MARC: MECHANISM-AWARE REFLECTIVE CORRECTION FOR ARTICULATED OBJECT CODE GENERATION

Anonymous authors
Paper under double-blind review

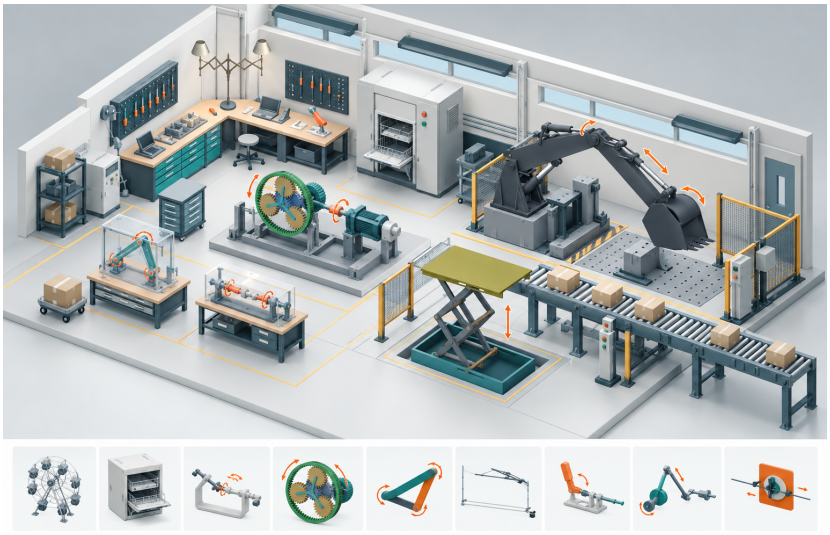

Figure 1: **Examples of articulated mechanisms generated with MARC.** A workshop scene brings together diverse linkages, transmissions, and everyday articulated objects.

ABSTRACT

Generating functional articulated objects from images and text requires part geometry and physical connections that jointly support the intended motion. Coupled and closed-chain mechanisms are especially difficult: an assembly of plausible parts and joints may still separate, lock, or move incorrectly when driven. We present MARC, an agent harness built around an executable kinematic graph. The graph represents rigid bodies, socket-bound connections, loop closures, and joint-coordinate couplings. An agent authors and revises these declarations through an SDK, while numerical tools assemble the mechanism, solve follower motion under prescribed driver inputs, and return diagnostics tied to specific connections and configurations. Evaluation and reflective correction use this evidence to locate failures and revise the implicated connections or geometry; each revised program is retested along the same prescribed paths. On a 20-object test set, MARC with Kimi-K3 achieves 67.8% valid motion coverage and 60.0% overall success, exceeding the evaluated baselines. Ablations support the contributions of reflective correction, graph-based mechanism support, and graph-guided repair.

1 INTRODUCTION

Articulated 3D assets support robot learning, manipulation planning, and interactive simulation, where functionality depends on connected parts moving as intended (Xiang et al., 2020). Generating such

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.

assets from reference images and textual requirements requires specifying part geometry, physical connections, and the relations between joint coordinates. This becomes particularly challenging when connections form closed loops or joint motions depend on one another. A scissor lift, for example, must coordinate the rotation of multiple links under a single driving input to raise its platform while preserving the hinged connections. Plausible part geometry and individually movable joints do not ensure that the assembled mechanism can perform this motion: an assembly that matches the reference in a static pose may separate, collide, or lock when driven. The central challenge is therefore to make geometry and connections jointly support the intended motion of the complete mechanism.

Existing methods have made progress in generating part geometry and articulation structure (Liu et al., 2024; Wang et al., 2026a). Program-generating agents further express asset construction as editable code and iteratively revise it through execution, inspection, and feedback. Articraft provides an SDK for geometry and articulation authoring together with structured validation (Zhou et al., 2026), while Procedura and aDSL combine assembly or spatial relations with execution feedback and visual assessment to identify and correct generated defects (Lin et al., 2026; Wang et al., 2026b). For closed-chain and coupled mechanisms, however, motion failures often involve several interdependent connections. Motion blocked at a passive joint may result from incorrect connectivity, attachment placement, joint limits, or geometric interference, and a local edit may disrupt other connections. Visual discrepancies and individual checks can reveal these failures without identifying which connections should change and which motion constraints should be preserved. Generation and correction therefore require relating the mechanism's connections and kinematic constraints to execution evidence, both to determine whether the assembly can perform the required motion and to guide repairs.

We introduce MARC, *Mechanism-Aware Reflective Correction*, an agent harness built around an executable kinematic graph for generating articulated-object programs from reference images and text. Building on Articraft's programmatic authoring interface, MARC binds physical connections to local attachment frames, or sockets, on the participating rigid bodies. The agent declares connection types and joint-coordinate couplings, from which numerical tools construct kinematic constraints. This representation provides a common basis for constructing, testing, and revising a mechanism. Given prescribed driving inputs, the tools solve for the remaining joint coordinates and associate diagnostic findings with the relevant parts, joints, sockets, and configurations. Using this evidence together with the reference, the agent investigates possible failure causes through targeted probes and develops repair plans specifying what to modify and what to preserve. Each revised program is then tested along the same prescribed paths, allowing its effects on the complete mechanism to be assessed.

We evaluate MARC on articulated-object generation tasks with reference images and text, measuring executable rate, structural correctness, valid motion coverage, coupled-motion correctness, and overall success rate. On the current 20-object evaluation, MARC with Kimi-K3 achieves 67.8% valid motion coverage and a 60.0% overall success rate. Starting from the same initial candidates, reflective correction improves these scores from 54.1% and 40.0%, respectively, corresponding to gains of 13.7 and 20.0 percentage points. These results support the benefit of reflective correction for extending valid motion and improving task completion within the evaluated collection. Comparisons with existing generators and analysis of remaining closure, limit, and interference failures further characterize the capabilities and limitations of the approach.

Our method makes three contributions:

- An agent harness for articulated-object generation. MARC uses an executable kinematic graph to connect program construction, motion solving, and reflective correction for coupled and closed-chain mechanisms.
- A shared representation for construction and motion solving. Physical connections bind local attachment frames to part geometry, while connection types and coordinate couplings specify kinematic constraints. Numerical tools solve for motion and associate constraint violations, joint limits, discontinuities, and interference with the relevant parts and connections.
- Graph-guided evaluation and reflective correction. The reference, kinematic graph, and execution diagnostics guide targeted probes and agent-generated repairs, which are tested again under the original motion requirements.

2 Related Work

Learning-based articulated object prediction and generation. Learning-based methods model part structure, geometry, and articulation. NAP and CAGE use graph representations and diffusion (Lei et al., 2023; Liu et al., 2024); MIDGArD separates graph and shape generation, while ArtFormer uses a token tree (Leboutet et al., 2024; Su et al., 2025). URDFormer and SINGAPO infer articulated assets from images, while PhysX-Anything generates physical assets from images (Chen et al., 2024; Liu et al., 2025; Cao et al., 2026). Particulate infers articulation from static meshes, and Instruct-Particulate adds kinematic control (Li et al., 2026a;b). ArtLLM and URDF-Anything+ autoregressively generate parts and joints; ArtLLM also applies physics-based joint-limit correction (Wang et al., 2026a; Wu et al., 2026). MARC uses an executable kinematic graph across construction, motion solving, and subsequent correction.

Agentic 3D asset code generation. Real2Code reconstructs articulated objects as code, and Articulate-Anything combines mesh retrieval with visual actor–critic refinement (Mandi et al., 2025; Le et al., 2025). Articraft provides a geometry and articulation SDK, a restricted harness, and structured feedback; its SDK underlies our authoring interface (Zhou et al., 2026). Procedura uses typed mates, assembly checks, visual refinement, and simulator validation; aDSL couples relational operators with a Plan–Execute–Critic workflow (Lin et al., 2026; Wang et al., 2026b). Related agent work studies tool interfaces and iterative feedback (Yang et al., 2024; Madaan et al., 2023), while 3DCodeBench studies procedural modeling and human preferences (Gao et al., 2026). MARC uses execution diagnostics associated with parts, connections, and kinematic constraints to guide repairs, then retests closure and intended motion along prescribed paths.

Closed-chain kinematics and computational mechanism design. Rigid-body kinematics characterizes compatible mechanism configurations, and graph-based geometric constraint solving provides computational tools (Lynch & Park, 2017; Fudos & Hoffmann, 1997). Mechanism-design systems jointly consider connectivity and continuous parameters to realize target motion (Coros et al., 2013; Thomaszewski et al., 2014). LinkEdit uses symbolic kinematics to preserve selected functional properties when editing a working planar linkage (Bächer et al., 2015). MARC applies these principles to generation from reference images and text, where the initial mechanism may be disconnected, immobile, or inconsistent with the reference. Numerical execution supplies evidence for diagnosis and revision.

3 Method

Given reference images I, a textual description x of the object and its intended behavior, and a target motion domain $\mathcal{U}$, the goal is to generate an executable program P that constructs an articulated object matching the required appearance, connections, and motion. Let $u \in \mathcal{U}$ denote a prescribed driver input that fixes one or more joint coordinates, and let q_f denote the remaining follower coordinates determined by the mechanism's constraints. The task specifies paths through $\mathcal{U}$ along which motion is tested. In the four-cylinder crank–slider of Figure 2, u is the crankshaft angle, while q_f includes the piston displacements and connecting-rod angles. Generating this mechanism requires both part geometry and the connections that couple these motions.

MARC is an **agent harness built around an executable kinematic graph**. It equips a multimodal LLM with an SDK for constructing mechanisms, numerical tools for motion solving and targeted probes, and a workflow for evaluation and correction. The agent writes connection and coupling declarations, interprets tool results, and chooses edits. The tools construct the corresponding kinematic constraints, solve for motion, and measure constraint violations. The harness maintains the current program, execution state, and diagnostic evidence for subsequent agent decisions. The kinematic graph connects these decisions to the same rigid bodies, connections, and sockets throughout the workflow.

As illustrated in Figure 2, the **Topology** role first constructs a mechanism program from the task inputs using the SDK. The **Solver**, a numerical component, assembles the mechanism and returns motion results and diagnostics. The **Evaluate** role compares these results and rendered views with the reference requirements. If correction is needed, the **Reflect** stage comprises three agent roles: **Judge** investigates possible causes through targeted probes and produces a repair plan, **Executor**

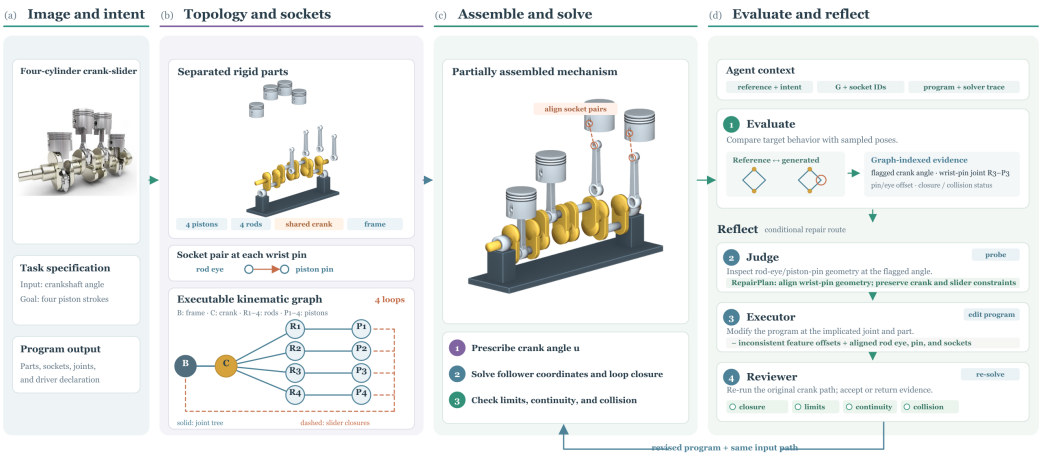

Figure 2: **MARC overview on a four-cylinder crank–slider.** (a) Input and target motion. (b) Parts, socket pairs, and four closed loops. (c) Solver assembly and driven motion. (d) Graph-indexed evidence guides Judge, Executor, and Reviewer; the revised program is tested on the same path. The socket edit is schematic.

implements the plan, and **Reviewer** checks the edits. The revised program returns directly to the Solver for testing along the original paths, followed by renewed evaluation. The process allows at most B_{corr} correction rounds. Sections 3.1–3.3 describe the representation, numerical tools, and evaluation and correction workflow, respectively.

3.1 An Executable Kinematic Graph for Mechanism Authoring

The agent uses this representation to construct and revise mechanisms. Articraft provides a URDF-based interface for declaring parts, joints, and mimic relationships (Zhou et al., 2026). A joint tree and affine coordinate couplings do not, by themselves, represent the additional physical connections or solve the constraints of a general closed chain. Connections must also specify where the participating bodies attach, so that geometric edits remain consistent with their permitted motion. We therefore extend the representation through physical connections and their attachment frames.

Connections and kinematic constraints. The kinematic graph is

$$G = (V, E_T \cup E_L), \tag{1}$$

where V contains rigid bodies, E_T is the joint tree used to compute body poses recursively, and E_L contains additional physical connections that close loops. Each body carries the geometry of its constituent parts; graph nodes denote rigid bodies, not individual geometric primitives or auxiliary software links. From the agent's SDK declarations, numerical tools construct the *kinematic constraints* $\mathcal{A}$ for all declared physical connections and explicit coordinate couplings. Thus, G records connectivity and connection attributes, while $\mathcal{A}$ expresses the conditions that valid configurations must satisfy. Dependencies between joint coordinates follow from these constraints, including those induced implicitly by loop closure.

Binding connections to geometry through sockets. Each physical connection edge $e = (a, b)$ binds two *body-local sockets*, S_{ae} and S_{be}, and specifies a connection type τ_e. A socket is an attachment frame fixed to its rigid body: the paired sockets locate the connection in the geometry, while τ_e determines the permitted relative motion. Fixed, revolute, prismatic, spherical, and helical connections impose different constraints on these frames. For example, the revolute closure in Figure 3(a) aligns the pin centers and axes while allowing rotation about the common axis. The off-tree screw–nut connection in Figure 3(b) belongs to E_L; it aligns the screw and nut axes and couples relative rotation to axial displacement according to the screw lead. Both its attachment and coupling constraints are included in $\mathcal{A}$.

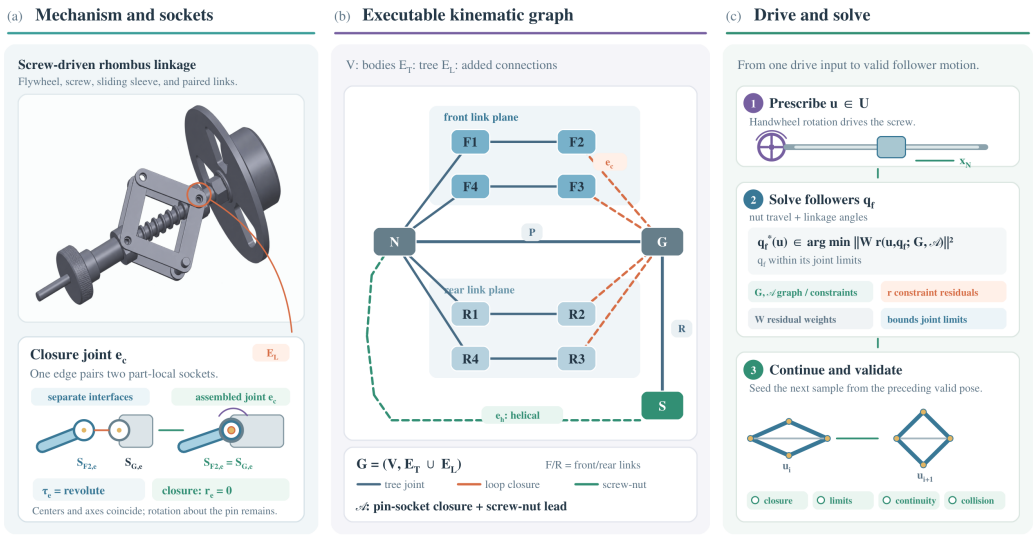

Figure 3: **Socket-bound loops and driven motion.** (a) Body-local sockets meet at a revolute closure joint. (b) Solid edges are tree joints; orange and green dashed edges denote loop closures and the screw–nut helical connection. (c) Flywheel rotation u determines nut travel and link angles, which are checked for closure, limits, continuity, and collision.

Sockets share construction parameters with geometric features such as holes and shafts. When changing the spacing between linkage layers, the agent can therefore update body positions, socket depths, and pin lengths together to improve clearance while preserving the connections. The representation identifies both the constraints that numerical tools check and the geometry and declarations that the agent can revise.

3.2 Motion Solving and Diagnostics from the Kinematic Graph

The Solver produces motion results and diagnostics for the agent. It uses the graph G, sockets, geometry, and kinematic constraints $\mathcal{A}$ obtained from the current program, together with the target motion domain $\mathcal{U}$ and prescribed paths. It proceeds in four steps.

Assembling the initial mechanism. The Solver seeks an initial configuration satisfying the declared kinematic constraints. The joint tree computes body poses from joint coordinates; off-tree connections constrain the relative poses of their socket frames. The initial solve must satisfy these connections and any explicit coordinate couplings together.

Solving for follower coordinates. For a driver input $u \in \mathcal{U}$, the Solver fixes the driver coordinates and solves for the remaining unknown follower coordinates q_f. In the screw-driven linkage, the screw–nut constraint relates handwheel rotation to nut travel, while loop closure constrains the connected links' rotations. Let $r(u, q_f; G, \mathcal{A})$ collect the residuals of constraints in $\mathcal{A}$ that are not already satisfied by the joint-tree parameterization or explicit coordinate substitution. Constraints enforced in this way are not counted again in the residual. The Solver computes

$$q_f^\star(u) \in \arg \min_{q_f \in [\underline{q}_f, \overline{q}_f]} \|W\, r(u, q_f; G, \mathcal{A})\|_2^2, \tag{2}$$

where $[\underline{q}_f, \overline{q}_f]$ contains the limits of the unknown follower coordinates and W normalizes the residual components. Driver values are sampled along the prescribed paths in $\mathcal{U}$; each solved configuration initializes the next sample to support continuation along a motion branch.

Checking motion validity. At each solved configuration, the Solver checks whether the kinematic constraints hold within tolerance, joint coordinates satisfy their limits, successive configurations are

continuous, and prohibited geometric interference is absent. A candidate returned by the optimizer is accepted only after these checks. Validation applies to the tested motion samples.

Associating diagnostics with the mechanism. The Solver returns motion results and diagnostics linked to the current program state, driver values, and named bodies, joints, or sockets. Examples include a joint reaching its limit, a closure residual exceeding its tolerance, or two bodies colliding. These measurements identify where and when a failure occurs, providing evidence for the agent's subsequent diagnosis.

3.3 Graph-Guided Evaluation and Reflective Correction

Existing 3D agents refine generated assets through visual comparison, programmatic checks, or simulation feedback (Le et al., 2025; Zhou et al., 2026; Wang et al., 2026b; Lin et al., 2026). In a closed-chain mechanism, however, a failure can involve several interdependent connections, so an observed discrepancy alone does not determine what should change. MARC uses the kinematic graph G, constraints $\mathcal{A}$, socket frames, and Solver diagnostics throughout evaluation and correction. They guide the agent in locating issues, investigating possible causes, and deciding which connections or parameters to revise while preserving the required motion.

Evaluate: identifying issues in the graph and geometry. Evaluate compares rendered views and motion results with the reference requirements. It uses G and $\mathcal{A}$ to inspect physical connections and the constraints relating driver and follower coordinates. Solver diagnostics identify the configurations and joints, sockets, or bodies involved in each execution failure. This assessment covers both missing or incorrect parts and connections required by the reference, and violations of the constraints actually declared in the program.

Judge: probing constraints and planning repairs. Within the Reflect stage, Judge uses the implicated graph edges and constraints to select geometric or kinematic probes at the reported failure configurations. For example, if closure fails when a follower reaches its limit, Judge can inspect the associated socket alignment and closure residuals, then temporarily relax the limit in a probe to test whether it restricts closure. Combining the probe results with the reference, Judge produces a repair plan specifying the parts, connections, or socket parameters to modify and the required connections and coordinate couplings to preserve. The graph therefore guides both probe selection and the scope of the repair; an issue at one connection is assessed in relation to the rest of the mechanism.

Executor and Reviewer: implementing and checking edits. Executor applies the plan, and Reviewer checks whether the requested changes were made and the required connections and coordinate couplings were preserved. The revised program returns to the Solver, which reconstructs its constraints and tests the complete mechanism along the original paths in $\mathcal{U}$. Evaluate then checks the new results against the reference. This repetition exposes both local improvements and failures elsewhere in the mechanism. Unresolved issues guide subsequent correction within the budget B_{corr}.

4 Experiments

Our evaluation addresses three questions: whether MARC improves the generation of complex coupled mechanisms relative to existing methods; whether its benefits transfer across foundation models; and which components contribute to structural and motion correctness.

4.1 Experimental Setup

Test set and baselines. We evaluate on a 20-object test set of complex articulated mechanisms spanning scissor and parallelogram linkages, multi-link closed chains, rotary–sliding couplings, gear transmissions, and folding structures. Each object has reference images and a standardized task description specifying the driver and intended motion. Human annotations identify critical part roles, required connections, and expected follower responses. Outputs are judged against these requirements rather than an exact reference part decomposition. We compare MARC with six methods: PhysX-Anything, URDF-Anything+, ArtLLM, Instruct-Particulate, Procedura, and Articraft (Cao et al.,

2026; Wu et al., 2026; Wang et al., 2026a; Li et al., 2026b; Lin et al., 2026; Zhou et al., 2026). Each method generates and executes objects in its native representation. We provide matched reference information where interfaces permit and record required input or format adaptations. Articraft, the closest agent-based baseline, uses the same backbone model as MARC in the paired comparison. A common evaluation procedure scores all outputs independently of generation; MARC's internal evaluation and reflection do not determine its final scores.

Evaluation metrics. We assess mechanism function with five metrics. **Executable rate (ER)** is the fraction of outputs that load and expose a controllable motion interface. **Structural correctness (SC)** is the per-object fraction of annotated part-role and connection requirements satisfied. **Valid motion coverage (VMC)** is the fraction of samples reached along a continuous branch satisfying required connections, joint limits, and collision criteria on each prescribed driver path; scores are averaged first over paths within an object and then equally over objects. **Coupled-motion correctness (CMC)** scores follower participation, direction, phase, and travel against the annotations. **Overall success rate (OSR)** requires all critical structural requirements and completion of the prescribed sampled motion paths. Missing outputs and unreachable samples fail; a stationary output cannot pass a motion task. For the same 20 objects, we additionally compare MARC and Articraft against manually constructed and annotated reference models using six agreement metrics: static 3D part-box gIoU and mIoU and part-surface symmetric mean-squared Chamfer distance (PC), plus gIoU, PC, and whole-object symmetric mean-squared Chamfer distance (OC) at approximately matched driver inputs. Part correspondences follow functional roles and connections and remain fixed across states; missing critical parts count as errors. Agreement scores are aggregated within each object and averaged equally across objects. The two metric groups are reported separately.

4.2 Comparison with Existing Methods

Table 1: Mechanism metrics on 20 objects (%; 95% intervals in Appendix C.5.2).

Method	ER	SC	VMC	CMC	OSR
PhysX-Anything	25.0	3.5	0.0	0.0	0.0
URDF-Anything+	70.0	5.2	0.0	0.0	0.0
ArtLLM	80.0	27.3	0.0	0.0	0.0
Instruct-Particulate	55.0	38.0	0.0	0.0	0.0
Procedura	90.0	70.5	6.4	10.5	0.0
Articraft (Kimi-K3)	100.0	62.5	0.0	24.6	0.0
MARC (Kimi-K3)	**100.0**	**100.0**	**67.8**	**98.2**	**60.0**

Table 1 shows that the first four baselines produce executable outputs for some objects, but satisfy relatively few critical part-role and connection requirements; none completes the prescribed mechanism tasks. Procedura and Articraft achieve higher executability or structural correctness, yet their valid motion coverage remains limited. In the direct comparison under the same backbone model, Articraft and MARC both achieve an executable rate of 100%, while their valid motion coverage differs substantially: 0% versus 67.8%. Articraft's coupled-motion correctness of 24.6% indicates that some follower responses match the annotations, but these local responses do not satisfy the validity criteria for required connections, joint limits, and collisions. In the dual-head lamp example, the lamp heads respond to the input while the crossing pivots separate during motion.

MARC achieves 100% structural correctness and 98.2% coupled-motion correctness, whereas its valid motion coverage and overall success rate are 67.8% and 60.0%, respectively. This gap shows that satisfying structural requirements and producing the intended follower response does not guarantee a valid trajectory: closure errors and geometric interference can still cause failure. For example, the folding desk encounters interference before completing its prescribed travel, and the excavator fails the bucket-closure check. These cases identify motion validity and geometric clearance as remaining limitations.

Across the six aggregate metrics in Table 2, MARC's part geometry and motion poses agree more closely with the references than Articraft's. The trend does not hold for every object and metric. For the gear-train example, Articraft has higher static gIoU and mIoU than MARC, yet its output does

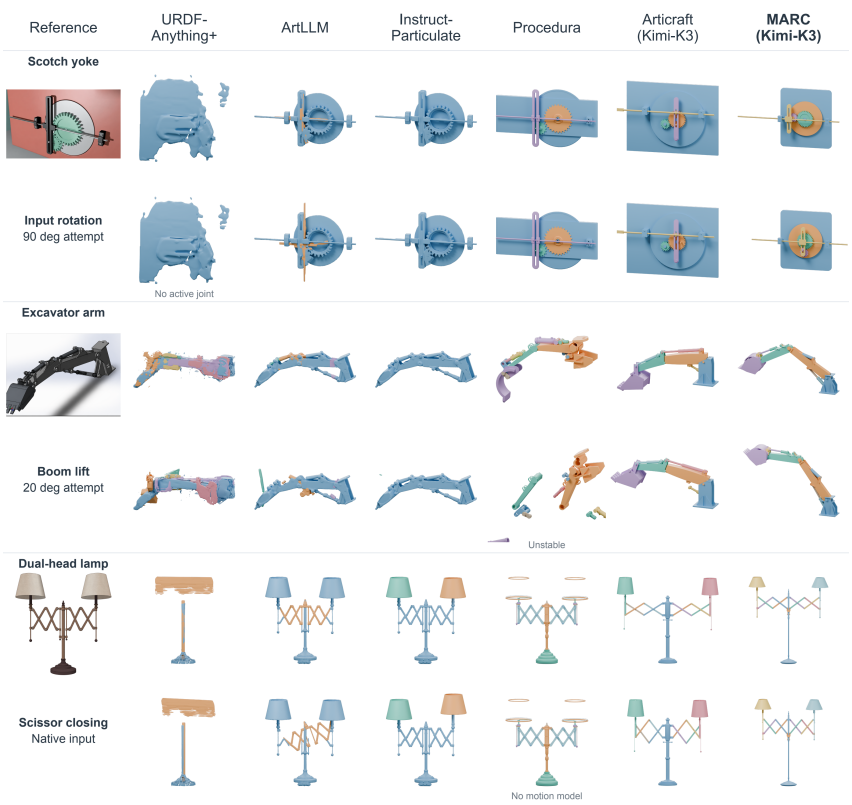

Figure 4: Native-control responses to input rotation, boom lifting, and scissor contraction (initial state above, driven state below). Camera, scale, and part colors are fixed within each pair. Stationary and unstable responses are retained; the MARC excavator still collides.

Table 2: Geometric agreement with manually constructed and annotated reference models for each object. Objects are equally weighted; motion inputs are approximately matched.

	Static			Motion (approximate)		
Method	gIoU ↑	PC ↓	mIoU ↑	gIoU ↑	PC ↓	OC ↓
Articraft (Kimi-K3)	−0.0479	0.0726	0.2316	−0.0497	0.0743	0.0103
MARC (Kimi-K3)	**0.1402**	**0.0100**	**0.3046**	**0.0897**	**0.0130**	**0.0053**

not satisfy the critical transmission requirement. Geometric overlap therefore measures agreement in shape and placement but cannot establish correct connections or task motion.

4.3 Ablation Analysis

We first compare direct API generation with the complete MARC harness under the same backbone model. Direct API generates a program from the task inputs without MARC's staged execution diagnostics and reflective correction; full MARC runs the complete workflow independently from the same reference inputs. The Kimi-K3 comparison covers 20 objects, and the GPT-6 Astra comparison covers 19. Within each backbone, both configurations are evaluated on the same objects.

The main difference lies in valid motion. With Kimi-K3, Direct API achieves zero valid motion coverage and overall success, whereas full MARC reaches 67.8% and 60.0%, respectively. With GPT-6 Astra, Direct API already approaches perfect structural and coupled-motion correctness, but its overall success rate is 31.6%; full MARC succeeds on all 19 tested objects. Thus, even when a backbone produces largely correct part relationships and follower responses, its output may fail to complete the prescribed sampled motion paths while satisfying connection and collision requirements.

Table 3: Backbone and harness comparison (%; 95% intervals in Appendix C.5.2).

Backbone	Configuration	ER	SC	VMC	CMC	OSR
Kimi-K3	Direct API	80.0	54.0	0.0	61.4	0.0
	MARC	100.0	100.0	67.8	98.2	60.0
GPT-6 Astra	Direct API	100.0	97.2	33.5	98.1	31.6
	MARC	100.0	100.0	100.0	100.0	100.0

Table 4: Ablation results under independent evaluation. The no-graph configuration is regenerated using the execution setup described in Appendix A.7.

Configuration	SC	VMC	CMC	OSR
Full MARC	100.0	67.8	98.2	60.0
Without reflection	100.0	54.1	98.2	40.0
Without graph-based mechanism support	73.5	34.4	45.8	30.0
Without graph-guided repair	85.7	56.3	82.9	45.0

This is a system-level comparison between direct generation and the complete harness, which also includes execution feedback, tool calls, and revision. It does not attribute the gain to any single component.

We evaluate three ablation configurations. **Without reflection**, we evaluate the initial Solver candidate without subsequent correction. **Without graph-based mechanism support**, we regenerate each object using the no-graph configuration described in Appendix A.7, removing the representation together with its associated execution and authoring support. **Without graph guidance during reflection**, we retain the kinematic graph and its numerical execution but withhold graph-based connection localization and constraint associations from the correction process. These configurations examine reflection, graph-based mechanism support, and graph-guided repair, respectively.

Removing reflection leaves structural and coupled-motion correctness unchanged, but reduces valid motion coverage and overall success. On this benchmark, the principal benefit of correction is therefore improved motion validity. Removing graph-based mechanism support causes a larger decline, particularly in coupled-motion correctness, which falls from 98.2% to 45.8%. This comparison measures the combined contribution of the representation and its associated execution and authoring support, rather than the graph representation alone. When the graph remains available for execution but its guidance is removed from reflection, all reported metrics also fall below those of the full method, supporting the benefit of graph-based connection and constraint information during correction.

The analysis tracks both repaired failures and regressions of initially correct properties. Every post-repair candidate receives a final independent structural and motion assessment; passing a configuration's internal execution checks alone is insufficient for semantic acceptance.

5 Conclusion

We presented MARC, an agent harness built around an executable kinematic graph for generating complex articulated objects. The agent declares geometry, connections, and coordinate couplings; numerical tools construct kinematic constraints and solve for motion. The graph and execution diagnostics guide evaluation and reflective correction, and each revised program is tested along the prescribed paths in the target motion domain. Repeated probes and solves incur computational overhead; future work will explore adaptive probe selection and result reuse.

AI Use Statement

We used ChatGPT to assist with drafting selected passages, translating portions of the manuscript into English, editing text for clarity and readability, and preparing and revising some figures. The authors reviewed and revised the AI-assisted material and take responsibility for the final manuscript, including its technical claims and reported results.

REFERENCES

Moritz Bächer, Stelian Coros, and Bernhard Thomaszewski. LinkEdit: Interactive linkage editing using symbolic kinematics. *ACM Transactions on Graphics*, 34(4):99:1–99:8, 2015. doi: 10.1145/2766985. URL https://la.disneyresearch.com/publication/linkedit/.

Ziang Cao, Fangzhou Hong, Zhaoxi Chen, Liang Pan, and Ziwei Liu. PhysX-Anything: Simulation-ready physical 3D assets from single image. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2026. URL https://arxiv.org/abs/2511.13648.

Qiuyu Chen, Aaron Walsman, Marius Memmel, Kaichun Mo, Alex Fang, Dieter Fox, and Abhishek Gupta. URDFormer: A pipeline for constructing articulated simulation environments from real-world images. In *Proceedings of Robotics: Science and Systems*, 2024. doi: 10.15607/RSS.2024.XX.124. URL https://www.roboticsproceedings.org/rss20/p124.html.

Stelian Coros, Bernhard Thomaszewski, Gioacchino Noris, Shinjiro Sueda, Moira Forberg, Robert W. Sumner, Wojciech Matusik, and Bernd Bickel. Computational design of mechanical characters. *ACM Transactions on Graphics*, 32(4):83:1–83:12, 2013. doi: 10.1145/2461912.2461953.

Ioannis Fudos and Christoph M. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16(2):179–216, 1997. doi: 10.1145/248210.248223.

Yipeng Gao, Lei Shu, Genzhi Ye, Xi Xiong, Ameesh Makadia, Meiqi Guo, Laurent Itti, and Jindong Chen. 3DCodeBench: Benchmarking agentic procedural 3D modeling via code. *arXiv preprint arXiv:2606.01057*, 2026. URL https://arxiv.org/abs/2606.01057.

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*, 2025. URL https://arxiv.org/abs/2410.13882.

Quentin Leboutet, Nina Wiedemann, Zhipeng Cai, Michael Paulitsch, and Kai Yuan. MIDGArD: Modular interpretable diffusion over graphs for articulated designs. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/0318de478e18308a5f64297f618299d3-Abstract-Conference.html.

Jiahui Lei, Congyue Deng, William B. Shen, Leonidas Guibas, and Kostas Daniilidis. NAP: Neural 3D articulated object prior. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/655846cc914cb7ff977a1ada40866441-Abstract-Conference.html.

Ruining Li, Yuxin Yao, Chuanxia Zheng, Christian Rupprecht, Joan Lasenby, Shangzhe Wu, and Andrea Vedaldi. Particulate: Feed-forward 3D object articulation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 27708–27718, 2026a. URL https://openaccess.thecvf.com/content/CVPR2026/html/Li_Particulate_Feed-Forward_3D_Object_Articulation_CVPR_2026_paper.html.

Ruining Li, Yuxin Yao, Matt Zhou, Chuanxia Zheng, Christian Rupprecht, Joan Lasenby, Shangzhe Wu, and Andrea Vedaldi. Instruct-Particulate: Scaling feed-forward 3D object articulation with kinematic control. *arXiv preprint arXiv:2606.14699*, 2026b. URL https://arxiv.org/abs/2606.14699.

Youtian Lin, Yikang Yang, Zhanpeng Hu, Mengqi Zhou, Feihu Zhang, Xun Cao, Jiaheng Liu, and Yao Yao. Procedura: Agentic 3D modeling with procedural control. *arXiv preprint arXiv:2608.26238*, 2026. URL https://arxiv.org/abs/2608.26238.

Jiayi Liu, Hou In Ivan Tam, Ali Mahdavi-Amiri, and Manolis Savva. CAGE: Controllable articulation GEneration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 17880–17889, 2024. URL https://arxiv.org/abs/2312.09570.

Jiayi Liu, Denys Iliash, Angel X. Chang, Manolis Savva, and Ali Mahdavi-Amiri. SINGAPO: Single image controlled generation of articulated parts in objects. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.16499.

Kevin M. Lynch and Frank C. Park. *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017. doi: 10.1017/9781316661239.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html.

Zhao Mandi, Yijia Weng, Dominik Bauer, and Shuran Song. Real2Code: Reconstruct articulated objects via code generation. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2406.08474.

Jiayi Su, Youhe Feng, Zheng Li, Jinhua Song, Yangfan He, Botao Ren, and Botian Xu. ArtFormer: Controllable generation of diverse 3D articulated objects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1894–1904, 2025. URL https://arxiv.org/abs/2412.07237.

Bernhard Thomaszewski, Stelian Coros, Damien Gauge, Vittorio Megaro, Eitan Grinspun, and Markus Gross. Computational design of linkage-based characters. *ACM Transactions on Graphics*, 33(4):64:1–64:9, 2014. doi: 10.1145/2601097.2601143.

Penghao Wang, Siyuan Xie, Hongyu Yan, Xianghui Yang, Jingwei Huang, Chunchao Guo, and Jiayuan Gu. ArtLLM: Generating articulated assets via 3D LLM. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 34281–34291, 2026a. URL https://arxiv.org/abs/2603.01142.

Rui-Huan Wang, Si-Tong Wei, Jia-Qi He, Heng-Yi Wei, Baoquan Chen, and Peng-Shuai Wang. aDSL: Agentic 3D creation via joint agent-program design. *arXiv preprint arXiv:2608.17975*, 2026b. URL https://arxiv.org/abs/2608.17975.

Zhuangzhe Wu, Yue Xin, Chengkai Hou, Minghao Chen, Yaoxu Lyu, Jieyu Zhang, and Shanghang Zhang. URDF-Anything+: End-to-end generation for simulation-ready articulated assets. *arXiv preprint arXiv:2603.14010*, 2026. URL https://arxiv.org/abs/2603.14010.

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*, pp. 11097–11107, 2020. URL https://arxiv.org/abs/2003.08515.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-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.

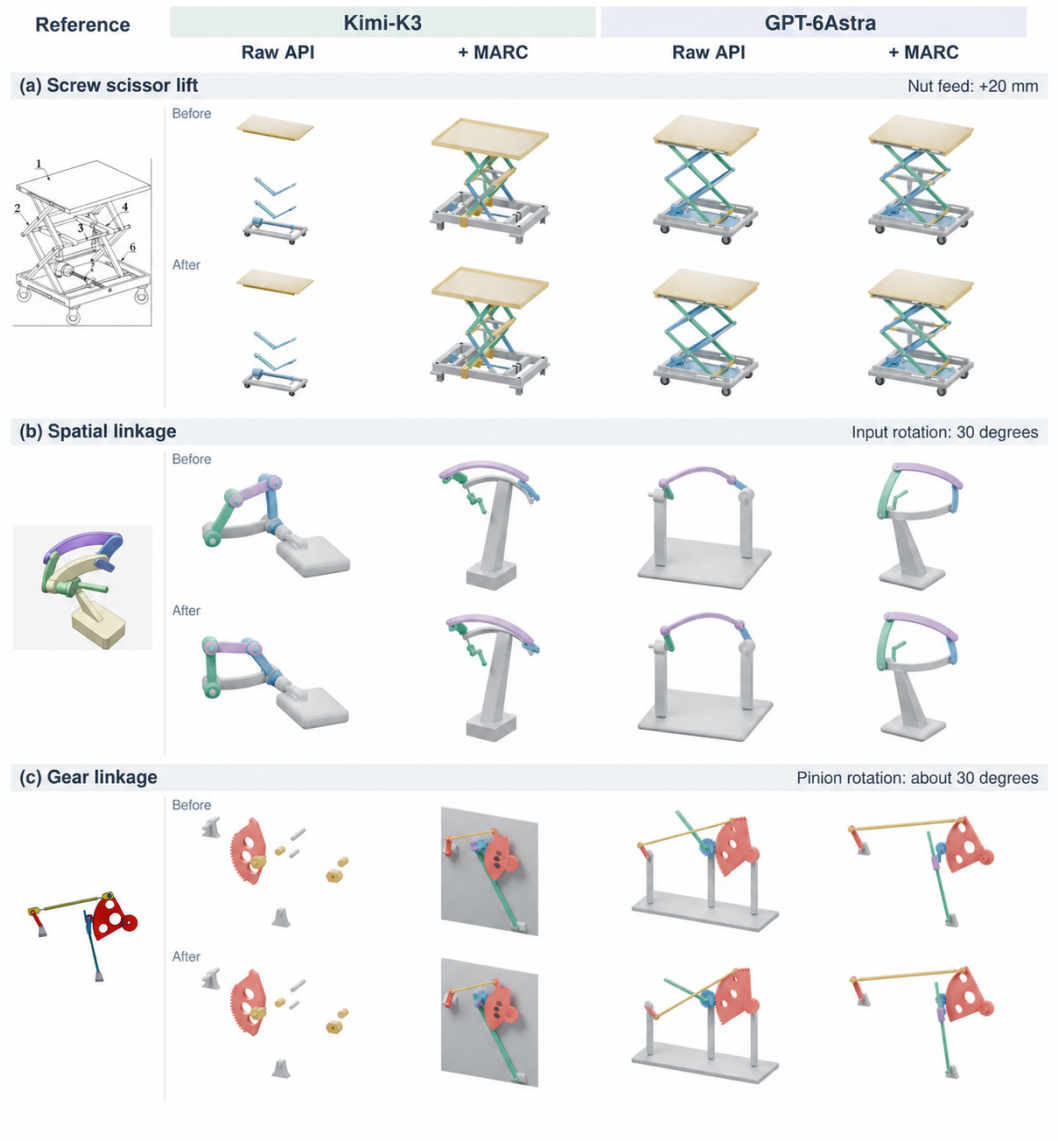

Figure 5: **Ablation 1: direct API generation and the full MARC harness across backbone models.** Within each backbone, direct API generation and the complete MARC harness use the same reference inputs. The comparison evaluates the system-level effect of the full harness, including tool calls, execution feedback, and reflective correction, rather than isolating a single component. The visualization illustrates the generated results; structural and motion metrics are computed according to the evaluation protocol in this appendix.

A Implementation Details

This appendix supplements the program interfaces, numerical execution, evidence organization, and reflective correction workflow of MARC. Following the notation in the main text, the inputs comprise reference images I, an object and motion description x, and a target motion domain $\mathcal{U}$ with prescribed test paths. The output is an executable articulated-object program P. An executable kinematic graph associates geometry, physical connections, motion constraints, and repair operations throughout the workflow.

Table 5: Responsibilities and outputs of the workflow stages.

Stage or role	Main responsibility	Output
Topology	Write a mechanism program from the reference inputs using SDK documentation, construction interfaces, and examples.	Geometry, connection, and coupling declarations.
Solver	Assemble the mechanism, compute follower motion, and check constraints.	Execution state, configurations, and numerical diagnostics.
Evaluate	Compare structure, motion, and geometry with the reference requirements.	Findings associated with mechanism entities.
Judge	Investigate causes using targeted probes and plan corrections.	A structured repair plan.
Executor	Implement the plan and provide execution evidence.	A revised candidate program.
Reviewer	Check implementation of the plan and preservation of required relationships.	A review decision and remaining items.

A.1 Programming Environment and Workflow

MARC is implemented in Python and extends Articraft's programmatic authoring interface with mechanism construction and diagnostic capabilities. Geometry is constructed using CadQuery, while numerical computation uses NumPy and SciPy. Lengths and angles are expressed in meters and radians, respectively. A generated program declares geometry, rigid-body membership, local attachment frames, joints, loop-closing connections, and coordinate couplings. The numerical backend constructs and executes the mechanism from these declarations.

The workflow comprises Topology, Solver, Evaluate, and Reflect. Reflect contains three roles: Judge, Executor, and Reviewer. Language-model roles have distinct task instructions, tool interfaces, and contexts; Solver performs numerical assembly and motion computation.

The Kimi-K3 configuration uses the high reasoning setting. Role specialization is implemented through inference-time context and tool access, without separately training a model for each role.

A.2 Kinematic Graph and Geometry Interfaces

The executable kinematic graph is

$$G = (V, E_T \cup E_L), \tag{3}$$

where V denotes rigid bodies, E_T is the joint tree used for forward kinematics, and E_L contains additional physical connections that close mechanism loops. A rigid body may contain multiple geometric components and participate in multiple loops through different connections. Geometric primitives and auxiliary coordinate carriers used to represent compound joints are not counted as separate physical bodies.

A physical connection $e = (a, b)$ binds two body-local sockets, S_{ae} and S_{be}, and specifies a connection type τ_e. Given body transforms $T_a(q)$ and $T_b(q)$, the corresponding attachment frames in world coordinates are

$$T_a(q)S_{ae}, \qquad T_b(q)S_{be}. \tag{4}$$

The numerical backend constrains their relative position and orientation according to the connection type.

For a helical connection, axial displacement and relative rotation satisfy

$$d = d_0 + h\theta, \tag{5}$$

where h is the signed axial travel per radian. Physical connection constraints and explicit coordinate couplings jointly form the constraint set $\mathcal{A}$ defined in the main text. Loop closure can additionally determine nonlinear follower relationships implicitly through geometry.

Geometric features and sockets share construction parameters. For example, named points on a rigid member determine both hole locations and attachment positions. Layer parameters jointly determine plate thicknesses, clearances, socket depths, and pin spans. The SDK provides composable interfaces

Table 6: Physical connections and coordinate couplings supported by the mechanism interface.

Relationship	Constraints	Remaining relative motion
Fixed connection	Position and orientation agreement.	None.
Revolute connection	Coincident attachment centers and aligned axes.	Rotation about the common axis.
Prismatic connection	Transverse position and orientation agreement.	Translation along the guide axis.
Spherical connection	Coincident attachment centers.	Relative rotation.
Helical connection	Aligned screw axes and coupled axial displacement and rotation.	Coupled translation and rotation.
Affine coordinate coupling	$q_b = \alpha q_a + \beta$.	The dependent coordinate is determined by the driving coordinate.

Table 7: Supplementary SDK guidance for common mechanical connections.

Connection	Geometry and interface guidance	Motion relationship
Ball joint	Construct the ball and receiving structure, place both sockets at the ball center, and provide rotational clearance.	Constrain the centers to coincide while allowing relative rotation.
Screw–nut connection	Declare the screw, nut, coaxial interfaces, and supporting or guiding geometry; use consistent lead, axis, and rotation conventions.	Relate axial displacement to relative rotation and solve jointly with the remaining mechanism constraints.
Bolted fastening and pin joints	Coordinate holes, shaft diameters and lengths, plate thicknesses, and layer clearances; specify rigid-body membership of the fastener.	Maintain a fixed relative pose for fastening, or explicitly declare a revolute connection when the shaft acts as a pivot.
Revolute and sliding connections	Align the rotation or guide axes, declare attachment positions and motion ranges, and construct the associated mounting or guiding geometry.	Retain rotation about the axis or translation along it.

for rigid members, planar construction, pin stacks, scissor cells, telescopic actuators, and affine drives, while retaining low-level geometry and connection declarations. These interfaces allow coordinated edits to geometry and attachment parameters.

SDK guidance for common joints and connections. Beyond the basic authoring interface, we supplement the SDK with implementation guidance and construction examples for common joints and connection structures. The guidance explains how to declare solid geometry, local attachment frames, and motion constraints together.

For a screw with signed lead p, the axial travel per radian is $h = p/(2\pi)$, with the sign determined by the axis and rotation conventions. Bolt geometry alone does not determine the connection's degrees of freedom: clamping corresponds to a fixed relationship, whereas using the shaft as a pivot requires a revolute declaration.

These instructions and examples are supplied as SDK context to help the model express common mechanical connections as executable geometry and constraints. The model determines the part composition, attachment locations, and parameters of each object from its reference inputs.

A.3 Assembly and Follower-Motion Solving

Solver constructs the joint tree, loop-closing connections, and constraint set from the current program. It first seeks an initial assembly satisfying the declared connections. For a prescribed driver input u, the driver coordinates remain fixed, and the unknown follower coordinates are solved within

Table 8: Principal numerical settings of the implementation.

Parameter	Setting
Attachment-position acceptance tolerance	10^{-4} m
Attachment-axis acceptance tolerance	10^{-3} rad
Default configuration snapshots for the assembly objective	5
Default surface samples per part	192
Maximum local–global assembly iterations	30
Langevin search-step budget	300
Monte Carlo sample budget	30
Numerical random seed	0
Initial samples for adaptive closed-loop scanning	5
Maximum samples for adaptive closed-loop scanning	33
Maximum adaptive refinement depth	4

their limits. Constraints already satisfied by the joint-tree parameterization or explicit coordinate substitution are not duplicated in the residual.

Follower solving uses the bounded residual objective from the main text:

$$q_f^\star(u) \in \arg\min_{q_f \in [\underline{q}_f, \overline{q}_f]} \|W\, r(u, q_f; G, \mathcal{A})\|_2^2 . \tag{6}$$

The implementation uses nonlinear least squares with separate scaling for positional and directional residuals. Along a prescribed input path, nearby solved configurations initialize subsequent samples to support continuation on a motion branch. Assembly optimization uses rigid transformations; part dimensions are controlled by the program's geometry parameters.

Candidate configurations are checked for attachment errors, joint limits, branch continuity, and geometric interference. Closed-loop motion checking supports adaptive refinement, adding driver samples where follower changes are large, residuals increase, or near-singular behavior is detected. The sampled inputs and check results are retained for each tested path.

Assembly-objective snapshots and motion-validation samples control different computational stages. Motion validity is assessed at the configurations actually tested along the prescribed paths, combining connection, limit, continuity, and collision evidence.

A.4 Graph-Associated Evidence and Layered Evaluation

Solver associates diagnostics with the current program state, driver inputs, and named bodies, joints, or sockets. Each finding records where a problem occurs, the tested configuration, and the relevant measurement, such as a loop-closure residual, an active joint limit, or interference between two bodies. Program versions and model hashes accompany the evidence so that subsequent repairs refer to the corresponding executed candidate.

Evaluate combines reference images, task descriptions, the kinematic graph, constraint information, multiview renderings, and numerical diagnostics. Findings are organized into three levels:

- **L1: structure and topology.** Check whether critical parts, rigid-body grouping, physical connections, and required transmission relationships agree with the reference.
- **L2: function and motion.** Check driver and follower responses, loop preservation, limits, branch continuity, and interference during motion.
- **L3: geometry and detail.** Check differences in part shapes, holes, slots, supports, and local dimensions.

Visual inspection combines overall views with local views when needed, using spatial layering and depth information to help distinguish occlusion and contact regions. Motion judgments also use executed configurations and constraint measurements. Findings are associated with relevant graph nodes, connections, and constraints, providing Judge with evidence for selecting probes and repair scopes.

A.5 Structured Reflection and Coordinated Repair

Judge investigates Evaluate's findings by inspecting the implicated connections, constraints, and geometry and selecting targeted probes. For example, when closure failure coincides with a follower reaching its limit, Judge can inspect attachment-position and axis errors, then temporarily modify the limit in a local probe to test whether it restricts closure. Probe results and reference requirements jointly inform the repair decision.

A repair plan specifies the target parts or connections, supporting diagnosis, repair intent, permitted scope, dependencies, relationships to preserve, expected effects, and rechecking requirements. Available repairs cover topology, attachment parameters, motion parameters, and geometry. For interdependent changes, the plan identifies the entities that must be updated together. Increasing linkage-layer clearance, for example, may require coordinated changes to body positions, socket depths, and pin lengths to preserve the intended connection.

Executor implements the plan in a candidate program and records execution evidence. Reviewer compares the implementation with the plan and verifies preservation of required connections and couplings. The implementation allows at most 24 execution-agent turns and three implementation-review rounds. These internal limits are managed separately from the outer correction budget B_{corr}.

After implementation review, the candidate returns directly to Solver, which reconstructs its constraints and tests the complete mechanism along the original paths in the target motion domain. Evaluate then compares the new results with the reference requirements. Both local improvements and regressions elsewhere inform the next decision, and unresolved findings guide further correction within the remaining budget.

Coordinate-consistent diagnosis. Coordinate changes introduced during assembly are interpreted consistently during diagnosis. If a rest offset δ_j is absorbed into the joint origin without changing the axis convention, then

$$q_j^{\mathrm{before}} = \delta_j + q_j^{\mathrm{after}}. \tag{7}$$

The corresponding limits are transformed by the same offset:

$$[\underline{q}_j, \overline{q}_j]^{\mathrm{after}} = [\underline{q}_j^{\mathrm{before}} - \delta_j,\, \overline{q}_j^{\mathrm{before}} - \delta_j]. \tag{8}$$

This mapping distinguishes a change of coordinate zero from a change in physical motion range, keeping limit analysis and subsequent edits consistent with the assembled mechanism.

A.6 Execution Control and Result Recording

The complete workflow follows the main text:

1. **Generation.** Topology writes an initial mechanism program from $(I, x, \mathcal{U})$.
2. **Solving.** Solver assembles the mechanism, computes motion along the prescribed driver paths, and produces diagnostics.
3. **Evaluation.** Evaluate checks structure, function, and geometry against the reference requirements.
4. **Reflective correction.** If correction is needed and budget remains, Judge produces a plan, Executor implements it, and Reviewer checks the changes.
5. **Revalidation.** The revised program returns to Solver and subsequently to Evaluate. The process repeats for at most B_{corr} correction rounds.

Stages share the current program, graph, and corresponding execution evidence while preserving version relationships between candidates. Output records include source programs, repair plans, relevant probes, solver configurations, sampled driver inputs, and diagnostic results, allowing changes to be traced across revisions.

Internal Evaluate judgments guide generation and correction. The ER, SC, VMC, CMC, and OSR values reported in the paper are computed by the independent evaluation procedure specified in the main text; internal judgments do not directly determine the final scores.

Table 9: Records used to reproduce and verify an experimental result.

Record category	Purpose
Inputs and task	Identify reference images, text, driver domains, and test paths.
Implementation and configuration	Identify the code version, SDK, prompts, example resources, and model-call settings.
Programs and repairs	Associate the initial candidate, repair plans, code changes, and final candidate.
Execution and evaluation	Associate solver settings, sampled configurations, diagnostics, exported artifacts, and object-level scores.

A.7 Implementation of the No-Graph Ablation

The no-graph configuration removes the graph-based mechanism construction and execution support of the full method. This includes the executable kinematic graph, socket-bound connection and loop-closure declarations, the original graph-based assembly and Solver components, and the supplementary mechanism-construction interfaces and SDK guidance described above. This is therefore a joint ablation of the representation and its associated execution and authoring support.

The configuration regenerates programs from the task inputs. The model uses general-purpose geometry and numerical libraries to define rigid-body geometry and directly implement the mapping from driver inputs to world-space body poses. The program exposes two interfaces: `build()` returns each body's local geometry and appearance, and `motion(controls)` returns its world-space rigid transformation for the supplied inputs. Geometry remains fixed during motion, and the generated program performs the motion calculations itself.

The model context excludes the original graph-based construction documentation, mechanism-example retrieval, and supplementary guidance for connections such as ball joints, screw–nut mechanisms, and layered pin assemblies. The graph-based assembly, closed-loop follower solver, and graph-associated repair interfaces are also unavailable. Retained general-purpose tools execute the program, check rigid transformations, render multiple views, and record sampled configurations and geometric contacts.

During correction, the model uses the reference images, candidate code, renderings, and execution diagnostics to produce a free-form repair plan, revise the complete program, and execute it again. This preserves opportunities for execution-conditioned revision while removing localization and planning interfaces based on graph nodes, connections, and constraints.

This configuration regenerates its initial candidates and uses its own direct-motion execution interface. Its outputs are assessed using the independent structural and motion evaluation protocol in the main text. The comparison measures the combined contribution of the kinematic graph, its associated Solver and construction interfaces, and SDK mechanism guidance; it does not isolate the effect of graph notation alone.

B Reproducibility Details

This appendix describes how to restore an experimental configuration, rerun generation and correction, and verify the evaluation of saved artifacts. Method and numerical implementation details are given in Appendix A; metric definitions, motion checks, and baseline adapters are described in Appendix C. We distinguish replaying and rescoring saved artifacts from rerunning the complete generation workflow from reference inputs.

B.1 Reproduction Units and Version Correspondence

The unit of reproduction is an object-level experimental record associating reference inputs, run configurations, program versions, and results. A stable object identifier links the reference images, task description, driver domains, and evaluation requirements. File hashes distinguish the programs used for generation, correction, and evaluation, preventing code, diagnostics, or scores from different candidates from being combined.

A correction experiment starts from its specified initial candidate and execution state; a regeneration experiment starts from reference inputs. These routes are recorded separately so that replaying or correcting an existing candidate is not counted as a new generation run.

B.2 Environment and Model Configuration

The implementation supports Python 3.11–3.12 and uses CadQuery, NumPy, SciPy, Trimesh, and collision-query components. Reproduction should restore the software versions and numerical configuration associated with the target experiment. Appendix A specifies the principal settings of the main method. Worker counts and compute devices are recorded as execution-environment parameters for interpreting resource use and latency.

The Kimi-K3 main-method configuration uses the high reasoning setting. The model identifier, reasoning setting, prompts, tool descriptions, SDK documentation, and example resources jointly define the model-call configuration. Internal execution-agent and review limits are configured separately from the outer correction budget, using the settings associated with the target experiment.

Numerical replay and external scoring of saved artifacts do not require new calls to the generation model. Regenerating candidates requires access to the corresponding model service. Numerical seeds control the numerical components; new calls to a remote generation model receive separate run identifiers, and their outputs are evaluated anew.

B.3 Two Reproduction Routes

Replay of saved artifacts. Load the specified program with its native connections, loops, couplings, and limits. Restore its driver inputs and test paths, execute the motion, and regenerate checking results. Apply the independent evaluation protocol to assess structure, valid motion, and follower responses, then recompute aggregate metrics from object-level results. Replay keeps the candidate unchanged and uses its program hash to verify correspondence with the reported artifact.

Rerunning the complete workflow. Start Topology from the specified reference inputs, execute Solver and Evaluate, and run Judge, Executor, and Reviewer within the corresponding budget. Each revised candidate returns to numerical execution and evaluation. The final candidate is scored by the independent evaluation procedure; internal model judgments do not replace the reported metrics.

For programs generated through the graph-based mechanism interface, replay preserves loop-closure and coordinate-coupling declarations. An execution that retains only the joint tree while discarding these relationships is not treated as an equivalent replay of the original mechanism.

B.4 Starting Points for Comparisons and Ablations

The configuration without reflection evaluates its corresponding initial Solver candidate without subsequent correction. Comparisons between correction mechanisms use the initial candidates specified by the respective experiment and retain the correspondence between inputs and candidates.

The no-graph configuration follows Appendix A.7: it regenerates programs from reference inputs, uses a direct-motion execution interface, and removes the original graph-based Solver and associated SDK guidance. Its reproduction does not load the full method's graph-based execution or instructional resources. Outputs from different configurations are assessed using the independent evaluation protocol, without requiring identical internal representations or execution backends.

Baselines use their native outputs and execution adapters, as detailed in Appendix C. Reproduction adapters load and execute existing declarations; they do not add missing connections or modify candidate geometry.

B.5 Verification and Aggregation

Verification establishes the object and program version, driver inputs and sampling configuration, execution results, and object-level scores. Programs that cannot be loaded, missing outputs, and unreachable prescribed motion samples contribute to the corresponding failures under the evaluation

protocol. Reruns and retries retain separate identifiers and provenance rather than silently overwriting the original candidate.

Reported aggregate metrics are recomputed from object-level evaluation results. Compilation, implementation review, and internal Solver checks are recorded as stage-specific outcomes, while final task conclusions follow the independent structural and motion assessment. This separates verification of generation outputs, correction effects, and the artifacts underlying the reported scores.

C Reference Ground Truth and Evaluation Protocol

C.1 Manually Constructed Reference Ground Truth

We manually construct and annotate reference models for 20 articulated objects. Starting from procedural modeling code, we inspect and correct these models against the input images, covering part geometry, rigid-body decomposition, connection locations, joint types, transmission relationships, and spatial relationships during motion. The reference ground truth includes both geometric models and functional requirements for assessing structural and motion correctness.

The reference information for each object covers three aspects. First, it specifies the functional roles and relative arrangement of required parts, rigid and articulated connections, and loop-closing relationships. Second, it identifies independent drivers, prescribed motion ranges, follower directions, phase relationships, and required travel. Third, it specifies the connections that must remain intact during motion, applicable joint limits, and regions of permitted contact or prohibited penetration.

Correspondences between generated and reference parts are established through functional roles and connection relationships. Identical part names or mesh decompositions are not required. For example, multiple submeshes rigidly attached to one another may correspond to a single reference part. Such correspondences must not conceal missing articulated connections or incorrect degrees of freedom. For details that cannot be uniquely determined from the input images, evaluation is restricted to requirements that can be judged reliably, while allowing reasonable alternative realizations with equivalent functionality.

C.2 Evaluation Units and Common Conventions

The reference collection contains $N = 20$ objects. For configuration m, let $\mathcal{D}_m$ denote its evaluated object set and let $n_m = |\mathcal{D}_m|$. Within each backbone, the direct API and full-harness configurations use the same object set. The Kimi comparison covers 20 objects, whereas the GPT-6 Astra comparison covers 19. Each metric uses the actual evaluation set of the corresponding configuration rather than assigning a denominator of 20 to every configuration.

For object i, let A_i be the set of structural requirements and R_i the set of coupled-motion requirements. Each requirement specifies the property being assessed and its acceptance condition. Requirements essential to the basic functionality of the mechanism are designated as critical. The following equations omit the configuration index and operate on one configuration's evaluation set $\mathcal{D}$, with $n = |\mathcal{D}|$.

Motion evaluation starts from a prescribed initial state and proceeds sequentially along specified driver paths. Sample locations are determined by the requested motion domain; the domain is not reduced retrospectively to match the range that a generated model can complete. Separate test paths are recorded for independent drivers and for forward and reverse motion.

Generated results are loaded through their corresponding execution interfaces while retaining their original connections, coordinate couplings, and limits. Execution adapters do not add missing parts or repair mechanism relationships. Successful loading, local joint motion, validity over a complete trajectory, and semantic mechanism correctness constitute distinct levels of evidence. All primary metrics are aggregated within each object before macro-averaging across objects.

C.3 Primary Evaluation Metrics

C.3.1 Executable Rate (ER)

Executable Rate measures whether a generated result can be loaded and exposes a usable motion-control interface. Let $e_i \in \{0, 1\}$ indicate whether object i meets this condition. Then

$$\mathrm{ER} = \frac{1}{n} \sum_{i \in \mathcal{D}} e_i \times 100\%. \tag{9}$$

A static mesh or rendered image alone does not satisfy this condition. ER does not establish correct part structure, transmission relationships, or motion behavior; these properties are assessed by the following metrics.

C.3.2 Structural Correctness (SC)

Structural Correctness measures the fraction of reference structural requirements satisfied by a generated result. For requirement $a \in A_i$, define

$$s_{ia} = \mathbf{1}\left[a \text{ is satisfied}\right], \tag{10}$$

where $\mathbf{1}[\cdot]$ is the indicator function. The object-level and aggregate scores are

$$\mathrm{SC}_i = \frac{1}{|A_i|} \sum_{a \in A_i} s_{ia}, \tag{11}$$

$$\mathrm{SC} = \frac{1}{n} \sum_{i \in \mathcal{D}} \mathrm{SC}_i \times 100\%. \tag{12}$$

Structural requirements assess the presence of necessary parts, their functional roles, the identities of connected bodies, joint types, attachment locations, and loop-closing relationships. A visually similar model with incorrect connections does not satisfy the corresponding requirements. Missing required parts or connections cause their associated requirements to fail.

C.3.3 Valid Motion Coverage (VMC)

Valid Motion Coverage measures the fraction of prescribed motion samples that can be reached from the initial state along a valid motion branch. Suppose object i has M_i test paths, with K_{ip} samples on path p. We set $v_{ipk} = 1$ only when sample k is reached from the prescribed initial state along that path and the traversal satisfies all applicable critical connection, loop-closure, joint-limit, and collision conditions; otherwise, $v_{ipk} = 0$.

The object-level score is

$$\mathrm{VMC}_i = \frac{1}{M_i} \sum_{p=1}^{M_i} \left(\frac{1}{K_{ip}} \sum_{k=1}^{K_{ip}} v_{ipk} \right), \tag{13}$$

and the aggregate score is

$$\mathrm{VMC} = \frac{1}{n} \sum_{i \in \mathcal{D}} \mathrm{VMC}_i \times 100\%. \tag{14}$$

After a blockage, connection separation, limit violation, or prohibited collision, jumping to a later pose does not extend the valid coverage of that path. An isolated feasible pose that cannot be reached along the prescribed path is not counted as a valid sample. Forward and reverse motion are assessed using their respective prescribed starts and paths. Full sampled coverage does not constitute a proof of continuous collision freedom between samples.

C.3.4 Coupled-Motion Correctness (CMC)

Coupled-Motion Correctness measures whether driver inputs produce the required follower responses, including follower participation, direction, phase, transmission relationships, and required travel. For requirement $r \in R_i$, define

$$c_{ir} = \mathbf{1}\left[r \text{ is satisfied over its prescribed interval}\right]. \tag{15}$$

The object-level score is

$$\mathrm{CMC}_i = \frac{1}{|R_i|} \sum_{r \in R_i} c_{ir}. \tag{16}$$

Let

$$\mathcal{D}_C = \{i \in \mathcal{D} : |R_i| > 0\} \tag{17}$$

denote the objects with coupled-motion requirements. The aggregate score is

$$\mathrm{CMC} = \frac{1}{|\mathcal{D}_C|} \sum_{i \in \mathcal{D}_C} \mathrm{CMC}_i \times 100\%. \tag{18}$$

Observed displacement alone is insufficient to satisfy a coupled-motion requirement. A gear rotating in the wrong direction or with an incorrect transmission relationship fails the corresponding requirement. Likewise, small oscillations do not satisfy a requirement for substantial follower travel.

Objects without coupled-motion requirements are excluded from the CMC denominator rather than assigned a vacuous perfect score. CMC and VMC assess response correctness and motion validity, respectively: a correct local follower response does not establish validity over the complete prescribed motion.

C.3.5 Overall Success Rate (OSR)

Overall Success Rate uses a binary object-level decision. An object succeeds only if all of the following conditions hold: the result is executable; all predesignated critical structural requirements are satisfied; all applicable critical motion requirements are satisfied; the prescribed paths have complete valid sampled coverage; and the overall layout is consistent with the functional structure of the reference object.

Let $o_i \in \{0, 1\}$ denote this decision. Then

$$\mathrm{OSR} = \frac{1}{n} \sum_{i \in \mathcal{D}} o_i \times 100\%. \tag{19}$$

OSR is not an arithmetic average of the other metrics. A model that satisfies many local requirements but lacks a critical connection, has an incorrect critical transmission, or cannot complete the prescribed motion is not counted as an overall success. Successful termination of the internal Solver does not by itself establish acceptance under the final evaluation protocol.

C.4 Geometric Agreement Metrics

C.4.1 Correspondence, Alignment, and Normalization

Geometric metrics compare the shape and spatial arrangement of a generated model with its manually constructed and annotated reference model. They are reported separately from structural and motion metrics. Part correspondences are established by functional role, and fixed submeshes belonging to the same rigid body may be merged into one comparison unit. These correspondences remain fixed throughout a motion sequence.

Geometry is compared after a single global coordinate and scale alignment. Distances are normalized by the reference model's bounding-box diagonal in the designated reference configuration. Dynamic comparisons retain the same alignment transform. Individual parts or frames are not realigned independently, because doing so could remove genuine layout or motion errors.

C.4.2 Part Bounding-Box Agreement: mIoU and gIoU

Let B and B^* be the aligned three-dimensional axis-aligned bounding boxes of a generated part and its reference counterpart. Their intersection over union is

$$\mathrm{IoU}(B, B^*) = \frac{\mathrm{Vol}(B \cap B^*)}{\mathrm{Vol}(B \cup B^*)}. \tag{20}$$

The object-level mean IoU is

$$\mathrm{mIoU}_i = \frac{1}{J_i} \sum_{j=1}^{J_i} \mathrm{IoU}(B_{ij}, B^*_{ij}), \tag{21}$$

where J_i is the number of reference comparison units.

Let C be the smallest axis-aligned bounding box enclosing both B and B^*. Generalized IoU is

$$\mathrm{gIoU}(B, B^*) = \mathrm{IoU}(B, B^*) - \frac{\mathrm{Vol}(C) - \mathrm{Vol}(B \cup B^*)}{\mathrm{Vol}(C)}. \tag{22}$$

Object-level gIoU is likewise averaged over parts. Higher mIoU and gIoU indicate closer agreement in spatial position and extent. These are three-dimensional part bounding-box metrics, not pixel-overlap metrics computed from rendered images.

C.4.3 Surface Distances: PC and OC

Point clouds are sampled from the normalized mesh surfaces. For two nonempty point sets P and Q, we use the symmetric mean-squared Chamfer distance:

$$\begin{aligned} d_{\mathrm{CD}}(P, Q) = &\frac{1}{2|P|} \sum_{p \in P} \min_{q \in Q} \|p - q\|_2^2 \\ &+ \frac{1}{2|Q|} \sum_{q \in Q} \min_{p \in P} \|q - p\|_2^2. \end{aligned} \tag{23}$$

Part-level surface distance, PC, is computed for corresponding parts and then averaged within each object:

$$\mathrm{PC}_i = \frac{1}{J_i} \sum_{j=1}^{J_i} d_{\mathrm{CD}}(P_{ij}, P^*_{ij}). \tag{24}$$

Object-level surface distance, OC, compares point clouds sampled from the complete object surfaces:

$$\mathrm{OC}_i = d_{\mathrm{CD}} \left(P_i^{\mathrm{obj}}, P_i^{*\mathrm{obj}} \right). \tag{25}$$

Lower PC and OC are better. PC emphasizes the shape and location of corresponding parts, whereas OC measures the discrepancy between complete object surfaces. Surface sampling uses common sampling rules and random seeds. Missing reference parts receive a common missing-part penalty; they are not removed from the comparison set before averaging over the remaining parts.

C.4.4 Static and Dynamic Comparisons

Static comparisons use corresponding reference configurations. Dynamic comparisons evaluate generated and reference models at corresponding driver inputs. For a per-state geometric metric g, define

$$g_i^{\mathrm{dyn}} = \frac{1}{K_i} \sum_{k=1}^{K_i} g\left(X_i(u_{ik}), X_i^*(u_{ik})\right), \tag{26}$$

where $X_i(u_{ik})$ and $X_i^*(u_{ik})$ denote the generated and reference states at the corresponding input.

When multiple paths are evaluated, scores are first averaged within each path and then across paths. State correspondence is established using actual driver coordinates or physical travel rather than frame indices alone. When discrete recordings provide only nearby inputs, the input mismatch is recorded and the pairing is treated as approximate. Non-executable results are not assigned dynamic geometric scores by constructing substitute poses.

Geometric similarity does not replace mechanism correctness. Models with similar shapes may still contain incorrect connections, while functionally correct mechanisms may retain local shape discrepancies.

C.5 Aggregation, Failure Handling, and Confidence Intervals

C.5.1 Aggregation and Failure Handling

ER and OSR aggregate binary object-level outcomes. SC, VMC, and CMC are computed per object and then macro-averaged. Geometric metrics are likewise aggregated at the object level; parts and motion frames are not pooled as independent observations.

Generation failures, missing final artifacts, and results that cannot form executable objects are treated as failures in the primary task metrics. For CMC, a failed object remains in the denominator whenever it belongs to the set with coupled-motion requirements. Infrastructure interruptions are recorded separately from generated-program failures and handled under a common retry policy. Failed objects are not removed to improve reported performance.

Position, angle, joint-limit, collision, and follower-travel judgments use common evaluation conventions. Permitted contact and prohibited penetration are distinguished using the reference requirements. Adjacency or the presence of a connection does not automatically exempt a pair of parts from collision assessment.

C.5.2 95% Confidence Intervals

All statistical analyses use the object as the unit of analysis. For the binary metrics ER and OSR, we use two-sided 95% Wilson score intervals. Let $\hat{p}$ be the observed success proportion, n the number of objects, and $z = \Phi^{-1}(0.975)$, where Φ is the standard normal cumulative distribution function. The interval is

$$\frac{\hat{p} + \dfrac{z^2}{2n} \;\pm\; z\sqrt{\dfrac{\hat{p}(1-\hat{p})}{n} + \dfrac{z^2}{4n^2}}}{1 + \dfrac{z^2}{n}}. \tag{27}$$

For SC, VMC, and CMC, whose object-level scores lie in $[0, 1]$, we use conservative mean intervals obtained by inverting a Chernoff–Hoeffding bound. Let n be the number of eligible objects and $\bar{x}$ their mean score. The interval consists of the values $\mu \in [0, 1]$ satisfying

$$n\,\mathrm{KL}(\bar{x} \| \mu) \le \log \frac{2}{\alpha}, \qquad \alpha = 0.05, \tag{28}$$

where

$$\mathrm{KL}(p \| q) = p \log \frac{p}{q} + (1-p) \log \frac{1-p}{1-q}. \tag{29}$$

Boundary cases are defined by continuity. This construction retains nonzero interval width when all observed scores are zero or one. Interval endpoints are converted to percentages for reporting.

Each configuration uses its actual evaluation set to determine the sample size. Both Kimi configurations contain 20 objects, and both GPT-6 Astra configurations contain 19. CMC excludes the dishwasher, which has no coupled-motion requirement, giving sample sizes of 19 and 18, respectively.

These are object-level marginal confidence intervals under an assumption of independent objects from a comparable population. They do not quantify repeated-generation variability or annotation and adjudication error, and they do not provide simultaneous coverage across multiple metrics. The intervals do not make the curated evaluation set a random population sample. The means in the main text describe performance on the fixed evaluation set; the intervals supplement them with uncertainty under the stated statistical assumptions.

C.5.3 Confidence Interval Results

Table 10 reports the two-sided 95% confidence intervals for all five primary metrics. Entries are interval endpoints, not means or standard deviations. The corresponding mean scores are reported in the main text.

Table 10: Object-level two-sided 95% confidence intervals. All endpoints are percentages.

Method	ER	SC	VMC	CMC	OSR
PhysX-Anything	[11.2, 46.9]	[0.0, 25.9]	[0.0, 16.8]	[0.0, 17.6]	[0.0, 16.1]
URDF-Anything+	[48.1, 85.5]	[0.1, 28.9]	[0.0, 16.8]	[0.0, 17.6]	[0.0, 16.1]
ArtLLM	[58.4, 91.9]	[7.3, 57.3]	[0.0, 16.8]	[0.0, 17.6]	[0.0, 16.1]
Instruct-Particulate	[34.2, 74.2]	[13.6, 67.7]	[0.0, 16.8]	[0.0, 17.6]	[0.0, 16.1]
Procedura	[69.9, 97.2]	[40.5, 91.5]	[0.1, 31.0]	[0.7, 38.1]	[0.0, 16.1]
Articraft (Kimi-K3)	[83.9, 100.0]	[32.8, 86.7]	[0.0, 16.8]	[5.6, 55.2]	[0.0, 16.1]
Kimi-K3 Direct API	[58.4, 91.9]	[25.4, 80.8]	[0.0, 16.8]	[31.1, 86.5]	[0.0, 16.1]
MARC (Kimi-K3)	[83.9, 100.0]	[83.2, 100.0]	[37.8, 90.0]	[77.0, 100.0]	[38.7, 78.1]
GPT-6 Astra Direct API	[83.2, 100.0]	[74.7, 100.0]	[10.4, 64.2]	[75.9, 100.0]	[15.4, 54.0]
MARC (GPT-6 Astra)	[83.2, 100.0]	[82.4, 100.0]	[82.4, 100.0]	[81.5, 100.0]	[83.2, 100.0]

Both GPT-6 Astra configurations use 19 objects; all other configurations use 20. CMC excludes the dishwasher, giving denominators of 18 and 19, respectively.

C.6 Ablation 1: Backbone Model and Harness Comparison

Ablation 1 compares direct API generation with the complete MARC harness for each backbone model. Within each backbone, the two configurations independently generate results from the same reference inputs and use the same object set and evaluation criteria. Direct API generation does not include MARC's staged execution diagnostics and reflective correction, whereas the full-harness configuration executes the complete workflow.

This experiment measures the system-level effect of the full harness, including tool calls, execution feedback, and reflective correction. It does not attribute the observed difference to any single component. Sample counts are disclosed separately for each backbone. Cross-backbone comparisons should distinguish results on each backbone's complete evaluation set from results restricted to their common object subset.

Figure 5 provides a qualitative comparison of the generated results. Structural and motion judgments follow the evaluation protocol defined in this appendix.