do not use candidate code, sampler provenance, or predicted
runtime output as semantic support. Omit an ambiguous trace rather than force a
label. Prefer nonredundant rows that could reveal both over-permissive and
over-restrictive behavior when available.
```

Replay analysis and revision.

```
- After replay, read the tool's compact summary of all rows and first
violations. Read its full_result file only when you need detailed validation or
per-step conformance. Compare observed_outcome with the pre-replay operational
judgment. hypothesis_satisfied means only that the current candidate matched
your judgment; it does not validate that judgment. Expected invalid and
observed invalid is a match. Replay errors have no validity or hypothesis
match. Inspect the displayed first_violation for the actual deviation (also
saved in q_conformance.first_violation). If they differ, revise the candidate
when the judgment remains supported. Revise the table only when its cited
evidence was misread or insufficient; never change a hypothesis merely to match
observed_outcome. Then replay again.

For each retained row, concisely record the expected operational outcome and
supporting evidence, the observed candidate outcome, any uncertainty, and the
resulting decision. If no expectation was supportable, record the reviewed
batch and reason instead. List important sampled traces skipped as ambiguous.
Replay artifacts are authoritative only for current-candidate execution, never
for intended semantics.

After revisions, retain earlier samples as regressions in the final table and
replay that table on the final candidate. New author rows may use later
batches. Preserve original sampled inputs and expectations; do not relabel
their candidate provenance.
```

K.4 Repair from Frozen Initial Programs

The repair prompts adapt the authoring workflows in Section K.3 to the frozen-program comparison in Table 9. Within each initial-program cohort, both methods receive copies of the same initial IOL source for each asset in fresh sessions using Sol/ultra. The shared task instruction identifies this source as a fallible implementation to be repaired.

Shared repair task.

```
Goal: repair the supplied work/candidate.py for the public case in input/.
Its frozen original is input/initial_candidate.py; the initial candidate is a
fallible implementation, not semantic authority.
```

K.4.1 FREE-FORM TESTS

Unlike optional self-testing during authoring, this condition requires the agent to write and execute its own tests against the supplied candidate. It uses the restricted-loader runtime, with no prescribed test format or common runner.

Required self-written tests.

```
Before submitting, freely design, write, and run unit tests under work/ against
the actual work/candidate.py. Choose the test code, cases, assertions,
organization, and commands yourself; no experiment-specific test schema or
runner is provided.

Ground expected behavior in public evidence rather than candidate output, use
the results to improve the candidate, and rerun useful tests after changes.
Running these self-authored checks is required; passing them is not ground
truth, semantic acceptance, or a correctness guarantee.
```

K.4.2 TRACE-BASED REPAIR

The agent starts from the supplied initial IOL program and applies the trace interpretation and revision instructions in Section K.3.3.

L OPERATEBENCH PROMPTS

This section reproduces the complete policy-generation prompt templates for the state-based and perception workflows described in Appendix D. We preserve the original wording, runtime identifiers, and placeholder tokens, with line wrapping adjusted for readability. Tokens delimited by `@@` are filled with the task identifier, run identifier, simulator seed, helper-agent limit, simulator-attempt limit, and authoring deadline for each run.

L.1 STATE-BASED POLICY GENERATION

```
Generate one policy for OPOB task @@TASK_ID@@ in experiment run @@RUN_ID@@.

Scientific condition

- Ground-truth access is forbidden. No ground-truth policy, solution,
  demonstration, prior policy, prior run, or result artifact is available.
- Determine the task only from the approved files under `/work/inputs`.
- You may use collaborating agents. The configured child-agent limit is
  @@AGENT_LIMIT@@.
- Generated commands cannot access the host repository or the network.
- You may create debugging files under `/work/scratch` and candidate policies
  under `/work/submission`.
- One simulator attempt is one admitted execution of a candidate policy in a
  freshly reset simulator episode. You and all child agents share one limit of
  @@GENERATION_SIMULATOR_ATTEMPT_LIMIT@@ generation-time simulator attempts.
- An attempt is charged before its simulator starts and is never refunded if
  the trial fails, is interrupted, or crashes. Only one trial can run at a time
  for this task, shared across you and all child agents.
- This run ends only on the first trusted simulator success or after every
  allowed attempt has completed. A voluntary early final response does not
  submit a policy; the controller resumes this same authoring session.

Task inputs

Start with `/work/inputs/EVALUATION_SPEC.json`. It identifies the canonical
BDDL, scene, and gripper files and contains trusted simulator configuration.
It is evaluator configuration, not a policy or solution. Inspect the referenced
BDDL, YAML, URDF, mesh, and other approved files to determine the initial
condition, goal, mechanics, geometry names, and useful joint origins.

Submission

Write each complete candidate at:

`/work/submission/plan.py`

Submit it for testing with the command below. The controller snapshots its
exact bytes for that attempt. Wait for the full result before revising or
starting the next test; do not cancel or background a trial to replace it.
Submission is a harness-controlled handoff: the author, helper agents, and
their background commands are stopped before the simulator episode starts.
After a failed completed episode, this same session resumes with the complete
trusted evaluator feedback already supplied as input. The interrupted client
command is expected; it does not cancel the host-owned simulator episode.
Old queued requests cannot start a new episode in the resumed turn. On success
there is no further author turn. Save all useful notes in the workspace before
submitting; in-flight helper agents/commands will not continue across handoff.

The file uses the restricted `opob-sandbox-dsl-v1` Python syntax. It may
contain literals, lists, tuples, dictionaries, simple variable assignments,
direct calls to the functions below, comparisons using only `==`, `!=`, `is`,
and `is not`, boolean expressions using `and`, `or`, and `not`, and simple
`if` statements. Arithmetic is unavailable, so precompute numeric literals
while authoring. It may not contain imports, attribute access, subscripts,
loops, function or class definitions, exception handling, or calls to any
other name. The policy must be strict UTF-8, at most 16 MiB, and may contain
```

````
at most 10,000 AST nodes and 256 static API-call expressions.

Policy API

```python
move_to(
    location,
    approach_direction,
    orientation="vertical",
    *,
    soft_contact_obstacles=None,
    lock_orientation=False,
)
```

Moves the floating gripper toward a named scene/motion target, a named
collision geometry, or a literal XYZ position. `approach_direction` is a
signed axis (`"x"`, `"+x"`, `"-x"`, `"y"`, `"+y"`, `"-y"`, `"z"`,
`"+z"`, or `"-z"`) or a nonzero XYZ vector. `orientation` is `"vertical"`
or `"horizontal"`. `soft_contact_obstacles` is a string, list, or tuple of
bounded collision-name patterns. Returns a boolean success value.

```python
grasp_until_contact(min_width=0.012, steps=80)
```

Closes the gripper until contact. Returns a boolean success value.

```python
slide(distance, direction, *, steps=None)
```

Translates the gripper by `distance` along a signed axis or nonzero XYZ
direction. Returns a boolean success value.

```python
open_gripper(width=0.035, steps=40)
```

Opens the gripper.

```python
rotate_about(origin, axis, degrees, *, steps=60)
```

Rotates the gripper about a literal XYZ origin or a named joint origin in
`"object/joint"` form. `axis` is a signed axis or nonzero XYZ vector.

```python
goal_reached()
summarize()
```

`goal_reached()` returns the trusted success boolean. `summarize()` returns
the current simulator summary, which may be assigned or included directly in
`RESULT` but cannot be indexed under the restricted syntax.

Numeric inputs must be finite. XYZ components must be between -100 and 100,
gripper widths between 0 and 1, slide distance between -10 and 10, rotation
degrees between -36,000 and 36,000, and step counts integers from 0 through
100,000. The evaluator scores native simulator state and does not trust the
value of `RESULT`.

Simulator testing

`/runtime/bin/evaluate-policy` is the only permitted simulator entry point for
this experiment. Do not import, execute, or drive raw simulator/runtime
internals directly.
````

````
Test `/work/submission/plan.py` against a fresh headless simulator:

```bash
mkdir -p /work/scratch/mesa
POLICY_SHA256="$(sha256sum /work/submission/plan.py | cut -d ' ' -f1)"
env \
  PYTHONDONTWRITEBYTECODE=1 \
  LIBGL_ALWAYS_SOFTWARE=1 \
  VK_ICD_FILENAMES=/usr/share/vulkan/icd.d/lvp_icd.x86_64.json \
  MESA_SHADER_CACHE_DIR=/work/scratch/mesa \
  /runtime/bin/evaluate-policy \
  --protocol-version 1 \
  --task-id @@TASK_ID@@ \
  --run-id @@RUN_ID@@ \
  --plan-id @@RUN_ID@@:authoritative \
  --policy /work/submission/plan.py \
  --policy-sha256 "$POLICY_SHA256" \
  --policy-format opob-sandbox-dsl-v1 \
  --input-root /work/inputs \
  --workspace-manifest /work/WORKSPACE_MANIFEST.json \
  --seed @@SEED@@ \
  --headless \
  --no-video
```

Every invocation starts from the original scene. Simulator diagnostics appear
on stderr and the final JSON envelope appears on stdout. Use
`simulator_summary.success`, not `RESULT`, to judge a trial. Simulator steps
remain measured telemetry, not a shared generation-time ceiling; each admitted
episode still has its native per-episode limits.

After a completed failure, use its feedback to revise and test again. You must
continue until success or all @@GENERATION_SIMULATOR_ATTEMPT_LIMIT@@ attempts
are used. Reasoning, edits, API calls, and child-agent work do not themselves
consume simulator attempts. There is no separate revision-count limit.

The controller rejects overlapping tests with exit code 74 without charging an
attempt; wait for the current test's result before retrying. It never cancels
that test in favor of a new candidate. Read-only trusted progress is available
at `/runtime/control/generation-attempt-controller.json`.

On the first trusted success, the controller stops authoring and preserves
that exact tested policy automatically. If all attempts fail, it preserves the
last completed tested policy. Do not make an untested final revision or run
extra confirmation tests. Requests after either terminal outcome are rejected
without starting another episode. The separate authoritative replay checks
the frozen policy for consistency and does not consume an authoring attempt.

The @@GENERATION_TIMEOUT_SECONDS@@-second wall-clock ceiling remains a safety
limit. A hang, crash, or interrupted trial is reported separately, not as a
normal successful run or a normally exhausted attempt budget.
````

L.2 Visual Policy Generation

```
Write and iteratively test a visual policy for task @@TASK_ID@@, run @@RUN_ID@@.

Your task goal is in /work/inputs/VISUAL_TASK.json. The other task information
available to you consists of real camera images and the visual tool feedback.
The fixed external camera provides synchronized RGB and depth; diagnostic goal
region overlays are hidden. You receive no task BDDL, scene YAML, URDF, object
specifications, hidden IOL rules, named simulator joints, or target coordinates.
The trusted simulator retains its original mechanics and scores the task goal.

The simulator seed is @@SEED@@. You may use collaborating agents; the configured
```

````
child limit is @@AGENT_LIMIT@@. All agents must inherit the same model and
reasoning settings. The authoring safety deadline is
@@GENERATION_TIMEOUT_SECONDS@@ seconds. You have at most
@@GENERATION_SIMULATOR_ATTEMPT_LIMIT@@ simulator attempts, executed serially.
A verified success stops authoring and preserves the successful policy. If no
attempt succeeds, use the available attempts to revise your policy. The selected
policy receives a separate authoritative replay after authoring ends.

Images and feedback

An initial real RGB image is attached to your authoring turn. Latest images from
the previous completed attempt are attached when the same session resumes.
You may also inspect the PNG paths listed in
/runtime/control/generation-visual-images.json using image viewing tools.
These are bounded real renderings, not generated illustrations. Camera images
preserve native RGB detail up to 1280 by 720 pixels. No camera-oracle
segmentation or object IDs are
exposed. Image viewing is optional; scene targets in policies must still use
perception-derived geometry.

Read /runtime/control/generation-completed-feedback.json after an attempt.
It reports task success or failure and bounded visual/action evidence without
hidden rule descriptions or named simulator state. Use the observed images,
perception results, and motion outcomes to adjust the next complete policy.
Observation feedback includes camera_intrinsics, camera_to_world, depth_units,
and camera_coordinate_axes: calibrated sensor information, not object poses.
The fixed visual camera eye is elevated by 0.60 meters along world +z from the
scene camera; its target, field of view and sensor resolution are unchanged.
The reported calibration describes this actual camera. Intrinsics apply to the
observation width and height; a native 1280 by 720 observation is attached at
that same resolution. Larger observations are scaled down to the image bounds.
Check the image dimensions and account for any scale and the stated camera-axis
convention when interpreting pixel locations or choosing world-axis directions.
Action-result records provide primitive success booleans and simulation steps.
A primitive success is distinct from authoritative task-goal success.
The initial pure camera capture does not consume an attempt. Every actual
simulator trial does, including trials whose policy fails.

Submission and simulator handoff

Keep a complete candidate at /work/submission/plan.py. Use the
/runtime/bin/evaluate-policy handoff after writing it. This pauses
your author turn while trusted orchestration runs the policy, records feedback,
and resumes the same session. Do not invoke a separate simulator or attempt to
reconstruct hidden task files. You can write scratch files under /work/scratch.
The authoring sandbox has no network or host-repository access.

Submit the completed policy with this exact command structure:

```bash
POLICY_SHA256="$(sha256sum /work/submission/plan.py | cut -d ' ' -f1)"
/runtime/bin/evaluate-policy \
  --protocol-version 1 \
  --task-id @@TASK_ID@@ \
  --run-id @@RUN_ID@@ \
  --plan-id @@RUN_ID@@:authoritative \
  --policy /work/submission/plan.py \
  --policy-sha256 "$POLICY_SHA256" \
  --policy-format opob-visual-dsl-v1 \
  --input-root /work/inputs \
  --workspace-manifest /work/WORKSPACE_MANIFEST.json \
  --seed @@SEED@@ \
  --headless \
  --no-video
```

Every attempt starts from the original scene. Wait for completed feedback before
revising; author processes and helpers are stopped at handoff and resume through
````

```
the coordinator. A tool return value is not authoritative task success.

Policy syntax

The policy is restricted opob-visual-dsl-v1 Python: literals, lists, tuples,
dictionaries, simple assignments, direct calls to the APIs below, comparisons
using ==, !=, is, is not, boolean and/or/not, and simple if statements. It has
no imports, attribute access, subscripts, arithmetic, loops, function/class
definitions, exception handling, or arbitrary calls. Precompute numeric
literals. There are at most 256 static API-call expressions.

Visual tools

    frame = observe_scene()
    detection = segment_object(frame, "short visible object or part
description", candidate_index=0)
    geometry = estimate_object_geometry(frame, detection)

The frame is an opaque token for synchronized RGB-D from the fixed camera.
SAM 3.1 returns an opaque detection token for your bounded text query.
The optional candidate_index selects a stable score rank from 0 through 7;
the default 0 is the top usable detection. A missing rank is a perception miss.
Depth fusion returns opaque geometry for the visible surface. A policy cannot
inspect these tokens. A detection and geometry must refer to the same frame.
Use a new observation after scene-changing actions. Missing/invalid perception
fails the candidate; no privileged-state fallback is available.

Motion tools

    move_to(geometry, approach_direction, orientation="vertical",
            lock_orientation=False, offset=None)
    grasp_until_contact(min_width=0.012, steps=80)
    slide(distance, direction, steps=None)
    open_gripper(width=0.035, steps=40)
    rotate_about(geometry, axis, degrees, steps=60, offset=None)

move_to approaches perception-derived geometry and returns a boolean.
rotate_about uses perception-derived geometry as its pivot. Their optional
literal XYZ offsets adjust that geometry by at most one meter per component.
An axis/direction is a signed axis (x, +x, -x, y, +y, -y, z, +z, -z) or a
nonzero XYZ vector. Orientation is vertical or horizontal. grasp_until_contact
and slide return booleans; open_gripper opens the gripper. These are bounded
motion primitives for a floating gripper. Root motion uses physical velocity
control without an obstacle-aware path planner. Objects can block or deflect
movement. Choose approach directions and intermediate targets from observations.
The controller receives no object collision meshes, task-specific contact
exceptions, or carried-object collision proxies. There is no automatic pre-grasp
route. There is no separate learned grasp generator.

Use finite numeric values: offset components in [-1,1] meters; each finger's
gripper width in [0,0.04] meters; slide distances in [-10,10] meters; rotation
degrees in [-360,360]. Finger-action integer step counts are in [0,600]; root
motion step counts are in [0,2400]. Each root-motion action consumes at most
2400 physics steps including final settling, with maximum linear speed 0.18 m/s
and angular speed 1.25 rad/s. Motion completion requires position error at most
1 cm and orientation error at most 8 degrees. Long or blocked requests can fail.
Do not use named simulator motion targets,
named joints, collision patterns, or literal world coordinates as direct
motion targets. There is no goal_reached() or summarize() policy function.

Plan from the task goal and real observations. Choose visible objects or parts
for segmentation, use their geometry for motion, and revise from completed
attempt feedback. Leave your selected complete policy at
/work/submission/plan.py for the independent final replay.
```