

APPENDIX

We evaluate *cuArms* on three additional multi-arm planning benchmarks: multi-goal, multi-modal path planning (MRMG, App. A), one-shot motion planning (xECBS, App. B), and bimanual LEGO assembly (APEX-MR, App. C). Then, we investigate whether coding agents can replace a hand-written scheduler in App. D and whether a coding agent can act as both the perception and the scheduling agent on a real bimanual robot (App. E). Unless otherwise denoted, t_0 is the time to the first solution, and J_0 and J_f are the costs of the first and final solutions.

A. Long-Horizon Multi-Arm Manipulation

Environment. MRMG [4] has four assembly tasks derived from [A1], as shown in Fig. 9. Two to four UR5 arms pick and place 7 to 24 parts to build a two-arm spiral tower (Spiral-2), a four-arm spiral tower (Spiral), a three-arm truss (Truss), and a four-arm cube (Cube). We fix the order of the picks and places, robot assignment, and joint-space goal(s) in the schedule skeleton according to the original benchmark. *cuArms* must then coordinate multiple arms through each assembly step while the collision geometry changes as objects are carried and placed at their targets.

Metric. We use the same cost function described in the original benchmark to directly compare with the published results in MRMG. The cost J sums, over the steps of a path, the longest joint-space distance traveled by any arm plus 1% of the joint distances of all arms [4, Eq. 2]. t_0 includes building the roadmaps.

Baselines. We compare against the two planners whose results are published with the benchmark: 1) a prioritized planner and 2) AIT* [A2], an asymptotically optimal sampling-based planner. The published results are medians of 25 runs, with final costs after 500 s (Spiral-2) or 1000 s of planning.

Results. Across 15 seeds per task, *cuArms* achieves a 100% success rate and finds an initial plan in 0.3–7.5 s, 3.5–24 $\times$ faster than AIT* and 16–152 $\times$ faster than prioritized planning. These first plans cost 21–35% less than AIT*’s. Prioritized planning achieves lower initial costs but takes 14–197 s to find a feasible solution. Refinement reduces the cost of *cuArms* by about 50% within 10–60 s (Fig. 9). The final plans found by *cuArms* cost 4.5–16.5% less than prioritized planning’s on every task and are within 8% of AIT*’s cost (lower on Spiral), which plans for 500–1000 s.

B. One-Shot Multi-Arm Motion Planning

Environment. We evaluate the six scenes of 7-DoF Franka Panda arms from the xECBS paper [3] (Fig. 10). In the Two, Four, Six, and Eight Robots environments, the arms stand on a circle around an open workspace. The Bin Picking environment features four arms reaching into a row of bins and the Shelves environment has eight arms reaching into one shelf unit. Each scene has 50 queries that move every arm from a start to a goal configuration within 60 s. We use all 300 and leave out a Ten-Robot scene whose files the authors flag as possibly inconsistent.

TABLE V: **MRMG [4].** Medians over 15 seeds for *cuArms*, refined for 10, 30, 30, and 60 s; published results for the baselines.

Task	Robots	Parts	<i>cuArms</i>			Prioritized			AIT*		
			t_0	J_0	J_f	t_0	J_0	J_f	t_0	J_0	J_f
Spiral-2	2	7	0.32	85.0	44.4	14.14	47.7	46.6	2.02	110.7	43.9
Spiral	4	15	1.29	202.2	88.9	196.74	122.9	106.5	31.00	311.5	94.0
Truss	3	14	1.07	144.0	68.9	24.68	89.3	76.8	3.77	211.9	66.4
Cube	4	24	7.53	336.3	129.5	117.86	161.5	135.6	59.44	423.7	119.8

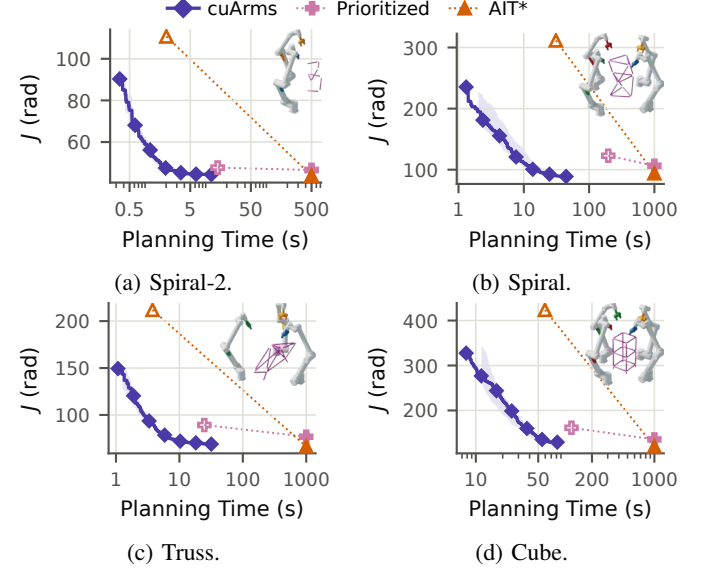


Fig. 9: **MRMG [4] cost over planning time.** The completed structures (bold) and the arms’ final configurations are shown in the top-right corners. Dotted lines join the baselines’ published first and final solutions.

Metric. We again keep the benchmark’s cost function for consistency: J (rad) is the joint-space distance (L_1) each arm travels, summed over the arms. Waiting time is not included in the cost. We count *cuArms*’s and VAMP-MR’s plans only if they pass our validator on the benchmark’s robot model. *cuArms* runs three seeds per query, builds its roadmaps once per scene (2.9–16.5 s, not counted in t_0), and refines each first plan for 10 s.

Baselines. xECBS speeds up ECBS, a bounded-suboptimal conflict-based search over the arms’ individual paths, by reusing paths from earlier searches. We also compare against the ECBS and Prioritized Planning baselines. We use the authors’ published per-query results, which have a 60 s time limit, evaluated on an Intel Core i9-12900H CPU. We also compare against VAMP-MR [5], a CPU-vectorized multi-arm planner that runs a composite RRT-Connect or a conflict-based search on roadmaps (CBS-MP). We run it once per query on an Intel Core Ultra 9 285K CPU with a 60 s time limit and then run shortcutting for 10 s.

Results. *cuArms* solves the most queries in five of the six scenes: it achieves a 100% success rate in Bin Picking and

TABLE VI: **xECBS benchmark [3]**. Success over 50 queries per scene (*cuArms*: three seeds each); medians of t_0 (s) and J_f (rad) over each method’s solved queries. xECBS, ECBS, and Prioritized: published results. Bold: best per scene.

Scene	<i>cuArms</i>			xECBS			ECBS			Prioritized			VAMP-MR (RRT)			VAMP-MR (CBS-MP)		
	Succ.	t_0	J_f	Succ.	t_0	J_f	Succ.	t_0	J_f	Succ.	t_0	J_f	Succ.	t_0	J_f	Succ.	t_0	J_f
Bin Picking	100%	0.083	26.5	96%	3.49	23.5	82%	13.1	23.0	82%	8.06	23.0	32%	0.51	24.6	32%	0.080	24.1
Shelves	38%	0.24	44.8	84%	7.98	40.3	40%	20.7	35.8	34%	21.6	36.6	0%	—	—	6%	0.49	33.4
Two Robots	100%	0.026	12.5	100%	0.092	12.6	100%	0.121	12.6	98%	0.148	12.6	98%	0.0002	12.5	96%	0.005	12.6
Four Robots	100%	0.052	19.3	98%	0.395	19.4	96%	0.994	19.5	82%	0.663	19.7	84%	0.006	19.9	84%	0.033	19.6
Six Robots	100%	0.083	33.7	92%	3.49	31.3	68%	13.3	31.6	64%	7.96	31.5	72%	0.76	39.0	58%	0.30	33.4
Eight Robots	98%	0.13	41.9	80%	9.44	40.9	34%	15.8	37.0	40%	14.8	36.3	64%	13.1	47.6	32%	2.51	39.4

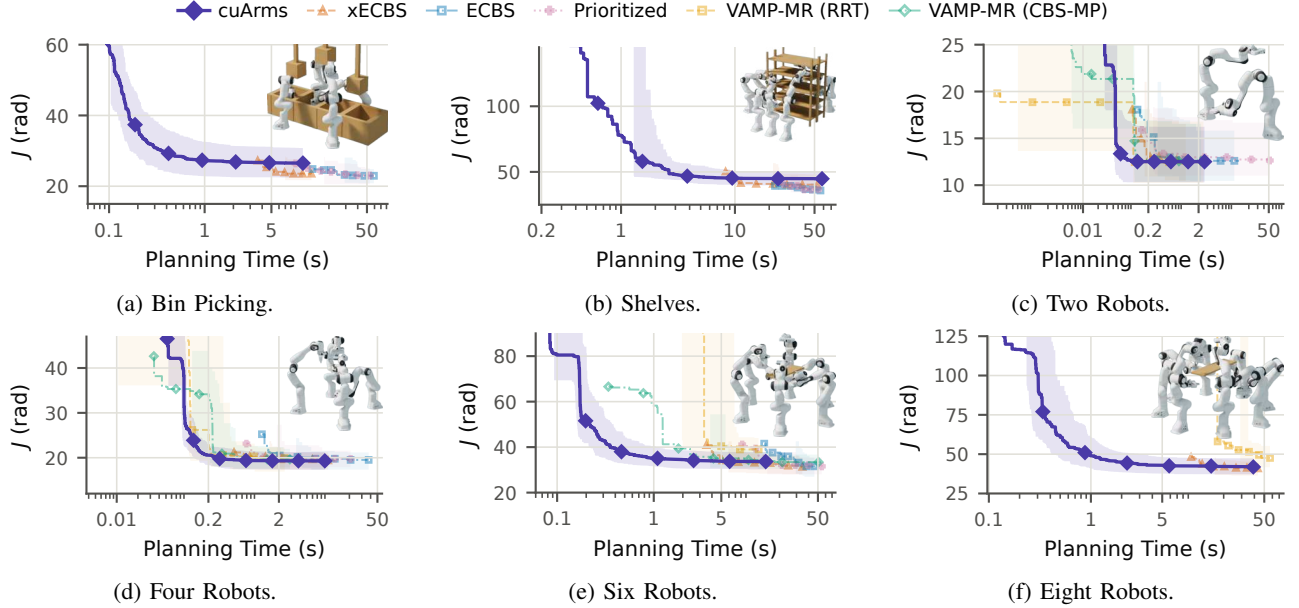


Fig. 10: **xECBS benchmark [3] cost over planning time**. The environment and robot configurations are shown in the top-right corners. Only methods that solve at least 50% of a scene’s queries (30% in Shelves) are shown.

with two, four, and six robots, and 98% with eight (Table VI). xECBS solves 80–100% of the queries in these scenes, and VAMP-MR as few as 32%. *cuArms* also finds first plans fastest, in a median 26–243 ms, $3.5\text{--}57\times$ faster than xECBS on the queries both solve. VAMP-MR is faster only on easy queries, in the open Two- and Four-Robot scenes (0.2–33 ms). On these queries, *cuArms*’s time is mostly per-query setup that connects the start and goal to the roadmaps; the search itself takes only about 20% of the time to the first solution. On harder scenes such as Bin Picking and Eight Robots, GPU acceleration and conflict-driven search make *cuArms* much faster than VAMP-MR’s CPU-based RRT (83 and 130 ms against 0.51 and 13.1 s). After refinement, *cuArms*’s cost is within 1% of xECBS’s with two and four robots and 6–13% higher in the other scenes (geometric means), where xECBS takes $22\text{--}57\times$ longer to find its first plan.

However, in the eight-arm Shelves environment, *cuArms* achieves a success rate of 38% against xECBS’s 84%. Our search expands states with PIBT, which resolves conflicts by pushing one agent at a time out of the way. Moving one arm collision-free often requires several other arms to be pushed away. With eight large arms reaching

TABLE VII: **LEGO assembly**. Medians of the planning time t (s) and makespan J (s) of the initial plan and of the final plan after 5 s of post-processing (20 s on Big Chair). The results are computed with three seeds. Bold: better per task.

Task	Bricks	<i>cuArms</i>				VAMP-MR			
		t_0	J_0	t_f	J_f	t_0	J_0	t_f	J_f
Cliff	12	1.30	207	5.0	144	1.37	273	7.5	145
Vessel	36	2.18	318	7.5	264	1.29	547	8.3	278
RSS	47	3.15	618	8.5	447	6.36	1007	16.4	514
Big Chair	258	13.6	1911	39.8	1756	3.36	3465	48.2	1789

into the same shelf, many arms contend for the same space as they approach their respective goals. Since PIBT often cannot resolve the multi-arm dependencies easily, resolving these conflicts in the search within the time limit remains challenging.

C. Multi-Arm LEGO Assembly

Environment. We evaluate four LEGO assembly tasks from APEX-MR [1], built by two Yaskawa GP4 arms with 12 to 258 bricks (Fig. 11). *cuArms* uses the same assembly order,

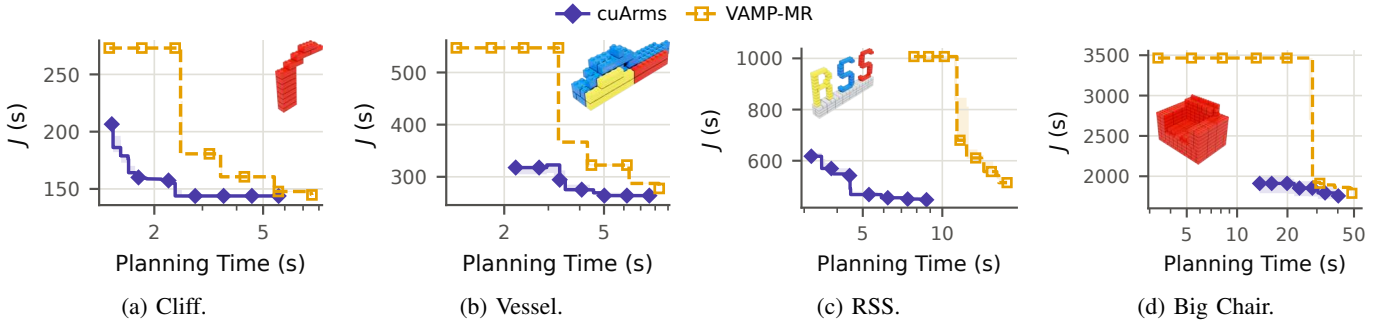


Fig. 11: **LEGO assembly [1] makespan over planning time.** The target LEGO structures are shown in the plot. VAMP-MR first finds an initial sequential plan, then post-processes it into asynchronous motion with TPG followed by shortcutting.

precedence constraints, task assignment, and skill parameters as APEX-MR as the schedule skeleton and plans all robot arm motions and skill executions.

Metric. J is the execution makespan, the time until both arms finish. Each motion runs at the joint velocity limit, and the arms move asynchronously, each waiting only where the other arm or the assembly order requires. Both *cuArms* and VAMP-MR execute their plans with a temporal plan graph (TPG).

Baseline. VAMP-MR [5] runs APEX-MR’s planner with its CPU-vectorized collision checker. The motion planner in VAMP-MR is specifically engineered for the domain: it plans one arm at a time with RRT between hand-designed waypoints around each brick’s pick and place locations, while assuming the other arm waits at a collision-free home pose. Then, VAMP-MR builds a TPG so both arms can move asynchronously instead of sequentially and then shortcuts that plan. Both methods get the same post-processing time, 5 s (20 s on Big Chair): refinement for *cuArms* and TPG shortcutting for VAMP-MR.

Results. As shown in Table VII, *cuArms* finds better plans on all four assembly tasks, and its final plans are ready sooner. *cuArms*’s initial plans are 24–45% shorter than VAMP-MR’s sequential plans, and the time-to-initial-solution is within a second of the sequential, single-robot planning in VAMP-MR on Cliff and Vessel and twice as fast on RSS. On Big Chair, *cuArms* needs 13.6 s for 258 bricks while VAMP-MR’s sequential plan takes 3.4 s; however, building the TPG that turns it into an asynchronous plan takes another 25 s. In the end, *cuArms*’s final plans are 1–13% shorter on all four tasks.

Qualitatively, we found that *cuArms* produces higher-quality motions with fewer detours than VAMP-MR. VAMP-MR plans each robot’s motions one at a time, between a hand-engineered home pose and each skill’s target pose. While this makes motion planning easier and faster, the motions can be suboptimal and often require more waiting from another arm. *cuArms* plans both arms’ trajectories simultaneously without manual waypoint engineering, so the arms move directly between skills, and either arm yields or waits when they would collide.

D. Coding Agents as the Scheduler

The schedulers of Sec. V are hand-engineered: even a domain-agnostic temporal planner such as ScheduleStream relies on an expert-written domain description, with its skills $\mathcal{A}$ and samplers for their parameters x_j , to generate candidate skeletons Σ for *cuArms* to ground. Instead, a coding agent can write this layer for each new problem, using *cuArms* as a tool to plan the coordinated motions (Fig. 12).

1) *Related Work:* Code as Policies [A3] was among the first frameworks in which an LLM writes programs that control a robot from natural-language instructions by composing perception and control primitives. Follow-up work uses LLMs to generate PDDL domains or problems for classical planners [A4], [A5], to synthesize generalized plans for PDDL domains [A6], to guide TAMP through generated constraints [A7], and to coordinate multiple arms through LLM agents that negotiate subtasks and waypoints [A8]. With recent reasoning models, coding agents improve their robot programs through multi-turn execution feedback [A9], and Merler et al. [A10] use coding agents to synthesize general programs for TAMP problems directly from a simulator. In our framework, a coding agent acts as a zero-shot scheduler that synthesizes and refines the task-level program for each problem, calling the *cuArms* GPU planner as a tool to coordinate multi-arm motions and skills.

2) *Method:* We expose *cuArms* through a small Python API that mirrors Sec. III-A. A *scene* defines the workspace $\mathcal{W}$; each skill a_j on one or two robots is defined by a joint trace, which gives q_j^{in} , q_j^{out} , and $\mathcal{B}_j$, a minimum duration d_j , and a grasp or release effect T_{a_j} ; a *skeleton* Σ orders the skills with precedence constraints $\prec_\Sigma$; and a *cost* sets the weights of $J(\tau)$. The planner fills in the *moves* and refines the timed plan τ until a deadline the agent sets, with default parameters (e.g., roadmap size and random seed) that the agent can change in code. Geometric utilities (kinematics, Cartesian paths, and collision checks) and a renderer help the agent write and inspect skills and plans. We describe the task’s goal, point the agent to the API, and ask it for a final plan, which is verified independently. Each problem is solved zero-shot in a fresh sandbox, without any domain-specific scheduler, samplers, or problem structure. In practice, the agent inspects the scene

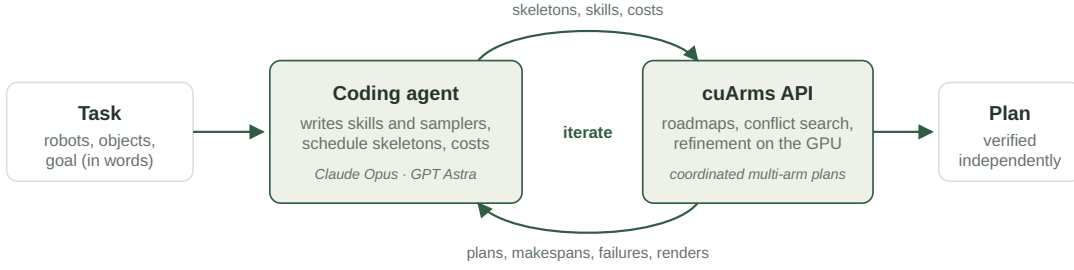


Fig. 12: **A coding agent as the scheduler.** Given the task, the coding agent writes skills, schedule skeletons, and costs and calls the *cuArms* API; *cuArms* returns coordinated plans and feedback. The agent iterates and outputs a plan that is verified independently.

and task, writes programs that choose the task assignment and ordering, joint targets, grasps, and object placements, evaluates the planned results, and iterates until deciding on a final plan.

The agent uses the following Python interface, where poses are $(x, y, z, q_w, q_x, q_y, q_z)$ and configurations are joint vectors.

```
Scene(robots, obstacles, objects) # Robot(id, model, base,
    start); Box / Mesh(id, ..., pose)
Option(trace={robot: q[T, dof]}, effect_tick=None)
Action(id, robots, options, effect=None, # effect: Grasp(obj
    , robot, ee_T_obj) | Release(obj, robot)
    min_duration=0.0, paced=False)
Skeleton(actions, before=(), exchanges=(), end={}) # per-
    robot order, cross-robot order, handovers
Cost(makespan=1.0, completion=1.0, distance=0.1, metric="
    joint") # weights of J(tau)
planner = Planner(scene, seed=0, roadmap_nodes=1024)
with planner.solve(skeleton, cost, deadline=t) as run: #
    streams anytime Solutions
    for solution in run: ... # run.best, run.status, run.
        error
geo = planner.geometry # fk, ik, limits, cartesian_path(s),
    # collides, trace_clear
Renderer(scene).image(plan, t, camera) / .video(plan, path)
```

In the notation of Sec. III-A, an Action is a skill a_j , and its Options are alternative joint traces that determine q_j^{in} , q_j^{out} , and $\mathcal{B}_j$. min_duration is d_j , and the effect defines T_{a_j} . The Skeleton’s action order and before pairs give $\prec_\Sigma$, and the Cost weights are those of $J(\tau)$. A carry is a two-robot action on a shared object, and a handover is a pair of single-robot actions linked in exchanges. The planner compiles the skeleton into *cuArms*’s multi-modal problem. It first checks that grasps meet their objects, releases leave objects on a support, and actions on one object are ordered, and reports violations by action name.

3) *Setup: Agents and harnesses.* We run Claude Opus 5.5 in Claude Code and GPT-6 Astra in Code, both with high reasoning efforts. Each session has a 20-minute wall-clock limit. The agent may read the installed library source, write and run code in its workspace, render images and videos, and run the solution validator. It cannot access our repository, the baselines, file systems, the web, or other problems.

Sandbox. Each session runs in a fresh Docker container with one GPU, Python with NumPy, SciPy, PyTorch, and cuRobo, and the measured module closure of the *cuArms* API. The API is read-only and owned by root, and the agent runs as an unprivileged user. A default-deny firewall allows only the model API endpoint. The workspace holds a read-only copy of

the validator, and a trusted copy outside the container scores the final plan.

Validator. The validator rebuilds the scene and replays the plan, checking configurations at most 0.02 rad apart per joint. It checks the start configurations, the joint position and velocity limits, and collisions among robots, obstacles, and held and resting objects with *cuArms*’s sphere models (6 mm penetration tolerance). Contact is allowed only between a robot and the object it grasps or releases, within 1 s of the event, and between two hands in a handover. Each grasp must meet the object within 5 mm and 0.05 rad, each release must leave the object supported, and hands holding one object must agree on its pose. Weld targets must be reached within 2.5 cm and 15° and held for the dwell, and objects must end on their goal surfaces or at their goal poses. It also reports the *executable makespan*, which is the makespan after the uniform slowdown that respects 15 rad/s² and 500 rad/s³ with corners blended over 0.1 s.

Benchmark and baseline. We run each agent once per instance (20-minute sessions) on 10 instances per family: Pack and Unpack with 2–8 objects and Stack with 2–4 objects on the tabletop, and RoboBallet with four arms and 10 or 40 targets and with eight arms and 40 targets. The baseline is the hand-written scheduler of Sec. V (Hungarian matching for RoboBallet, ScheduleStream for the tabletop) with *cuArms* as the planner. It is not compute-matched: it uses about a minute where an agent uses up to 20, but relies on hand-written domain code.

4) *Results:* On RoboBallet, Astra+ reaches 0.77–0.82× and Opus+ 0.94–1.07× the hand-written scheduler’s final makespan, at the price of minutes instead of seconds to the first plan (Table VIII). Across the tabletop families with 2–8 objects, Opus+ and Astra+ reach 0.51–0.79× and 0.32–0.61× the hand-written scheduler’s makespan on the same instances. Although the coding agents find better plans than the hand-written schedulers, each plan costs minutes of agent time and API usage, and how they improve on the hand-written schedules is not obvious.

Domain-specific improvements. In RoboBallet, the makespan depends mostly on how the welds are allocated, so the agents re-pose weld targets within the task’s tolerance, pick inverse-kinematics solutions, and re-assign targets to

TABLE VIII: **Coding agents as the scheduler.** Opus+ and Astra+: Opus and Astra as the scheduler with *cuArms*, one 20-minute session per instance; their t_0 is the time to the first validated plan. Solved instances, and medians of t_0 (s) and the makespan J (s) over instances all three solve. Bold: best per task.

Task	<i>cuArms</i>			Opus+			Astra+		
	Solved	t_0	J	Solved	t_0	J	Solved	t_0	J
RoboBallet 4×10	10/10	0.29	5.69	10/10	163	6.30	10/10	144	4.45
RoboBallet 4×40	10/10	0.49	18.1	10/10	341	18.9	10/10	169	15.2
RoboBallet 8×40	10/10	0.85	15.6	9/10	651	13.9	10/10	327	12.1
Stack (2–4)	30/30	0.85	7.01	30/30	279	4.06	30/30	118	3.48
Pack (2–8)	38/50	5.17	7.96	50/50	237	5.49	50/50	110	2.99
Unpack (2–8)	48/50	4.89	9.43	49/50	311	5.67	50/50	127	3.27

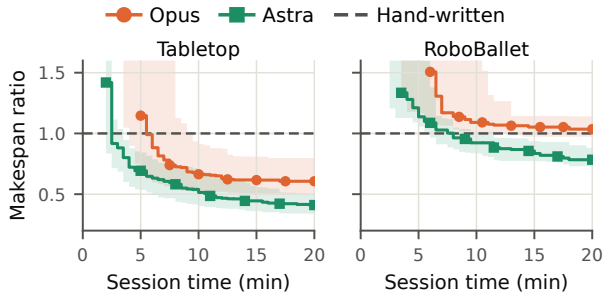


Fig. 13: **Anytime behavior of the agents.** Median best validated makespan relative to the hand-written scheduler over session time, for tabletop manipulation and RoboBallet.

balance the work across arms. In tabletop manipulation, they spend more of their effort on the skills: they adjust object placements, grasps, and approach and lift motions, which shortens every pick and place. Astra occasionally stitches together or retimes the planner’s trajectories, but usually submits them unchanged: its final plan is *cuArms*’s trajectory unchanged in 150 of 160 problems. Table IX counts these task-level techniques in each session’s code; keyword matching only approximates which techniques a session uses.

Comparison of agents. Both agents usually reach a first valid plan within minutes, but Astra keeps improving it for most of the session (Fig. 13). Astra writes scripts that sweep many parameters and settings, such as target assignments, grasp angles, lift heights, and planner seeds, and keeps the best validated plan. Opus changes one setting at a time and usually stops after about half of its session. On the 160 benchmark problems, Opus’s sessions last a median of 9.3 min (47% of the budget) with 11 commands, and Astra’s last 15.9 min (80%) with 68 commands. Across families, Astra’s plans are shorter at about 10–25× the cost.

Failures. Opus fails on two of the 160 benchmark problems. On one 8-arm RoboBallet instance, it found no collision-free configuration that reaches one weld target within tolerance, so it submitted a plan that welds the other 39 targets and gave up with that explanation, without claiming success. On one Unpack6 instance, it hit the wall-clock limit before writing a

TABLE IX: **Task-level techniques.** Share of agent sessions with *cuArms* whose code uses each technique, detected by keyword matching over the code and commands each agent wrote.

Technique (share of sessions)	Opus		Astra	
	Tabletop	RoboBallet	Tabletop	RoboBallet
placement / drop-spot search	29%	0%	64%	0%
approach-height sweep	65%	0%	98%	0%
grasp or tool angle sweep	98%	0%	95%	0%
weld pose re-posed within tolerance	0%	43%	5%	100%
assignment / route search	1%	33%	28%	50%
own timing (offsets, waits, LP)	15%	3%	62%	77%

plan.

Executable makespan. Retiming to the executable makespan lengthens the plans by a median factor of 1.49 for Astra, 1.66 for Opus, and 1.62 for the hand-written skeletons planned through the API. The agents’ shorter plans therefore do not come from more abrupt motion.

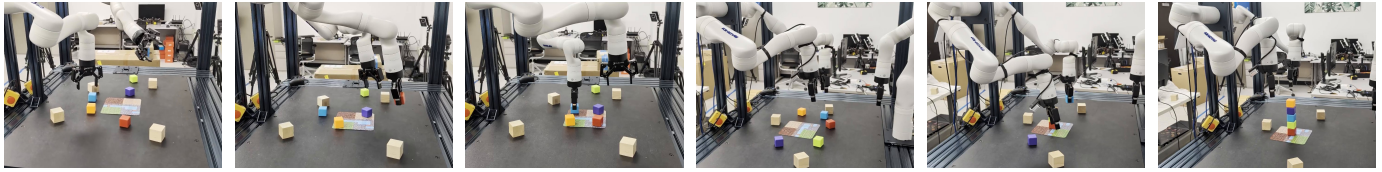
E. Real-Robot Tabletop Manipulation with a Coding Agent

We also conduct additional real-robot experiments for tabletop manipulation with a coding agent for both perception and scheduling (Fig. 14). Our setup includes two 7-DoF Kinova Gen3 arms arranged in a humanoid-like configuration with both arms facing outward, and a ZED Mini camera mounted at the head position. Given a natural-language description of the task (e.g., pack all the colored magnetic blocks) and a camera image, the coding agent reconstructs the planning scene in simulation, then plans the schedule skeleton and generates coordinated bimanual manipulation plans with the *cuArms* API.

For the experiment, we assume that the robot configurations, camera pose, table height, and object shapes are known. The agent is given access to a SAM 3 [A11] inference endpoint and is prompted to use its VLM capabilities and SAM 3 to identify each object on the table; the pose of each object (i.e., xy position and yaw orientation) is then estimated by fitting a rendered box of its known shape to its SAM 3 mask. Then, following the procedure in App. D, the agent assigns the tasks, generates the grasp poses and object placements, schedules the execution order, and calls *cuArms* to plan the coordinated manipulation plan. The resulting trajectories are then visualized and inspected by a human before being executed on the real robot, where each arm’s motions are retimed with TOPP-RA [A12] and synchronized through a TPG [1].

APPENDIX REFERENCES

- [A1] J. Chen, J. Li, Y. Huang, C. Garrett, D. Sun, C. Fan, A. Hofmann, C. Mueller, S. Koenig, and B. C. Williams, “Cooperative task and motion planning for multi-arm assembly systems,” arXiv preprint arXiv:2203.02475, 2022.
- [A2] M. P. Strub and J. D. Gammell, “Adaptively informed trees (AIT*): Fast asymptotically optimal path planning through adaptive heuristics,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2020, pp. 3191–3198.



(a) **Pack.** Both arms place the colored blocks on the mat.

(b) **Stack.** Both arms stack the colored blocks into a tower.

Fig. 14: **Real-robot tabletop manipulation with a coding agent and *cuArms*.**

- [A3] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng, “Code as policies: Language model programs for embodied control,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2023, pp. 9493–9500.
- [A4] L. Guan, K. Valmeekam, S. Sreedharan, and S. Kambhampati, “Leveraging pre-trained large language models to construct and utilize world models for model-based task planning,” in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 79 081–79 094.
- [A5] B. Liu, Y. Jiang, X. Zhang, Q. Liu, S. Zhang, J. Biswas, and P. Stone, “LLM+P: Empowering large language models with optimal planning proficiency,” arXiv preprint arXiv:2304.11477, 2023.
- [A6] T. Silver, S. Dan, K. Srinivas, J. B. Tenenbaum, L. Kaelbling, and M. Katz, “Generalized planning in PDDL domains with pretrained large language models,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 18, 2024, pp. 20 256–20 264.
- [A7] N. Kumar, W. Shen, F. Ramos, D. Fox, T. Lozano-Pérez, L. P. Kaelbling, and C. R. Garrett, “Open-world task and motion planning via vision-language model generated constraints,” *IEEE Robotics and Automation Letters*, vol. 11, no. 3, pp. 3366–3373, 2026.
- [A8] Z. Mandi, S. Jain, and S. Song, “RoCo: Dialectic multi-robot collaboration with large language models,” in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2024.
- [A9] L. Fu, J. Yu, K. El-Refai, E. Kou, H. Xue, H. Huang, W. Xiao, L. Fei-Fei, G. Shi, J. Wu, S. S. Sastry, Y. Zhu, K. Goldberg, and L. Fan, “CaP-X: A framework for benchmarking and improving coding agents for robot manipulation,” in *Proceedings of the 43rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 306, 2026, pp. 32 155–32 212.
- [A10] M. Merler, B. Li, J. Roy, Y. Liang, Q. Wang, Y. Huang, and T. Silver, “Coding agents for generalized task and motion planning problems,” arXiv preprint arXiv:2609.30233, 2026.
- [A11] N. Carion *et al.*, “SAM 3: Segment anything with concepts,” in *International Conference on Learning Representations (ICLR)*, 2026.
- [A12] H. Pham and Q.-C. Pham, “A new approach to time-optimal path parameterization based on reachability analysis,” *IEEE Transactions on Robotics*, vol. 34, no. 3, pp. 645–659, 2018.