CONTACT CODE: A TYPED PROGRAM INTERFACE FOR MEASURABLE AND REPAIRABLE CONTACT-RICH MANIPULATION

Anonymous authors
Paper under double-blind review

ABSTRACT

The ability to adapt to physical feedback is essential for robots performing contact-rich manipulation. Vision-language models (VLMs) provide a promising way to interpret task instructions and configure robot behavior from visual observations. However, adapting these decisions after physical interaction requires connecting execution feedback to the parameters that can be revised. We introduce Contact Code, a typed program interface that supports this connection by organizing controller parameters, measured execution evidence, and permitted updates around individual contact stages. Within predefined task templates, a VLM proposes a task configuration that is validated and instantiated into an executable program. During execution, robot motion and contact measurements provide feedback for controller-level recovery and evidence for subsequent program revision. If execution still fails, the system uses this evidence to select a contact stage and guides the VLM to propose a constrained update before the next attempt. We evaluate Contact Code on eight tasks and 800 held-out instances across ManiSkill, Isaac Lab, and UniVTAC-based simulation, spanning wrist force and fingertip tactile sensing as well as single-arm and bimanual settings. Contact Code increases success from 60.4% on the first rollout to 73.6% after repair and outperforms free-form parameter repair by 8.3% under the same feedback and rollout budget. On a Flexiv arm, the same planning and repair pipeline, with only the platform-facing adapters changed, improves peg-insertion success from 80% on the first trial to 96% after repair, providing evidence of portability across robot embodiments and control APIs. This interface provides a foundation for studying how physical feedback can guide program revision, offering a path toward more adaptable language-guided robotic systems.

1 INTRODUCTION

Reliable contact-rich manipulation requires adapting robot behavior to what happens during physical interaction. Small variations in pose, friction, compliance, or grasp can alter an interaction after motion begins (Noseworthy et al., 2025; Tang et al., 2023; Xu et al., 2019). Consider the insertion case in Figure 1 (a): the commanded motion continues, the actual motion stalls, and the contact force rises. These observations indicate obstructed progress, but they do not uniquely identify its cause or the parameter change that would resolve it. To revise the failed program, the robot must connect this partial evidence to the decisions that can be changed before another attempt.

Vision-language models (VLMs) offer a way to interpret task instructions and configure robot behavior through programs built from reusable APIs (Liang et al., 2023; Fu et al., 2026). Prior work supports contact-rich execution through compliant actions and force-conditioned termination (Burns et al., 2024), and incorporates execution feedback, failure explanation, and graph-structured policies to revise behavior (Huang et al., 2022; Liu et al., 2023; Lu et al., 2026; Chen et al., 2026b). These capabilities provide components for acting, observing, and correcting. Connecting them requires an explicit correspondence between the physical interaction and the program being revised. In the insertion example, the repair process needs to establish which stage the measurements describe, which controller parameters belong to that stage, and which of those parameters a proposed update may change.

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

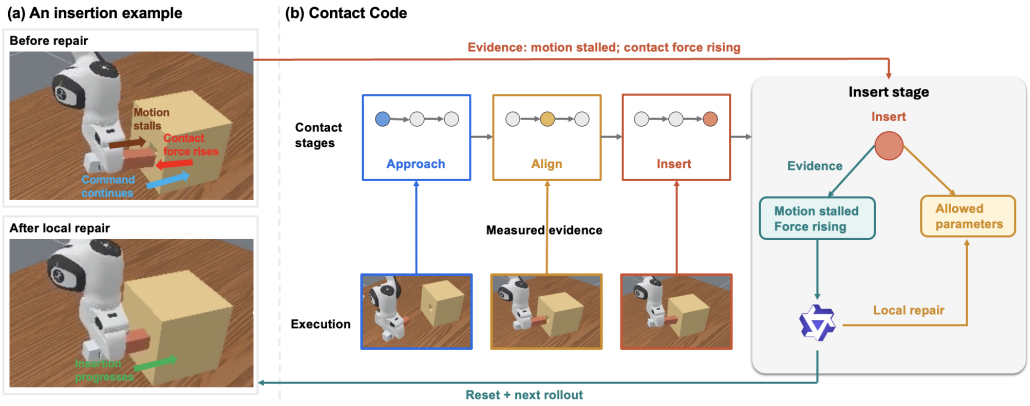


Figure 1: **Contact Code uses physical feedback to guide program updates.** (a) The motion command continues, but the peg stops and contact force rises. The lower image shows insertion after a repair. (b) The graph links each stage to its measured feedback and allowed parameter changes. The VLM proposes a typed repair. The system checks it, resets the task, and tries again.

We introduce Contact Code, a typed program interface that uses contact stages to organize execution and repair. Each stage associates controller parameters with measured execution evidence and permitted repair fields; dependencies between stages form a directed primitive graph (Figure 1 (b)). A task contract specifies the graph template, legal parameter fields, and constraints. Given a task instruction and an initial observation, a VLM proposes the exposed fields of a typed task configuration, or TaskSpec, which the system validates and uses to instantiate the program. Task-specific adapters supply executable grounding, motion candidates, and controllers within the fixed graph template. This structure connects what the model may configure, what execution can reveal, and what a subsequent repair may revise.

During execution, Contact Code records commanded and actual robot motion together with wrist force or fingertip tactile measurements. Evidence adapters associate these histories with the corresponding stages, while task-specific controllers may use the measurements for bounded recovery within a rollout. After a failed rollout, the system selects a repair base from the executed configurations. A dependency-aware scheduler uses execution evidence and repair history to select an unresolved stage and its permitted fields. The VLM proposes a local patch, which the system checks against the task contract. Fields outside the permitted scope retain their values from the repair base. After accepting the patch, the system resets the task and executes the revised program in a new rollout. Thus, controller-level recovery acts within an execution, whereas program repair changes the configuration for a subsequent attempt.

We evaluate Contact Code on eight tasks and 800 held-out instances across ManiSkill, Isaac Lab, and UniVTAC-based simulation, covering two contact-sensing modalities and both single-arm and bimanual settings. Contact Code increases success from 60.4% on the first rollout to 73.6% after repair, recovering 106 initial failures. Under the same execution feedback and rollout budget, it improves average final success over free-form parameter repair by 8.3%. We further deploy the same planning and repair interface on a Flexiv arm, changing only the platform-facing adapters. Across 25 physical trials, stage-local repair increases success from 80% to 96% and recovers 80% of the initial failures. Controlled comparisons and ablations examine the roles of the initial VLM proposal, measurement-conditioned execution, and program repair. These results support organizing feedback and permitted updates around contact stages within task-defined execution frameworks, providing a basis for studying how physical measurements can guide robot program revision. Our contributions are:

- A typed primitive-graph interface that associates each contact stage with controller parameters, measured execution evidence, and contract-permitted repair fields.
- An execution and repair procedure that uses stage dependencies and execution evidence to restrict VLM-proposed updates, and validates each patch before a new rollout.

- An evaluation on 8 simulated tasks and 800 held-out instances, with controlled comparisons under matched feedback and rollout budgets alongside ablations of proposal, execution, and repair, complemented by real-robot peg-insertion experiments that provide evidence of transfer across robot embodiments.

2 Related Work

Language Models for Robot Program Generation. Language models have been used to translate task instructions into executable robot programs built from reusable perception and control (Ahn et al., 2022; Singh et al., 2022; Huang et al., 2023). Prior work spans waypoint-based and reactive policies, deployable ROS programs, compliant contact actions, and modular graph-structured policies (Liang et al., 2023; Chen et al., 2024; Burns et al., 2024; Chen et al., 2026b). CaP-X studies how action interfaces, perception, interaction, and execution feedback affect these coding agents (Fu et al., 2026). Contact Code provides the primitive-graph program structure and lets the VLM instantiate a typed TaskSpec by selecting task grounding, a success criterion, and contract-exposed execution parameters. When execution fails, the system uses motion and contact measurements to select a stage to repair. The VLM then updates only the parameters permitted for that stage.

Contact-Rich Manipulation with Physical Feedback. Contact-rich manipulation methods use geometry, compliance, and feedback to handle changes in pose and contact conditions (Tang et al., 2023; 2024; Burns et al., 2024; Noseworthy et al., 2025; Suomalainen et al., 2022). Language-conditioned policies increasingly incorporate wrist force measurements into action generation and hybrid force-position control (Yu et al., 2026; Li et al., 2026). In parallel, tactile simulators and visuotactile policies use fingertip measurements to represent local contact, slip, and contact change (Akinola et al., 2025; Hao et al., 2025; Zhang et al., 2026). These approaches primarily use physical measurements as controller or policy inputs. Contact Code also uses them to revise the program after failure. It records motion and contact signals for each program step, then uses these records to guide parameter changes in a selected step.

Program Validation, Execution Feedback, and Repair. Closed-loop language-model systems use environment observations, success signals, visual differences, multisensory histories, and execution traces to explain failures or revise robot behavior (Huang et al., 2022; Liu et al., 2023; Fu et al., 2026; Lu et al., 2026; Chen et al., 2026c). Natural-language correction methods instead use human feedback to revise robot objectives, constraints, or motion plans(Sharma et al., 2022). Complementary work uses explicit program structure to support inspection and validation, including syntax and simulation checks for generated programs, readable behavior trees(Colledanchise & Ögren, 2018), and graph-structured policy refinement (Chen et al., 2024; Styrud et al., 2025; Chen et al., 2026b). Together, these directions establish the value of execution feedback, structured policies, and iterative correction. Contact Code connects these elements by linking execution feedback to program updates. The system uses this feedback and dependencies between stages to decide which parameters the VLM may change. It then checks the proposed changes against the task contract before running the revised program.

3 Method

3.1 Overview

Contact Code connects a proposal made before contact to physical evidence that becomes available only during execution. Its central representation consists of a typed TaskSpec T and a directed primitive graph G. The TaskSpec makes the VLM's choices explicit, while the graph uses the same contact stages to organize parameter ownership, measured evidence, and repair scope.

Given a task instruction ℓ and initial observation o_0, Contact Code first checks an unvalidated VLM proposal against the task contract and combines its accepted fields with a task-specific fallback to form a validated TaskSpec. It then generates candidate motions and instantiates an executable program. During execution, it records robot motion and contact measurements and permits bounded controller-level recovery. A rejected precheck produces structured violations without a physical rollout, and a failed rollout produces measured execution evidence. In either case, the next VLM update is restricted to contract-permitted fields; after an executed failure, those fields are further

localized to one relevant contact stage. For executed failures, Algorithm 1 summarizes the repair procedure from stage-aligned execution evidence to a validated stage-local patch. This pipeline makes the Contact Code program verifiable before execution, its execution measurable during contact, and its stage-local decisions repairable through contract-permitted updates grounded in measured evidence, without requiring ground-truth knowledge of the contact state. Appendix A provides the complete initialization, precheck, execution, and repair procedure.

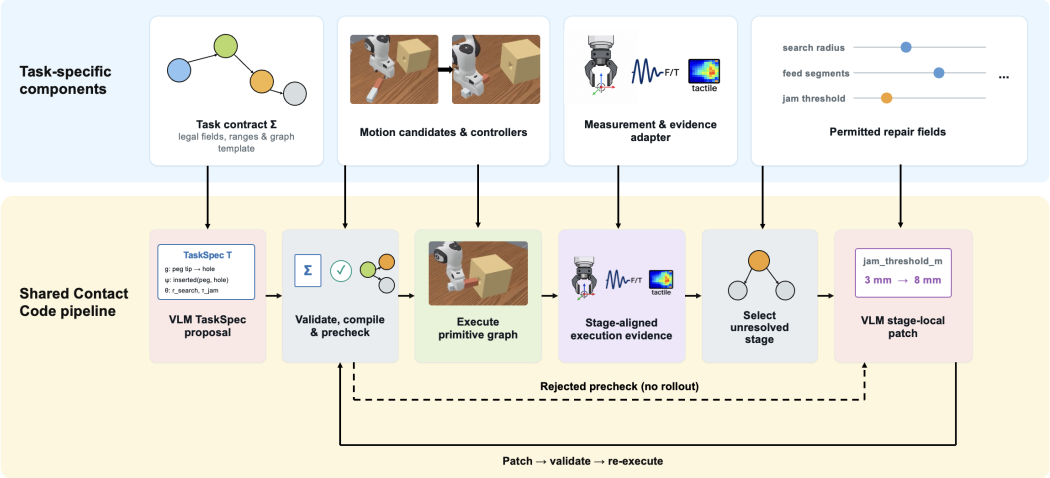


Figure 2: **Overview of the Contact Code.** Given a task and visual observations, a VLM proposes a TaskSpec. The system checks it against the task contract, builds a primitive graph, and compiles and checks the program before execution. Motion and contact measurements guide recovery during execution and provide evidence for repair. After an execution failure, the system selects one stage, and the VLM changes only its allowed parameters. Each update is checked before the next attempt. Failed prechecks also return to the repair loop.

3.2 Verifiable Programs

For each task, a task contract Σ defines the legal TaskSpec fields, a fixed primitive-graph template, and the system bindings needed for execution. A TaskSpec $T = (g, \psi, \theta)$ records task grounding g, a success criterion ψ, and contract-exposed execution parameters θ.

Let q_{prop} denote the VLM proposal mapping. Given the task instruction ℓ and initial observation o_0, the VLM produces an unvalidated proposal

$$\widetilde{T}_0 = q_{\text{prop}}(\ell, o_0; \Sigma). \tag{1}$$

Contact Code checks the proposal against Σ. For a usable proposal, it retains the contract-valid VLM fields and fills the remaining fields from a task-specific fallback TaskSpec. If the proposal is unusable, it uses the fallback TaskSpec in full. We denote the resulting validated TaskSpec by T_0. This initialization handles invalid or missing proposal fields through fallback completion rather than stage-local repair; no primitive graph is instantiated and no physical rollout is consumed until the validated TaskSpec T_0 has been formed. The VLM can therefore set only contract-exposed fields. The proposed success criterion records the VLM's interpretation of the goal, while the task-specific evaluator determines the official outcome.

The VLM expresses motion choices by name rather than directly predicting 6-DoF poses. The system generates and checks the corresponding motion configurations from o_0, then resolves the choices in T_0 to a concrete configuration c_0. This keeps semantic choices with the VLM and geometric motion generation with the system.

The validated TaskSpec and motion configuration instantiate the Contact Code program as a directed primitive graph G_0, which is then compiled into an executable policy:

$$\begin{aligned} G_0 &= \text{instantiate}_\Sigma(T_0, c_0) = (V_0, E_0), \\ \pi_0 &= \text{compile}_\Sigma(T_0, G_0). \end{aligned} \tag{2}$$

Contact Code Algorithm 1: Repair from Measured Execution Evidence

System inputs: failed rollout history H_i, executed TaskSpec T_i and graph G_i, task contract Σ, and repair history $\mathcal{D}_{<i}$

Output: updated TaskSpec T_{i+1} or failure

`// Step 1: Align measurements with contact stages`

$e_i \leftarrow \mathcal{A}_\Sigma(H_i, G_i)$

`// Step 2: Select repair base, evidence, and unresolved stage`

Select $b(i)$ from the current and prior failed executions using task-specific measured progress

Select e_i^{rep} from their measured evidence for diagnosis

$(v_i, \mathcal{F}_i) \leftarrow \text{schedule}_\Sigma(G_{b(i)}, e_i^{\text{rep}}, \mathcal{D}_{<i})$

`// Step 3: Propose and validate a stage-local patch`

Obtain δ_i from $q_{\text{rep}}(T_{b(i)}, e_i^{\text{rep}}, v_i, \mathcal{F}_i, \mathcal{D}_{<i})$ and validate it against Σ and $\mathcal{F}_i$

if *no valid δ_i is obtained* **then**

  return failure

$T_{i+1} \leftarrow T_{b(i)} \oplus \delta_i$

Update $\mathcal{D}_{<i+1}$ and **return** T_{i+1}

Each node in V_0 is a contact stage with controller parameters, observable evidence fields, and permitted repair fields; each edge in E_0 is a stage dependency. Before execution, Contact Code validates this graph and applies the available geometric and task-specific prechecks. Passing these checks verifies conformance to the implemented interface, not physical task success. Appendix C lists all eight task templates and their repair-field scopes, with representative TaskSpec and static-precheck examples.

3.3 Measurable Execution

We use i to index the current program iteration. At iteration i, the validated TaskSpec T_i and primitive graph G_i are compiled into an executable policy π_i; $i = 0$ denotes the initial program. If π_i passes the static precheck and is executed, Contact Code records the history up to time t,

$$H_{i,t} = \left(x_{i,0:t}^{\text{cmd}}, x_{i,0:t}^{\text{act}}, y_{i,0:t}\right), \tag{3}$$

where x^{cmd} and x^{act} are the commanded and actual Tool Center Point (TCP) poses. The contact signal y is wrist force/torque or fingertip tactile sensing, depending on the task. If the rollout terminates at time τ_i, its complete history is $H_i = H_{i,\tau_i}$. The task-specific evaluator then reports a binary success indicator $s_i \in \{0, 1\}$.

Raw measurements have different meanings under different contact mechanisms. Let $\mathcal{A}_\Sigma$ denote the task-specific evidence adapter. It organizes the execution history H_i according to the contact stages in G_i, producing the stage-aligned evidence record

$$e_i = \mathcal{A}_\Sigma(H_i, G_i). \tag{4}$$

The record summarizes quantities such as measured progress, commanded-to-actual motion residuals, completed motion segments, and contact-recovery events. It associates these quantities with contact stages without requiring one universal contact-state classifier.

During a rollout, task-specific controllers may use these measurements for bounded controller-level contact recovery without invoking the VLM. If the rollout still fails, Contact Code passes the stage-aligned evidence e_i to program repair.

3.4 Repairable Program Decisions

After a failed rollout, the system selects a repair base from the current and previously executed TaskSpecs. It ranks the failed executions using task-specific measured evidence and retains the best-ranked version. Let $b(i)$ denote the index of the selected version, so that $T_{b(i)}$ and $G_{b(i)}$ are the TaskSpec and primitive graph used as the repair base.

Together with the repair base $T_{b(i)}$, the system selects the measured execution evidence used for the current repair, denoted by e_i^{rep}. The repair base and repair evidence may come from different

Table 1: **Task success in simulation.** We evaluate six assembly tasks using wrist force/torque feedback and two tasks—HDMI insertion and key removal—using fingertip tactile feedback. Contact Code raises overall success from 60.4% to 73.6% after repair and achieves much higher final success than GenCHiP and CaP-X on their supported tasks. These results show that Contact Code improves contact manipulation with both force and tactile feedback.

	Contact Code				Baselines	
Task	First	Final	Δ	Rollouts	GenCHiP	CaP-X M4-style
Wrist force/torque						
MS Peg	73	81	**+8**	1.50	n/a	0
MS Plug	75	89	**+14**	1.39	n/a	0
FG Peg	77	81	**+4**	1.28	20	8
FG Nut	23	44	**+21**	2.40	1	0
FG Gear	49	69	**+20**	1.91	2	12
MS Bimanual	46	55	**+9**	2.02	n/a	0
Fingertip tactile						
UV HDMI	52	74	**+22**	1.79	n/a	2
UV Key	88	96	**+8**	1.16	n/a	0
Overall	60.4	**73.6**	**+13.2**	1.68	—	—

Note: Success is reported in %; Δ is Final minus First, in %. Rollouts: mean physical executions per instance. We implement GenCHiP (Burns et al., 2024) and CaP-X M4-style (Fu et al., 2026) in these task environments; n/a marks unsupported pairs. MS: ManiSkill; FG: Isaac Forge; UV: UniVTAC.

executed iterations: the former determines which parameter values are retained, while the latter provides the failure diagnosis used to select the next update. The repair model receives e_i^{rep} and previous outcomes, but not ground-truth knowledge of the underlying contact state.

Using the repair-base graph $G_{b(i)}$, repair evidence e_i^{rep}, and repair history $\mathcal{D}_{<i}$, a dependency-aware scheduler selects an unresolved contact stage v_i and the repair fields $\mathcal{F}_i$ that the task contract permits the VLM to change:

$$(v_i, \mathcal{F}_i) = \text{schedule}_\Sigma \left(G_{b(i)}, e_i^{\text{rep}}, \mathcal{D}_{<i} \right), \tag{5}$$

where $\mathcal{D}_{<i}$ records previous TaskSpecs, execution evidence, repairs, and outcomes. The scheduler also tracks previous interventions to avoid immediately repeating an unsuccessful update.

Let q_{rep} denote the repair mapping implemented by the VLM. Given the selected repair base $T_{b(i)}$, repair evidence e_i^{rep}, selected contact stage v_i, permitted repair fields $\mathcal{F}_i$, and repair history $\mathcal{D}_{<i}$, the VLM proposes a typed patch δ_i:

$$\delta_i = q_{\text{rep}} \left(T_{b(i)}, e_i^{\text{rep}}, v_i, \mathcal{F}_i, \mathcal{D}_{<i} \right), \qquad T_{i+1} = T_{b(i)} \oplus \delta_i, \tag{6}$$

where $\oplus$ applies the patch only to the permitted repair fields, while all other fields retain their values from $T_{b(i)}$.

Contact Code validates each proposed patch against the task contract. A valid patch produces T_{i+1}, from which the system instantiates G_{i+1}, compiles π_{i+1}, resets the task, and begins the next rollout. An invalid patch is rejected without modifying the TaskSpec or consuming a physical rollout. A rejected static precheck instead returns structured violations for contract-constrained correction and also consumes no physical rollout. This repair loop continues until the task succeeds or either the program-iteration limit or the physical-rollout limit is reached.

4 Experiments

We evaluate whether Contact Code can complete contact-rich tasks and improve after a failed attempt. We further analyse which program and execution choices contribute to these results, and how the initial proposal and repair affect success.

4.1 Simulation Setup

We evaluate eight tasks and 800 held-out instances across ManiSkill (Tao et al., 2024), Isaac Lab (Mittal et al., 2025), and UniVTAC-based (Chen et al., 2026a) simulation, with 100 instances per task. We use environment seeds 100 to 199 for evaluation, with disjoint instances used for system development. The tasks cover peg and plug insertion, nut threading, gear meshing, bimanual insertion, HDMI insertion, and key removal. Six tasks use wrist force/torque measurements; the two UniVTAC tasks use fingertip tactile measurements.

Each instance starts with a task instruction and an initial observation. The proposal receives the initial sensor bundle and reset-time geometry exported by the simulator, while oracle state, evaluator-internal poses and metrics, and final object poses remain hidden from proposal, execution control, diagnosis, and repair (Appendix B). Contact Code instantiates and executes a policy from a validated TaskSpec, and may revise its contract-exposed execution parameters after failure. Both the initial proposal and subsequent repairs use Qwen3-VL-Plus (Bai et al., 2025) with temperature 0.1. Controller-level recovery acts within a rollout; program repair is followed by a task reset to the same seeded initial state and a new rollout. Each instance is limited to four program iterations, including the initial program, and three physical rollouts. Invalid patches and rejected prechecks do not consume a physical rollout. Within each repair step, the system makes up to five attempts to obtain a valid repair patch.

We report first-rollout success, final success if any rollout within budget succeeds, and mean physical rollouts per instance; execution stops after the first success. Differences in success are reported in %. The repair-only comparison reports recovery on the matched subset of 287 seeds defined in Section 4.4.

4.2 Simulation Results

Contact Code raises success from 60.4% on the first rollout to 73.6% after repair, recovering 106 initial failures with 1.68 rollouts per instance on average (Table 1). Success increases on all eight tasks. The largest gains occur in HDMI insertion (22%), nut threading (21%), and gear meshing (20%). Nut threading remains the hardest task, with 44% final success. Thus, repair helps across the evaluated tasks but does not resolve all contact failures.

Table 1 compares Contact Code with task-adapted reimplementations of GenCHiP and CaP-X M4, all using Qwen3-VL-Plus. GenCHiP is evaluated only on Forge, whose backend provides the required Cartesian impedance interface, whereas CaP-X M4 is evaluated on all eight tasks. These results characterize our implementations under matched task interfaces and budgets, rather than scores reported in the original papers; full protocols are provided in Appendix D.

4.3 Program Repair and Contact Execution

Restricting program updates. Under matched measured feedback and budgets, Contact Code exceeds free-form parameter repair by 8.3% on average (Figure 3a). Free-form repair may change parameters across stages, whereas Contact Code patches only permitted fields in one selected stage. Replacing measured evidence in the free-form condition with CaP-X M4-style visual-difference feedback widens the gap to 14.8%. These comparisons show complementary benefits from typed repair scope and measured contact evidence. Appendix F visualizes these interface differences and reports the single-turn and contract-removal comparisons.

Using feedback during execution. Figure 3b shows how changes to execution affect final success. *Blind fixed* removes all measurement-conditioned execution decisions. *Fixed verdict* retains the measurements but replaces the online contact-state label with a fixed label. Their effects vary widely across tasks. For example, blind fixed reduces success by 73% on MS Plug and 86% on UV Key, but causes no drop on FG Gear.

The sensor ablations also depend on the task. Removing wrist force/torque reduces success by 23% on both MS Peg and MS Plug, while removing tactile input reduces success by 48% on UV HDMI and 85% on UV Key. Disabling controller-level contact recovery reduces success by 29% on MS Peg and 12% on FG Nut. Some ablations have little effect or slightly improve the reported score, so the results

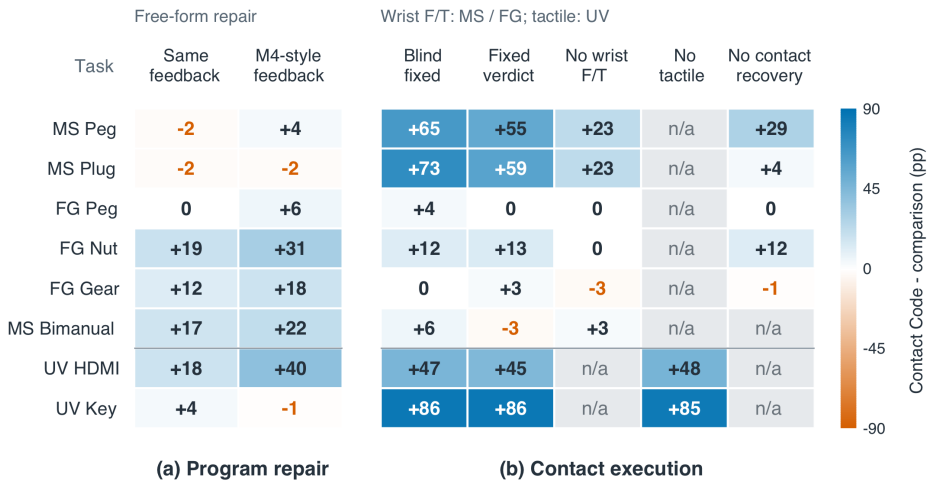


Figure 3: **Effects of program repair and contact feedback.** (a) With measured execution feedback matched, Contact Code exceeds free-form repair by 8.3% on average. Replacing only the measured evidence in the free-form condition with CaP-X M4-style feedback decreases its average success by a further 6.5%, widening the gap to 14.8%. (b) Removing wrist force/torque or fingertip tactile feedback reduces success by up to 23% and 85%, respectively. These results support limiting which parameters can change and using feedback during execution, though benefits vary by task. Values show Contact Code minus each comparison in final success, in %. Blue favors Contact Code; orange favors the comparison; gray marks cases that do not apply.

do not establish that every component is needed for every task. Overall, measurement-conditioned execution is most valuable when task success depends on resolving local contact uncertainty, while the relevant sensing and recovery mechanism remains task specific.

4.4 Initial Proposal and Failure Repair

Initial proposal. With VLM repair in both configurations, using a VLM proposal instead of a heuristic proposal raises first-rollout success from 51.9% to 60.4% and final success from 65.3% to 73.6% (Figure 4a). The heuristic-proposal configuration with uniform sampling within the contract-defined domains of the permitted repair fields reaches 62.3% final success. The two heuristic-proposal configurations use the same deterministic initial TaskSpec, and their repair strategies are invoked only after the first rollout fails. The small difference in first-rollout success (53.1% versus 51.9%) therefore reflects repeat-execution variation rather than a difference in the initial proposal. We isolate the effect of the repair method through the matched-initial-failure comparison below.

Repairing the same initial failures. The full evaluation contains 317 first-rollout failures. For the repair comparison, we retain the 287 seeds whose first rollout fails in all four conditions, ensuring that every repair method is evaluated from an initial failure on the same seed. For each matched seed, all four conditions start from the same frozen initial TaskSpec, use the same controller and task contract, reset the environment to the same seed-specific initial state before each subsequent rollout, and share a total per-instance budget of at most four program iterations and three physical rollouts. On this matched set, retrying the unchanged TaskSpec recovers 6 cases, uniform search over legal parameter values recovers 61, and VLM repair with measured execution evidence recovers 78 (Figure 4b). VLM repair with only a failure flag recovers 72 cases. Measured evidence therefore adds 6 recoveries, corresponding to 2.1% on the matched set, beyond failure-only feedback. Overall, the comparison identifies two complementary benefits: structured parameter updates enable recovery beyond repeated execution, while measured evidence further guides those updates.

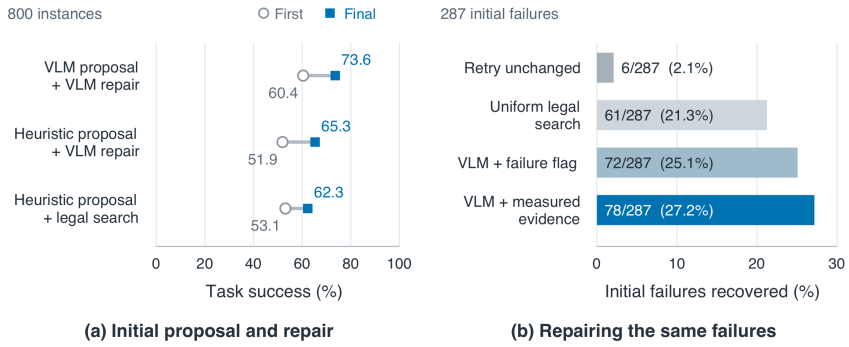


Figure 4: **Effects of the initial proposal and repair feedback.** (a) On 800 instances, a VLM proposal with VLM repair reaches 73.6% final success, compared with 65.3% for a heuristic proposal with the same repair method. (b) On 287 shared initial failures, VLM repair with measured evidence recovers 78 cases (27.2%). These results suggest that combining VLM proposals with measured feedback is a promising way to improve task success and recovery from failure.

4.5 Real-Robot Experiments

We evaluate Contact Code on a peg-insertion setup matching the geometry of ManiSkills PegInsertionSide-v1, using a Flexiv Enlight arm, Flexiv Grav two-finger gripper, and ZED X Nano RGB-D camera. Each trial receives one initial RGB-D observation; registered metric geometry and robot calibration remain private to the hardware executor.

All simulation experiments use Franka/Panda-family arms, whereas the real-robot experiments use a Flexiv Enlight arm. We deploy the same Contact Code planning and repair pipeline, changing only the platform-facing observation and execution adapters. The Flexiv executor realizes the primitive graph with Cartesian motion and impedance control, and exposes TCP motion and the robot-reported six-axis wrench as execution evidence. This setting tests whether the Contact Code interface and stage-local repair procedure transfer across robot embodiments and control APIs.

Across 25 independent physical trials, Contact Code succeeds on 20/25 (80%) first rollouts and 24/25 (96%) within the three-rollout budget, recovering 4/5 initial failures with 1.28 physical rollouts per trial on average. These results provide evidence of transfer of the typed interface and stage-local repair procedure to a Flexiv platform through platform-facing adapters. Appendix E provides the hardware setup and representative physical repairs.

5 Discussion and Conclusion

We presented Contact Code, a typed program interface that connects physical feedback to program repair for contact-rich manipulation. Each contact stage links controller parameters, measured execution evidence, and the changes the VLM may propose. The system uses this structure to select a stage for repair and validate each update before another attempt. Across eight simulated tasks and 800 held-out instances, repair raises success from 60.4% to 73.6%. Controlled comparisons under matched feedback and rollout budgets support the value of typed, stage-local repair, while the matched-failure analysis shows that structured parameter updates and measured evidence provide complementary recovery benefits. Real-robot peg-insertion experiments further provide evidence of transfer across robot embodiments: the deployment changes only the platform-facing adapters, while stage-local repair increases success from 80% to 96%. Together, these findings suggest that explicitly linking contact stages, physical evidence, and editable parameters provides a useful basis for revising robot programs after failure. Reducing the need for predefined task templates remains an important next step toward broader use and more adaptable manipulation. Contact Code does not synthesize new contact skills. Supporting a new contact mode currently requires a fixed primitive graph, task contract, evidence adapter, and controller; our contribution is the typed interface that makes these components verifiable, measurable, and repairable, while learning new primitives or graph structure remains future work.

AI USE STATEMENT

In this work, we used generative AI tools for language editing, clarity, and presentation. We have not used generative AI tools for generating content, experimental results, analysis, or conclusions, and other tasks with required disclosure are not applicable to this work. We have reviewed all AI-assisted work. Specifically, all AI-assisted text was reviewed, revised, and verified by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies simulated and laboratory robot manipulation and does not involve human subjects, personal data, or deployment in public environments. The real-robot experiments were conducted in a controlled laboratory setting with operator supervision. Robot actions and program changes were constrained by task contracts, bounded parameter ranges, static prechecks, and platform-specific operating limits. The experiments were limited to non-human object manipulation, and deployment on additional robots or tasks should follow the corresponding platform validation and operating procedures.

REPRODUCIBILITY STATEMENT

Section 3 and Appendix A describe the complete proposal, validation, execution, and repair procedure. Section 4 and Appendix B specify the evaluated tasks, information boundaries, success metrics, random seeds, and rollout budgets. Appendix C documents the task templates and permitted repair fields with concrete TaskSpec and precheck examples; Appendices D and E describe the comparison implementations and real-robot protocol. Additional controlled comparisons and representative outcomes are provided in Appendices F and C.4. Together, these sections document the model settings, task-specific implementation choices, and evaluation protocols needed to support reproduction of the reported experiments.

REFERENCES

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

Iretiayo Akinola, Jie Xu, Jan Carius, Dieter Fox, and Yashraj Narang. Tacsl: A library for visuotactile sensor simulation and learning. *IEEE Transactions on Robotics*, 41:2645–2661, 2025.

Shuai Bai, Yuxuan Cai, Ruizhe Chen, Keqin Chen, Xionghui Chen, Zesen Cheng, Lianghao Deng, Wei Ding, Chang Gao, Chunjiang Ge, et al. Qwen3-vl technical report. *arXiv preprint arXiv:2511.21631*, 2025.

Kaylee Burns, Ajinkya Jain, Keegan Go, Fei Xia, Michael Stark, Stefan Schaal, and Karol Hausman. Genchip: Generating robot policy code for high-precision and contact-rich manipulation tasks. In *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 9596–9603. IEEE, 2024.

Baijun Chen, Weijie Wan, Tianxing Chen, Xianda Guo, Congsheng Xu, Yuanyang Qi, Haojie Zhang, Longyan Wu, Tianling Xu, Zixuan Li, et al. Univtac: A unified simulation platform for visuo-tactile manipulation data generation, learning, and benchmarking. *arXiv preprint arXiv:2602.10093*, 2026a.

Junting Chen, Yao Mu, Qiaojun Yu, Tianming Wei, Silang Wu, Zhecheng Yuan, Zhixuan Liang, Chao Yang, Kaipeng Zhang, Wenqi Shao, et al. Roboscript: Code generation for free-form manipulation tasks across real and simulation. *arXiv preprint arXiv:2402.14623*, 2024.

Kaiyuan Chen, Shuangyu Xie, Letian Fu, Justin Yu, William Pacini, Sandeep Bajamahal, Hudson Kim, Jaimyn Drake, Daehwa Kim, Haoru Xue, et al. Gap: A graph-as-policy multi-agent self-learning harness for variational automation tasks. *arXiv preprint arXiv:2607.05369*, 2026b.

Siyi Chen, Hugo Hadfield, Alex Zook, Mikaela Angelina Uy, Chan Hee Song, Erwin Coumans, Xuning Yang, Faisal Ladhak, Qing Qu, Stan Birchfield, et al. Volo: A physical orchestrator for open-vocabulary long-horizon manipulation. *arXiv preprint arXiv:2606.07723*, 2026c.

Michele Colledanchise and Petter Ögren. *Behavior trees in robotics and AI: An introduction*. CRC Press, 2018.

Siyuan Dong, Wenzhen Yuan, and Edward H Adelson. Improved gelsight tactile sensor for measuring geometry and slip. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 137–144. IEEE, 2017.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Dantong Niu, Fei-Fei Li, et al. Cap-x: A framework for benchmarking and improving coding agents for robot manipulation. *arXiv preprint arXiv:2603.22435*, 2026.

Peng Hao, Chaofan Zhang, Dingzhe Li, Xiaoge Cao, Xiaoshuai Hao, Shaowei Cui, and Shuo Wang. Tla: Tactile-language-action model for contact-rich manipulation. *arXiv preprint arXiv:2503.08548*, 2025.

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

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models. *arXiv preprint arXiv:2307.05973*, 2023.

Yang Li, Zhaxizhuoma Zhaxizhuoma, Hongru Jiang, Junjie Xia, Hongquan Zhang, Jinda Du, Yunsong Zhou, Jia Zeng, Ce Hao, Jieji Ren, et al. Forcevla2: Unleashing hybrid force-position control with force awareness for contact-rich manipulation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8911–8920, 2026.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International conference on robotics and automation (ICRA)*, pp. 9493–9500. IEEE, 2023.

Zeyi Liu, Arpit Bahety, and Shuran Song. Reflect: Summarizing robot experiences for failure explanation and correction. *arXiv preprint arXiv:2306.15724*, 2023.

Runyu Lu, Yubo Wu, Ethan Kou, Letian Fu, Wenli Xiao, Ajay Mandlekar, Yinzhen Xu, Guanya Shi, Ken Goldberg, Ang Chen, et al. Aspire: Agentic/skills discovery for robotics. *arXiv preprint arXiv:2607.00272*, 2026.

Mayank Mittal, Pascal Roth, James Tigue, Antoine Richard, Octi Zhang, Peter Du, Antonio Serrano-Munoz, Xinjie Yao, René Zurbrügg, Nikita Rudin, et al. Isaac lab: A gpu-accelerated simulation framework for multi-modal robot learning. *arXiv preprint arXiv:2511.04831*, 2025.

Yashraj Narang, Kier Storey, Iretiayo Akinola, Miles Macklin, Philipp Reist, Lukasz Wawrzyniak, Yunrong Guo, Adam Moravanszky, Gavriel State, Michelle Lu, et al. Factory: Fast contact for robotic assembly. *arXiv preprint arXiv:2205.03532*, 2022.

Michael Noseworthy, Bingjie Tang, Bowen Wen, Ankur Handa, Chad Kessens, Nicholas Roy, Dieter Fox, Fabio Ramos, Yashraj Narang, and Iretiayo Akinola. Forge: Force-guided exploration for robust contact-rich manipulation under uncertainty. *IEEE Robotics and Automation Letters*, 10(5): 4436–4443, 2025.

Pratyusha Sharma, Balakumar Sundaralingam, Valts Blukis, Chris Paxton, Tucker Hermans, Antonio Torralba, Jacob Andreas, and Dieter Fox. Correcting robot plans with natural language feedback. *arXiv preprint arXiv:2204.05186*, 2022.

Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

Jonathan Styrud, Matteo Iovino, Mikael Norrlöf, Mårten Björkman, and Christian Smith. Automatic behavior tree expansion with llms for robotic manipulation. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1225–1232. IEEE, 2025.

Markku Suomalainen, Yiannis Karayiannidis, and Ville Kyrki. A survey of robot manipulation in contact. *Robotics and Autonomous Systems*, 156:104224, 2022.

Bingjie Tang, Michael A Lin, Iretiayo Akinola, Ankur Handa, Gaurav S Sukhatme, Fabio Ramos, Dieter Fox, and Yashraj Narang. Industreal: Transferring contact-rich assembly tasks from simulation to reality. *arXiv preprint arXiv:2305.17110*, 2023.

Bingjie Tang, Iretiayo Akinola, Jie Xu, Bowen Wen, Ankur Handa, Karl Van Wyk, Dieter Fox, Gaurav S Sukhatme, Fabio Ramos, and Yashraj Narang. Automate: Specialist and generalist assembly policies over diverse geometries. In *Robotics: Science and Systems*, volume 1, 2024.

Stone Tao, Fanbo Xiang, Arth Shukla, Yuzhe Qin, Xander Hinrichsen, Xiaodi Yuan, Chen Bao, Xinsong Lin, Yulin Liu, Tse-kai Chan, et al. Maniskill3: Gpu parallelized robotics simulation and rendering for generalizable embodied ai. *arXiv preprint arXiv:2410.00425*, 2024.

Jing Xu, Zhimin Hou, Zhi Liu, and Hong Qiao. Compare contact model-based control and contact model-free learning: A survey of robotic peg-in-hole assembly strategies. *arXiv preprint arXiv:1904.05240*, 2019.

Jiawen Yu, Hairuo Liu, Qiaojun Yu, Jieji Ren, Ce Hao, Haitong Ding, Guangyu Huang, Guofan Huang, Yan Song, Panpan Cai, et al. Forcevla: Enhancing vla models with a force-aware moe for contact-rich manipulation. *Advances in Neural Information Processing Systems*, 38:93409–93439, 2026.

Xidong Zhang, Yichi Zhang, Jiaxin Shi, Fucai Zhu, Siyu Zhu, Michael Yu Wang, Xiaojun Wu, and Weihao Yuan. Unitacvla: Unified tactile understanding and prediction in vision language action models. *arXiv preprint arXiv:2606.31723*, 2026.

A Full Contact Code Procedure

Contact Code Algorithm 2: Full Contact Code procedure

System inputs: task instruction ℓ, initial observation o_0, task contract Σ, and limits on program iterations, physical rollouts, and VLM response attempts per repair
Output: success or failure; the validated TaskSpec on success

```
// Initialization: accept valid VLM fields over a task-specific fallback
```

$\widetilde{T}_0 \leftarrow q_{\text{prop}}(\ell, o_0; \Sigma)$
Initialize T_0 with the task-specific heuristic TaskSpec
Validate $\widetilde{T}_0$ against Σ
if *the proposal is usable* **then**
  Merge its accepted fields into T_0
$\mathcal{D}_{<0} \leftarrow \emptyset$
for *each permitted program iteration* $i = 0, 1, \ldots$ **do**
  `// Step 1: Resolve motion and instantiate`
  Generate and check motion candidates from o_0 under Σ
  Attempt to select c_i matching the motion choices in T_i
  if *a valid* c_i *exists* **then**
    $G_i \leftarrow \text{instantiate}_\Sigma(T_i, c_i) = (V_i, E_i)$
    $\pi_i \leftarrow \text{compile}_\Sigma(T_i, G_i)$
    Run the static precheck on T_i and G_i, recording any violations
  else
    Record a structured motion-feasibility violation
  if *a motion-feasibility or static-precheck violation was recorded* **then**
    `// Correct` T_i `from violations; no` e_i `exists for this iteration`
    if *no repair iteration remains* **then**
      return failure
    Request and validate a VLM patch δ_i using the structured violations, within the response-attempt limit
    if *no valid* δ_i *is obtained* **then**
      return failure
    $T_{i+1} \leftarrow T_i \oplus \delta_i$
  else
    `// Step 2: Execute and align measured evidence`
    $(H_i, s_i) \leftarrow \text{rollout}(\pi_i)$ `// ` s_i `comes from the task evaluator`
    if $s_i = 1$ **then**
      return success and T_i
    $e_i \leftarrow \mathcal{A}_\Sigma(H_i, G_i)$ `// Align measurements with contact stages`
    if *no further iteration or physical rollout remains* **then**
      return failure
    `// Step 3: Select a repair base and patch one contact stage`
    Select $b(i)$ from the current or previously executed TaskSpecs using measured progress
    Select repair evidence e_i^{rep} from executed rollouts
    $(v_i, \mathcal{F}_i) \leftarrow \text{schedule}_\Sigma(G_{b(i)}, e_i^{\text{rep}}, \mathcal{D}_{<i})$
    Request and validate δ_i from $q_{\text{rep}}(T_{b(i)}, e_i^{\text{rep}}, v_i, \mathcal{F}_i, \mathcal{D}_{<i})$, within the response-attempt limit
    if *no valid* δ_i *is obtained* **then**
      return failure
    $T_{i+1} \leftarrow T_{b(i)} \oplus \delta_i$
    Reset the task to its initial state
  `// Save this iteration for later repairs`
  Update $\mathcal{D}_{<i+1}$
return failure

B Evaluation Protocol and Task Definitions

This section describes only the simulation experiments conducted in ManiSkill (Tao et al., 2024), Isaac Forge (Mittal et al., 2025; Noseworthy et al., 2025; Narang et al., 2022), and UniVTAC (Chen et al., 2026a). In these experiments, initial geometric information is exported from the simulator through `scene.json`.

Evaluation instances. We evaluate eight tasks, with 100 held-out simulator seeds per task. The reported split uses seeds 100–199, while system development uses disjoint seeds. Each seed is exported once into a scene directory and the same exported scene is shared by all methods evaluated on that task. Contact Code and the controlled repair comparisons reset the simulator to the same seed before executing an updated program. The task-adapted CaP-X M4-style reimplementation instead keeps one persistent episode per seed, following its multi-turn protocol. The seed fixes the task reset but does not make model sampling deterministic. Contact Code and its controlled proposal and repair conditions use temperature 0.1; method-specific decoding settings are given in Section D.

Planning-visible inputs and held-out state. Each exported simulation scene separates information available during planning from evaluator-only information. The proposal pipeline reads `scene.json` together with an initial sensor-image bundle. The task name, environment identifier, task adapter, and grounding vocabulary are provided to the proposal model. A natural-language instruction is additionally provided when one is included in the exported scene. For ManiSkill, the image bundle contains base-camera RGB, depth, and semantic-segmentation previews. For Isaac Forge, it contains base-camera RGB and depth. For UniVTAC, it contains a base-camera RGB image and the two initial GelSight Mini marker images (Dong et al., 2017). The proposal model cannot access `oracle.json`, the final simulator state, or the rollout success label.

Table 2 makes this information boundary explicit. In particular, the binary task outcome returned after a rollout is available for repair, but the poses and intermediate metrics used to compute that outcome remain evaluator-only.

Table 2: Information-access audit for the simulation evaluation. "Yes" indicates that the information is available to the corresponding Contact Code component; evaluator and audit state is never exposed to proposal, execution control, or repair.

Information	Proposal	Execution	Repair	Evaluator / audit only
Reset-time geometry from `scene.json`	Yes	Yes	Inherited via TaskSpec	–
Initial RGB/depth, segmentation, or marker images	Yes	–	–	–
Runtime TCP motion and force/torque or tactile evidence	–	Yes	Yes	–
Official binary rollout outcome	–	–	Yes	Computed
Final object pose and success-metric internals	–	–	–	Yes
Hidden task quantities in `oracle.json`	–	–	–	Yes

Note. Repair receives inherited validated TaskSpec fields, measured execution evidence, and the official binary outcome. It does not receive `oracle.json`, final poses, or intermediate evaluator metrics.

For these simulation experiments, we assume that the initial geometric state of the objects and fixtures is available through a state-estimation interface. In the current implementation, the geometric fields in `scene.json` are exported directly from the simulator reset state rather than estimated by a learned detector. For ManiSkill, these fields are exported without additional pose perturbation. The Isaac Forge exporter similarly records the initial geometry of the held object and fixture after the seeded reset. For UniVTAC, the socket or lock pose is provided, while the connector-to-gripper transform and the initial key rotation are withheld for `UniVTAC-InsertHDMI-v0` and `UniVTAC-PullOutKey-v0`, respectively. These hidden quantities are stored only in `oracle.json` and are unavailable to program proposal, execution control, failure diagnosis, or repair.

The simulation results therefore evaluate program proposal, contact execution, and repair given initial geometric information. They do not evaluate robustness to errors from a learned perception system. The real-robot proposal instead uses an initial RGB-D observation, while registered metric geometry and robot-specific calibration are available only to the hardware executor.

Simulation tasks. The simulation evaluation includes the following tasks:

ManiSkill. `PegInsertionSide-v1` and `PlugCharger-v1`.

Bimanual extension. `BimanualPegInsertionSide-v1`, our bimanual extension of ManiSkill.

Isaac Forge. `Isaac-Forge-PegInsert-Direct-v0`, `Isaac-Forge-NutThread-Direct-v0`, and `Isaac-Forge-GearMesh-Direct-v0`.

UniVTAC. `UniVTAC-InsertHDMI-v0` and `UniVTAC-PullOutKey-v0`.

Figure 5 provides a visual overview of the eight simulation tasks and their feedback modalities.

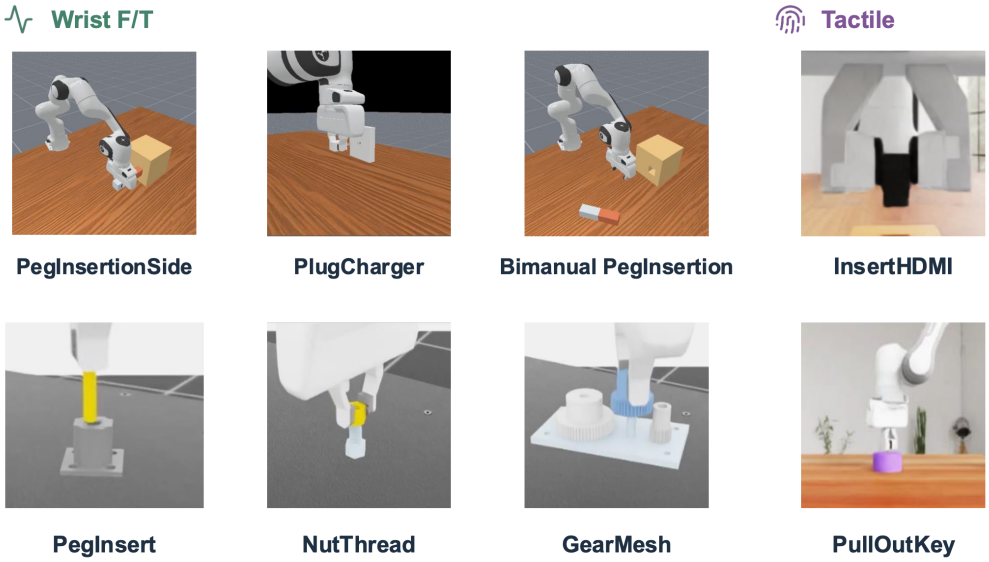


Figure 5: Overview of the eight simulation tasks. The six tasks in the left three columns use wrist force/torque feedback, while the two tasks in the rightmost column use tactile feedback.

ManiSkill tasks. The two single-arm tasks use one Panda under `pd_joint_pos` control (Tao et al., 2024). For `PegInsertionSide-v1`, the environment randomizes peg half-length, peg radius, hole offset, and object poses. The exported peg half-length lies in $[0.085, 0.125]$ m, the radius lies in $[0.015, 0.025]$ m, and radial clearance is 3 mm. `PlugCharger-v1` uses fixed part dimensions but randomized charger and receptacle poses; its slot clearance is 0.5 mm. The exporters derive the hole or slot entry frame from initial object geometry and do not expose the environment goal pose to planning.

Bimanual extension. `BimanualPegInsertionSide-v1` is our custom bimanual extension of ManiSkill's `PegInsertionSide-v1`, rather than an upstream ManiSkill environment. We retain the original peg and socket geometry, bore clearance, engagement depth, and insertion success predicate. We add a second Panda, convert the socket into a free rigid body with a grasp handle, and assign different roles to the two arms: the holder arm grasps and presents the socket as a movable fixture, while the mover arm grasps and inserts the peg.

Isaac Forge tasks. The three tasks use the Isaac Lab FORGE environments (Mittal et al., 2025; Noseworthy et al., 2025): `Isaac-Forge-PegInsert-Direct-v0`, `Isaac-Forge-NutThread-Direct-v0`, and `Isaac-Forge-GearMesh-Direct-v0`. These environments extend the Factory assembly environments (Narang et al., 2022) with force sensing, observation noise, and domain randomization. All three tasks begin with the manipulated part in the gripper. The exported scene contains the initial held and fixed poses, assembly frames, asset dimensions, and a base-camera RGB/depth observation. Final asset poses

Table 3: Simulation environments, controlled interactions, and evaluation signals for the eight tasks.

Task	Simulator and initial state	Controlled interaction	Execution evidence and task evaluator
MS Peg	ManiSkill; peg on table	Grasp the peg, approach the box, and insert the orange end through the side opening.	Commanded/achieved TCP poses and simulated peg–box wrist wrench. Evaluator: ManiSkill `env.evaluate()` insertion predicate.
MS Plug	ManiSkill; charger on table	Grasp the charger and insert both prongs into the receptacle slots.	Commanded/achieved TCP poses and charger–receptacle wrench. Evaluator: `env.evaluate()`; target tolerance 5 mm and 0.2 rad.
FG Peg	Isaac Forge; peg held	Align the peg and perform guarded axial insertion with bounded lateral recovery.	TCP tracking, wrist force, and phase-specific progress. Evaluator: `_get_curr_successes` with the task threshold and no rotation check.
FG Nut	Isaac Forge; nut held	Align the nut, search for thread engagement, and follow a commanded helical feed.	TCP motion, axial progress, wrench history, stalls, and phase completion. Evaluator: `_get_curr_successes` with rotation check.
FG Gear	Isaac Forge; gear held	Place the gear on the medium shaft and search angular phase until the teeth engage.	TCP tracking, axial progress, wrist force, stalls, and phase-search history. Evaluator: `_get_curr_successes` with no rotation check.
MS Bimanual	ManiSkill-based (ours); two arms and a movable socket	The holder presents the grasped socket as a moving fixture while the mover inserts the peg.	Both TCP traces, peg–socket wrench, holder deflection, and capture probe. Evaluator: the unchanged PegInsertionSide insertion predicate.
UV HDMI	UniVTAC; connector held after setup	Estimate in-hand offset from both gel pads, then execute guarded insertion and lateral search.	Dual-pad depth, contact-area/centroid features, and TCP residuals. Evaluator: `task.check_success()`.
UV Key	UniVTAC; key held after setup	Rotate until dual-pad shear release, back off, and extract the key.	Dual-pad marker displacement and contact area, followed by segmented pull execution. Evaluator: `task.check_success()`.

Note. "Held" indicates that benchmark setup places the manipulated object in the gripper before Contact Code begins. TCP denotes Tool Center Point. The task evaluator supplies only the binary rollout outcome; its internal poses and metric values are not exposed to proposal, execution, diagnosis, or repair.

and success labels remain held out. The rollout backend uses the task's native direct-control interface and records the Forge force signal together with commanded and achieved motion. `Isaac-Forge-PegInsert-Direct-v0` and `Isaac-Forge-GearMesh-Direct-v0` use the official translational success test; `Isaac-Forge-NutThread-Direct-v0` also enables the official rotation check.

UniVTAC tasks. We evaluate two UniVTAC tasks (Chen et al., 2026a): `UniVTAC-InsertHDMI-v0` and `UniVTAC-PullOutKey-v0`. Both tasks begin after UniVTAC's scripted setup has grasped the manipulated object and moved it to the interaction region. We treat this setup as part of the task initial state, analogous to the already-held Isaac Forge tasks. During execution, the controller does not access the live connector or key pose. `UniVTAC-InsertHDMI-v0` uses the two gel depth maps, whereas `UniVTAC-PullOutKey-v0` uses gel marker displacement and depth. Both tasks use UniVTAC's upstream `task.check_success()` implementation as the official success evaluator.

Official task-success predicates. The following predicates are computed from evaluator-only simulator state and used to terminate and score a physical rollout. Their object poses and intermediate metric values are not available to proposal, execution control, diagnosis, or repair.

MS Peg. Let (x, y, z) be the peg-head position expressed in the hole frame and let r_h be the hole half-width. ManiSkill reports success when $x \geq -15$ mm, $|y| \leq r_h$, and $|z| \leq r_h$, where r_h equals the sampled peg half-width plus 3 mm of clearance. The final predicate does not add a grasp or orientation requirement.

MS Plug. The charger pose must be within 5 mm Euclidean translation and 0.2 rad rotation of the receptacle-defined goal pose.

MS Bimanual. The task inherits the MS Peg predicate exactly, but evaluates the peg head in the current frame of the movable socket held by the second arm.

FG Peg. The held peg base must be within 2.5 mm in the horizontal plane of its target base pose, and its vertical displacement from that target must be below 1.0 mm, obtained from the 25 mm socket height multiplied by the official success fraction 0.04. No rotation check is applied.

FG Gear. The medium-gear base must be within 2.5 mm in the horizontal plane of the target shaft pose, and its vertical displacement must be below 1.0 mm, obtained from the 20 mm gear-base height multiplied by the success fraction 0.05. The official predicate does not impose a terminal angular-phase check.

FG Nut. The nut base must be within 2.5 mm in the horizontal plane of the target bolt pose, and its vertical displacement from that target must be below 0.75 mm, obtained from the 2 mm thread pitch multiplied by the success fraction 0.375. In addition, the wrapped end-effector yaw must be below the configured success yaw of 0 rad.

UV HDMI. In the upstream evaluator, the connector pose relative to the target must satisfy a 5 mm bound on the checked lateral coordinate and have relative height below 5 mm; the end-effector world height must exceed 145 mm; and the connector's local vertical axis must have dot product greater than 0.965 with the world vertical axis (approximately $15°$). The upstream position expression uses `p[1:2]` and therefore does not include the other lateral coordinate. We record a corrected two-coordinate predicate as a held-out audit metric, but do not use it for control, repair, or the success rates in the main paper.

UV Key. The key height must exceed 90 mm and its local vertical axis must have dot product greater than 0.965 with the world vertical axis. The lock's local x axis must retain dot product greater than 0.99 with its initial direction, and the absolute vertical separation between the end effector and key must remain below 140 mm.

Iteration and rollout accounting. Contact Code permits at most four program iterations, including the initial program, and at most three physical rollouts per instance. A rejected proposal field is dropped or completed from the task-specific fallback before execution. An invalid repair patch is rejected atomically without changing the TaskSpec; within one repair step, the system makes up to five model-response attempts to obtain a valid patch. Neither an invalid patch nor a rejected static precheck consumes a physical rollout. Execution terminates after the first official success. Repair-enabled controlled comparisons use the same four-iteration and three-rollout limits; the single-turn and GenCHiP protocols are described in Sections F and D, respectively.

C Contact Code Implementation Details

C.1 Task Contracts, TaskSpecs, and Primitive Graphs

Benchmark boundary. For each task, the contract defines the legal TaskSpec fields and a fixed primitive graph. The adapter supplies executable grounding mappings, motion candidates, and controller bindings. The VLM selects only contract-exposed TaskSpec fields, including vocabulary-constrained grounding, a success criterion, and execution parameters; it does not discover arbitrary targets or generate a new primitive sequence.

TaskSpec representation. Each proposal is a typed JSON `TaskSpec` containing the target relation, contract-exposed primitive parameters, and a named verification predicate. The same field types and numerical ranges validate both the initial proposal and later repairs. The official binary outcome is computed independently by the task evaluator; raw model-generated code is never executed.

Table 4: Primitive-graph templates and principal contract-exposed repair fields for the evaluated tasks.

Task	Fixed contact stages	Principal repair fields
MS Peg, MS Plug	Acquire/maintain grasp, axis align, entry localize, guarded insert, verify	Grasp choice and offsets; approach offset; search radii, ordering, and lateral offset; feed segments and jam threshold
FG Peg	Maintain grasp, axis align, lead in, entry localize, guarded insert, verify	Search center and radii; search order; feed segments, jam threshold, terminal settling, and guarded extension
FG Nut	Maintain grasp, axis align, lead in, thread catch, helical follow, verify	Catch rotation, reverse/preload schedule, lateral search; thread direction, turns, rotation step, and feed ratio
FG Gear	Maintain grasp, axis align, contact acquire, phase engage, axial seat, verify	Lateral search radius and points; descent step and stall budget; retry backoff; angular range and phase step
MS Bimanual	Holder and mover grasp branches, axis align, entry localize, guarded insert, verify	Holder grasp and presentation parameters; mover grasp; approach standoff; insertion search, segments, and jam threshold
UV HDMI	Maintain grasp, axis align, tactile localize, entry localize, guarded insert, verify	Tactile centroid gain and contact threshold; search radii and retreat; segmented feed, slip thresholds, and jam threshold
UV Key	Maintain grasp, tactile baseline, unlock search, guarded extract, verify	Minimum contact area; rotation step and range; shear-release and pad-asymmetry thresholds; backoff and segmented pull

Note. Every template ends in an evaluator-backed verification stage, which owns no repair fields.

Fixed primitive-graph templates. The validated TaskSpec instantiates the task's fixed graph. Each node owns its execution binding, evidence, and repairable fields, which limits a repair to the contact stage implicated by the rollout.

Concrete contract example. For FG Peg, the fixed graph and field ownership are:

FG Peg contract excerpt

```
{
  "task": "FG Peg",
  "fixed_graph": [
    "maintain_grasp", "axis_align", "lead_in",
    "entry_localize", "guarded_insert", "verify"
  ],
  "field_ownership": {
    "entry_localize": [
      "search_radii_m", "search_angles", "search_order"
    ],
    "guarded_insert": ["segments", "jam_threshold_m"],
    "verify": []
  },
  "adapter_supplies": [
    "target frames", "motion candidates", "controller bindings"
  ],
  "vlm_selects": "values for contract-exposed fields only"
}
```

The graph validator checks node uniqueness, valid acyclic dependencies, a terminal verification node, and unambiguous ownership of every repairable field. As illustrated above, verification owns no repair fields.

C.2 VLM Proposal Interface and Examples

Before the first rollout, Qwen3-VL-Plus (Bai et al., 2025) receives the task instruction, the planning-visible scene estimate, the task contract and grounding vocabulary, and the initial sensor-image bundle described in Section B. At temperature 0.1, it returns one JSON proposal containing vocabulary-constrained grounding, a verification predicate, and contract-exposed execution parameters that can be selected from the initial observation. It receives no rollout evidence at this stage. Parameters that are not planning-visible at reset are omitted from the proposal and filled from the task-specific fallback TaskSpec; some may become repairable after execution produces the required evidence.

Figure 6 shows representative validated proposals from three held-out tasks. The examples illustrate that the proposal is not a free-form motion program: the task contract fixes the available fields, the adapter fixes their executable mapping, and the VLM supplies values within that interface.

C.3 Validation, Prechecks, and Program Compilation

Proposal normalization. The proposal is normalized field by field before it can affect execution. Part and region names must belong to the legal vocabulary defined by the task contract, and primitive parameters must satisfy their type, range, and task-applicability contracts. Unknown fields, malformed values, out-of-range values, and parameters not consumed by the current executor are rejected and recorded in the normalization report; numerical values are not silently clipped. Accepted fields are combined with deterministic task-specific defaults. If no valid proposal remains, the complete deterministic fallback is used.

Candidate resolution and static prechecks. The task adapter derives geometric candidates from the initial scene estimate and the normalized `TaskSpec`. Depending on the task, these include predeclared grasp alternatives, aligned and offset approach candidates, insertion search schedules, threading schedules, or gear-phase strategies. Candidates marked unreachable, colliding, or incompatible with the gripper are removed. The remaining alternatives are ranked using the quantities available for that task, including alignment, centering, motion cost, and accepted part or grasp preferences.

The selected candidate then passes through task-specific static prechecks. These checks test quantities available before a rollout, including axis and centering consistency, bounded search motion, total feed depth, thread-phase and episode budgets, feed-ratio bounds, gear-phase coverage, and tactile search or extraction budgets. For ManiSkill insertion, non-blocking advisories also record search coverage, tracking noise, and impact budget. A hard violation rejects the candidate before execution and reports the measured value and violated bound. Advisories are retained as evidence but do not block execution. Passing a precheck therefore establishes only that the proposed parameters are internally consistent with the task contract; it does not predict physical success.

Concrete precheck example. Figure 7 shows a held-out MS Bimanual instance in which a typed repair passes field validation but exceeds the search region permitted by the current socket geometry. The static precheck reports the measured quantity and bound, rejects execution without consuming a rollout, and supplies a specific constraint for the next repair.

Compilation. The compiler combines the normalized `TaskSpec`, selected motion candidate, verification predicate, and instantiated graph into an executable policy. System bindings defined by the task contract map each node's validated parameters to the implemented task-specific controller and associate its measurements with the corresponding contact stage. The current implementation does not interpret arbitrary model-generated graph topology as controller code. After every accepted repair, the complete `TaskSpec` is validated again, the fixed primitive-graph template is reinstantiated, and the policy is recompiled before the next rollout.

C.4 Execution Evidence and Stage-Local Repair

Stage-aligned execution evidence. The backend aligns controller-visible evidence with the graph node that produced it. Depending on the task, this includes TCP motion, force/torque, axial progress, controller events, search history, or tactile measurements. Evaluator-only poses and metrics are

MS Peg

```
{
  "grounding": {
    "part": "peg.orange_head",
    "region": "box_with_hole.hole",
    "grasp": "peg.white_tail"
  },
  "stages": {
    "entry_localize": {
      "search_radii_m": [0.005, 0.010],
      "search_angles": 8
    },
    "guarded_insert": {
      "segments": 6,
      "jam_threshold_m": 0.002
    }
  },
  "verification": "inserted"
}
```

FG Gear

```
{
  "grounding": {
    "part": "gear.teeth",
    "region": "gear_base.shaft_entry"
  },
  "stages": {
    "contact_acquire": {
      "strategy": "gentle",
      "descent_step_m": 0.001,
      "lateral_radius_m": 0.002
    },
    "phase_engage": {
      "phase_step_rad": 0.010,
      "phase_half_range_rad": 1.0
    }
  },
  "verification": "meshed"
}
```

UV HDMI

```
{
  "grounding": {
    "part": "connector.tip",
    "region": "socket.opening"
  },
  "stages": {
    "entry_localize": {
      "search_radii_m": [0.001, 0.002, 0.0035]
    }
  },
  "verification": "inserted"
}
```

Figure 6: Examples of initial VLM proposals. Only selected contract-exposed fields are shown.

excluded. Bounded recovery inside a primitive remains part of the current rollout; repair begins only after an unsuccessful rollout or a hard precheck rejection.

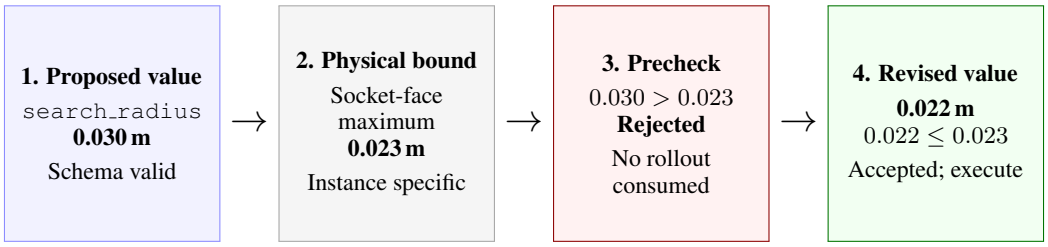


Figure 7: Semantic view of an instance-specific static precheck. A typed value can pass schema validation yet violate the physical bound derived from the current socket geometry. Rejection is explicit and consumes no physical rollout; the revised value is accepted before execution resumes.

Repair base and intervention scope. The repair may inherit parameters from the best previous execution while using the newest failure evidence for diagnosis. The scheduler selects one unresolved node on the active dependency path and exposes only fields owned by that node. Completed upstream stages and unrelated branches remain fixed. If no typed intervention exists at the observable frontier, repair stops.

Atomic patch application. The repair model returns a node-local typed patch. Field ownership, atomic groups, numerical contracts, and task-specific bounds are checked before application. Rejection changes nothing and consumes no rollout. An accepted patch inherits every unmodified field, after which the complete `TaskSpec` is revalidated and recompiled.

Concrete TaskSpec and repair example. Figure 8 follows one held-out FG Peg instance from its validated initial specification through measured failure evidence and an accepted repair. The example shows how the evidence selects one graph node and changes only a field owned by that node.

1. Validated initial TaskSpec

```
{
  "task": "FG Peg",
  "stages": {
    "entry_localize": {
      "search_radii_m": [0.001, 0.002],
      "search_angles": 4,
      "search_order": "up_first"
    },
    "guarded_insert": {
      "segments": 8,
      "jam_threshold_m": 0.002
    }
  },
  "verification": "inserted"
}
```

2. Stage-aligned evidence after rollout 1

```
{
  "stage": "guarded_insert",
  "failure": "axial_stall_under_contact",
  "axial_advance_m": 0.00435,
  "target_stroke_m": 0.02894,
  "peak_force_n": 8.548,
  "force_limit_n": 7.500
}
```

⇓

3. Accepted typed patch and outcome

```
{
  "stage": "guarded_insert",
  "field": "segments",
  "from": 8,
  "to": 5,
  "validation": "accepted",
  "next_rollout": "success"
}
```

Figure 8: TaskSpec, execution evidence, and accepted repair for a held-out FG Peg instance. The repair changes the guarded-insertion segmentation from eight to five steps, after which the next rollout succeeds. Only fields relevant to this stage-local repair are shown.

D COMPARISON METHOD REIMPLEMENTATIONS

Table 5 exposes the protocol choices that most directly affect interpretation of the comparison results. The rows share the same VLM family, but differ in feedback, episode state, editable program space, and controller API.

Table 5: Protocol comparison for Contact Code and the two independent method reimplementations. The table separates model feedback and editing scope from the low-level controller made available to each method.

Method	VLM	Feedback after failure	F/T or tactile	Episode state	Editable space	Attempts / budget	Controller API
Contact Code	Qwen3-VL-Plus; T=0.1	Stage-aligned motion/contact evidence and binary outcome	Yes; stage aligned	Reset to same seed	One typed patch at one contract-owned stage	$\leq$ 4 versions; $\leq$ 3 rollouts; $\leq$ 5 responses per repair	Fixed primitive graph with task-specific contact controller
CaP-X M4-style	Qwen3-VL-Plus; T=1.0	Previous code, stdout/stderr, outcome, and VDM description	No	Persistent episode	Full Python-program rewrite	$\leq$ 4 versions; $\leq$ 3 executions; one candidate per turn	Pose queries, Cartesian position motion, and gripper commands
GenCHiP	Qwen3-VL-Plus; T=0.0	No iterative repair feedback	No raw signal	No persistent repair state	Whole program per candidate; no repair	Five completions with execution-based selection	Compliant Cartesian motion and gripper commands

Note. The F/T-or-tactile column records whether raw contact observations are exposed to the generated program or repair model. GenCHiP uses a compliant controller, but the reimplementation does not expose raw wrist force/torque or fingertip tactile signals to the model.

D.1 TASK-ADAPTED CAP-X M4-STYLE

Our task-adapted CaP-X M4-style reimplementation (Fu et al., 2026) exposes a low-level Python API for object-pose queries, Cartesian pose motion, TCP queries, and gripper commands. The generated program has no wrist force/torque or fingertip tactile input and acts through position commands. Its prompt contains the task instruction, API documentation, and worked examples. We use Qwen3-VL-Plus at temperature 1.0 and generate one program candidate per turn.

Each held-out seed runs as one persistent episode. After a failed attempt, the model fully rewrites the program while receiving the previous program, standard output and error, the binary task outcome, and a textual description of the difference between the initial and current camera frames from the Visual Differencing Module. Non-privileged camera-based object estimates are refreshed between turns for the three Forge tasks, ManiSkill Peg, Bimanual Insertion, and both UniVTAC tasks; ManiSkill Plug uses its initial observation because the corresponding live-camera adapter is not implemented. The method is limited to four code versions and three execution attempts, with the total step allowance derived from each task's native episode length. We do not add cross-seed skill-library accumulation or a multi-model ensemble, because either would change the held-out evaluation unit or move beyond the M4-style condition evaluated here.

D.2 GENCHIP

The GenCHiP reimplementation (Burns et al., 2024) is evaluated only on the three Isaac Forge tasks, whose backend provides the task-space impedance control required by its compliant Cartesian action space. We use the same Qwen3-VL-Plus model as the other rows while following the published generation operating point: five completions at temperature 0.0, with the completion selected through execution. The prompt exposes compliant Cartesian motion and gripper actions over the initial scene estimate. GenCHiP does not perform iterative program repair. We report one outcome for each of the 100 distinct held-out seeds, rather than ten repeated executions of a single task instance. ManiSkill and UniVTAC are marked unsupported because their evaluated backends do not expose an equivalent compliant action mode.

D.3 Scope and Interpretation

Both comparison rows are reimplementations on our task interfaces, not the scores reported in the original papers. We keep the VLM fixed to Qwen3-VL-Plus so that changes in representation, feedback, and action API are not confounded with a different model. All prompts and interface choices are fixed on development seeds disjoint from seeds 100–199. A generated program that raises an exception, exceeds the common execution deadline, or produces no successful official evaluation counts as a method failure; infrastructure failures are rerun rather than scored.

Zero success in a reported row is not interpreted as task impossibility. Task-specific control programs restricted to the same pose-level commands achieve non-zero success on ManiSkill Plug and Bimanual Insertion, confirming that those action interfaces admit successful solutions. The comparison table therefore measures the complete reimplemented method under the documented prompt, feedback, and budget, rather than an upper bound on what could be hand-programmed through its API.

E Real-Robot Evaluation Details

E.1 Experimental Setup

Figure 9 shows the setup used for the physical peg-insertion evaluation.

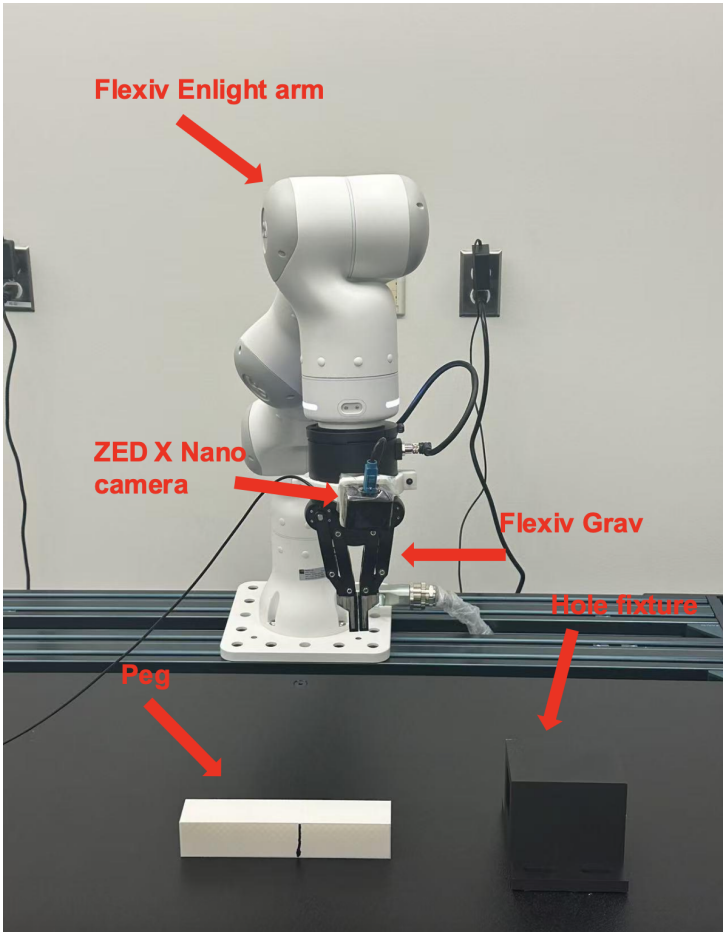
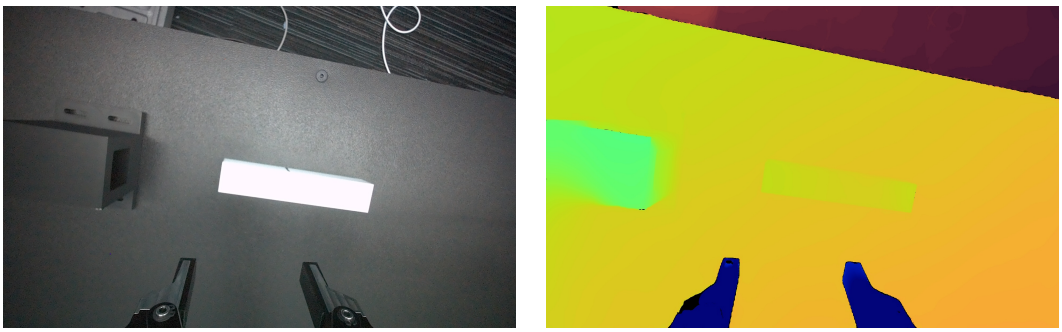


Figure 9: Real-robot experimental setup. The system consists of a Flexiv Enlight arm, a Flexiv Grav two-finger force-controlled gripper, a ZED X Nano RGB-D camera, and the peg and hole fixture used for insertion.

Trial initialization and reset. All 25 physical trials use the same nominal initial configuration; the initial peg and receptacle poses are not deliberately varied across trials. Before every physical rollout, including a repaired rollout within the same trial, the experimenter manually returns the scene to this nominal configuration. This reset is the only manual intervention: no manual correction is applied after a rollout begins or during failure diagnosis and repair. Small uncontrolled placement differences may remain after manual reset, but the evaluation does not impose a designed initial-pose sweep.

Initial visual observation. At the beginning of each trial, the ZED X Nano captures one RGB image and its corresponding depth image. This RGB-D pair is provided to the VLM for the initial TaskSpec proposal. The observation remains associated with the trial, but the camera input is not recaptured or refreshed after a failed rollout; subsequent repair uses the recorded TCP motion and force/torque evidence. Registered metric geometry and robot-specific calibration are available only to the hardware executor. The executor realizes the primitive graph through Cartesian motion and impedance control and records TCP motion and the robot-reported six-axis wrench as execution evidence. Figure 10 shows a representative input pair.



(a) RGB image (b) Colorized depth visualization

Figure 10: Representative initial RGB-D observation provided to the VLM. One pair is captured after the manual reset at the beginning of each physical trial and is not refreshed during repair.

Repair budget. The physical experiments use the same budget as the simulation experiments: at most four program iterations, including the initial program, and three physical rollouts per trial. At each repair step, the system may make up to five attempts to obtain a valid patch. Invalid patches and rejected static prechecks do not consume a physical rollout. After an accepted repair, the scene is manually reset before the next physical execution.

Success criterion. A physical rollout is counted as successful when at least 90 mm of the peg has entered the receptacle, as indicated by the mark passing the receptacle entry plane.

E.2 Representative Physical Repairs

Among the five trials that fail on the first rollout, four are recovered within the three-rollout budget. The recovered executions exhibit three distinct repair patterns. Figure 11 shows one representative recovery from the failed initial rollout to the repaired execution.



Figure 11: Representative real-robot repair. The left frame shows the failed initial rollout; the remaining frames show the repaired rollout as the robot searches, aligns the peg with the opening, and completes insertion.

1. **The peg enters the hole but does not reach the required depth.** The controller successfully finds the opening and inserts the peg by approximately 80 mm. However, the physical 90 mm success mark does not pass the fixture entry plane, so the rollout is counted as a failure. The repair enlarges the lateral search range from 3.0 to 4.5 mm, allowing the next rollout to enter the hole with better alignment and reach the required depth.

2. **The opening lies outside the initial search range.** The peg repeatedly contacts the fixture face while searching in two perpendicular lateral directions, but it does not encounter the opening within the original 3.0 mm range. The repair expands the maximum search range to 4.5 mm. This wider search covers the residual alignment error, and the next rollout succeeds.

3. **Residual misalignment causes an insertion jam.** The controller finds the opening, but residual misalignment produces hard contact and the peg jams during insertion. Motion and force evidence identify an insertion-stage jam. The repair improves contact-guided alignment before axial insertion, reducing the offset and allowing insertion to reach the required depth.

F Controlled Program Comparisons

Figure 12 summarizes the interface, feedback, and repair-scope differences evaluated in this section, while Table 6 reports the single-turn and contract-removal comparisons. Every condition uses Qwen3-VL-Plus, the same held-out seeds, task-specific controller, and task-specific evaluator as Contact Code. The task-specific primitive sequence remains fixed in all rows, so these comparisons isolate the representation and repair interface rather than the availability of a different low-level controller.

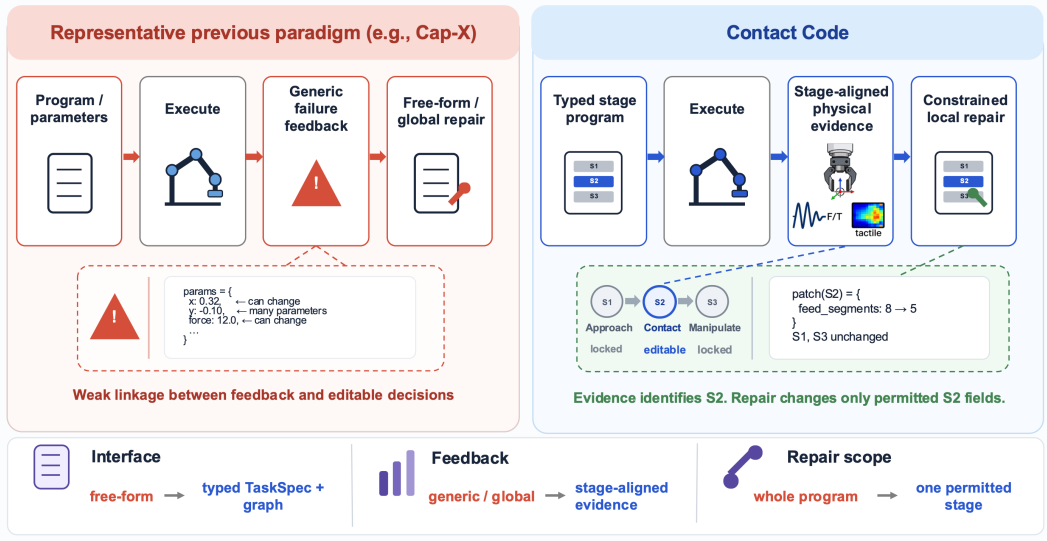


Figure 12: **Comparison of program and repair interfaces.** A representative free-form paradigm operates over a complete program or parameter set, uses global execution feedback, and may revise the full program. Contact Code instead uses a typed stage program, converts physical measurements into stage-aligned execution evidence, and restricts each repair to the permitted fields of the identified stage. Table 6 evaluates these distinctions under controlled conditions.

Table 6: Each entry is Contact Code minus the comparison condition in final-success percentage points, so $+X$ means that Contact Code succeeds by X points more. Across all 8 tasks, Contact Code gains 8.3 points over free-form parameter repair under the same evidence and rollout budget, with larger margins for single-turn or contract-free conditions, supporting the benefits of typed program constraints and evidence-grounded, stage-local repair.

Condition	MS Peg	MS Plug	FG Peg	FG Nut	FG Gear	MS Bimanual	UV HDMI	UV Key	Mean (8 tasks)
Single-turn code generation	+10	+4	+9	+36	+19	+42	+68	+24	**+26.5**
Free-form parameter repair, same evidence feedback	-2	-2	0	+19	+12	+17	+18	+4	**+8.3**
Free-form parameter repair + CaP-X M4-style feedback Fu et al. (2026)	+4	-2	+6	+31	+18	+22	+40	-1	**+14.8**
No parameter contract, single turn	+11	+1	+7	+37	+21	+39	+74	+9	**+24.9**
No contract, single turn	+14	+6	+7	+40	+23	+38	+69	+20	**+27.1**

Single-turn code generation. The model receives the task instruction, scene observation, API documentation, and worked examples, and generates one complete Python program at temperature 0.1. The program emits parameters for the fixed task-specific primitive sequence and is executed once. It receives no rollout feedback and has no repair turn.

Free-form parameter repair. This condition rewrites the complete Python program after each failed rollout, but the program can only emit parameters for the same fixed primitive sequence used by Contact Code. It is therefore a free-form parameter-repair interface, not unrestricted program synthesis. The same-feedback row receives the measured stage and contact evidence available to Contact Code and uses the same budget of four program versions and three rollouts. The M4-style feedback row changes only the feedback channel to the previous code, standard output and error, binary task outcome, and a textual visual-difference description. It is a feedback-controlled ablation with reset rollouts, not the independent persistent-episode CaP-X M4-style reimplementation in Section D.

Contract-removal conditions. *No parameter contract* uses the single-turn code prompt but disables the typed parameter schema and static precheck, allowing emitted numerical values to reach the executor without contract rejection. The evaluated task and its primitive sequence remain fixed. *No contract* additionally removes typed validation of the remaining model-generated fields, including the grasp part. The executor still uses the fixed task-specific primitive sequence and returns a failure for an incompatible output. Program exceptions and unsuccessful executions are also counted as failures. These rows test the protection supplied by the interface boundary; they do not constitute reimplementations of a published method.

G Qualitative Repair Cases and Failure Analysis

Table 7 gives representative records from the held-out evaluation logs. Each successful example starts from a failed physical rollout, applies one accepted node-local patch, resets to the same seeded initial state, and succeeds on the next rollout. The final two examples show that bounded repair can also stop without inventing an unsupported action or can exhaust its rollout budget after legal changes fail to resolve contact.

Table 7: Representative stage-local repairs from held-out evaluation instances.

Task	First-rollout evidence	Accepted intervention	Outcome
FG Peg	Axial stall under contact	`guarded_insert`: set feed `segments` to 5	Success on rollout 2
FG Nut	Insufficient realized turns	`helical_follow`: set `rotation_step_rad` to 0.07	Success on rollout 2
MS Bimanual	Motion-planning failure at axis alignment	`axis_align`: set `approach_standoff_m` to 0.08	Success on rollout 2
UV HDMI	Terminal verification failed	`tactile_localize`: set `centroid_gain` to 1	Success on rollout 2
UV Key	Terminal verification failed	`unlock_search`: set `unlock_shear_drop_px` to 3	Success on rollout 2
FG Peg	Pre-contact alignment did not converge	No typed intervention exists at the observable causal frontier	Stops after rollout 1
MS Peg	Insertion jam after bounded search	Expand search radii to $[0.005, 0.012]$ m, then set `jam_threshold_m` to 0.008	Fails after 3 rollouts

Note. Parameter values are reproduced from the held-out evaluation logs.

These cases illustrate the two roles of the contract. It translates measured failure evidence into a small intervention set when a supported repair exists, and it makes the boundary explicit when the observed failure cannot be changed through the currently exposed fields. An accepted repair is therefore a validated change, not a guarantee of eventual success.

G.1 Remaining Failure Categories

The 800-instance evaluation contains 211 final failures, of which 118 can be categorized directly from the stored stop reasons and rollout records. The remaining cases do not admit a unique failure attribution from the recorded evidence. Four stop because no typed intervention exists for the observable frontier, and 10 exhaust the repair-patch attempt budget without obtaining an accepted update. Six terminate with a motion-planning failure, and 11 with a missed or lost grasp. In 87 further cases, at least one parameter update is accepted, but every subsequent rollout fails; we categorize these cases as budget exhausted.

H Limitations and Scope

Task-specific contact knowledge. Contact Code does not remove the need to design contact skills. Each evaluated task uses a fixed primitive-graph template, task contract, evidence adapter, and controller. Adding a new contact mode currently requires defining these components and identifying parameters that produce meaningful physical interventions. The shared contribution is the typed interface that connects them to proposal, evidence, and repair; the present system does not synthesize new primitive dynamics or arbitrary graph topology.

Grounding and perception. In our evaluation, the task adapter determines the executable grounding and primitive sequence for every method. Consequently, the simulation study does not test open-vocabulary grounding or long-horizon skill sequencing. It also uses simulator-exported initial geometry rather than estimates from a learned perception model. Beyond this initial geometric

estimate, runtime diagnosis and repair use only task-available proprioceptive, force/torque, or tactile evidence plus the official binary outcome.

Repair coverage Repair is restricted to the fields exposed by the current contract. It may stop when no typed intervention exists, and legal changes may still fail within the four-iteration and three-rollout budget. These restrictions make changes inspectable but do not constitute a general safety or success guarantee.