Two things that can *by themselves* give you 0 success after 50M steps (before rewards)

Even though you asked about rewards, I need to call these out because “**success = 0 after 50 million steps**” is often *not* a shaping problem—it’s a **signal plumbing** or **task feasibility** problem.

1) “Single wrist camera” + many parallel envs can silently break learning

In Isaac Lab RL you usually run **N envs in parallel**. A correct vision setup must give you an image batch shaped like:

- `rgbd.shape == (num_envs, C, H, W)` (or equivalent)

If you replaced the tiled/per-env camera system with a **single** camera prim that renders only one env (or broadcasts the same image to all envs), then:

- proprio differs per env
- image is identical / wrong per env

That makes the observation **inconsistent**, and PPO will look “alive” (losses change) but **never solves**.

Sanity check (do this first):

- print `rgbd.mean(dim=(1,2,3))` across envs
  If most envs have the *same* mean → you’re not getting per-env images.

If you’re training with `num_envs > 1`, you almost always want **TiledCamera** (or a properly batched camera setup). This one issue can explain “0 success forever” regardless of reward.

2) Is cube_2 actually *identifiable* from RGBD?

If the 3 cubes are visually identical and you don’t provide an ID signal (color/marker/segmentation), then choosing “cube_2” is **partially unobservable**: there’s no way to know which is “2” from pixels. In that case success should be near 0 forever, and any shaped reward that doesn’t leak identity will struggle.

Why your current reward system is failing (the RL mechanics)

I’m going to be blunt: the structure you described is almost guaranteed to fail for visuomotor pick-and-place unless you get very lucky.

A) You have no dense “go toward cube_2” reward

You wrote:

- **reach**: +5 (**grasp detector**)

That’s not a reach reward; it’s basically “reward grasped.”
So before the first grasp ever happens, the agent sees:

- **almost no positive reward**
- a pile of penalties (time + limits + contacts + avoid-other-cubes)

In vision-based continuous control, a random policy grasping the correct cube is **extremely rare**. PPO can’t bootstrap from nothing.

Symptom: learning plateaus at “do nothing safely.”

B) You “grasp-gated” *everything* important

- lift: grasp-gated
- place: grasp-gated

So the agent must first discover grasping with essentially no guidance. That’s the hardest part.

C) Your penalties likely dominate early training

These can be fine later, but early they kill exploration:

- time penalty (depends on dt scaling, but your `-0.5` is suspiciously large)
- joint vel / effort soft limits
- undesired contacts (often accidentally penalizes *necessary* finger/cube contact if filtering is wrong)
- “avoid other cubes” (if it penalizes *proximity*, it can make approaching *any* cube bad)

D) Your success condition is a long chain of rare events

> lifted above 0.08m while grasping at least once, ends in goal rectangle, other cubes not in goal, gripper released

That’s basically: correct cube selection + successful grasp + lift high enough + transport + accurate placement + release + don’t disturb others.

With sparse success, PPO needs **dense shaping** for the earlier subskills or a curriculum.

E) Your reward conflicts with your success

If your *place shaping* is grasp-gated, the agent can learn:

- “stay grasping near goal forever” (keeps place reward)
- but success requires releasing, which might *drop* reward right before termination

If there isn’t a strong **release-when-ready** incentive, releasing never becomes optimal.

What a robust reward system must include for your exact task

You need **dense reward signals for each subskill**, and **smooth gating** between them:

1. **Approach** cube_2 (dense distance shaping)
2. **Pre-grasp alignment** (optional but helps: gripper/cube alignment)
3. **Close** when in pre-grasp region (teaches grasp attempts)
4. **Lift** to a target carry height (teaches “lift higher”)
5. **Carry** cube to goal XY (only after it’s clearly lifted—prevents pushing)
6. **Lower** only when above goal region
7. **Release gently** near table (explicit reward + anti-drop term)
8. **Don’t disturb other cubes** (penalize *moving them*, not being near them)

Reward System A (recommended): “Stage-shaped dense reward” (single policy, no curriculum required)

This is the most robust general-purpose structure for pick-and-place with PPO.

Definitions (per env)

Let:

- `p_ee`: end-effector (TCP) position (world)
- `p_c`: cube_2 position (world)
- `p_goal`: goal center (world)
- `d_reach = ||p_ee - p_c||`
- `d_goal_xy = ||(p_c.xy - p_goal.xy)||`
- `h = p_c.z - table_z` (height above table plane)
- `h_place = cube_half_height + 0.005` (target “resting on table” height)

- `h_carry = 0.12` (carry height; tweak to “lift more”)
- `grasp`: your grasp detector boolean (or heuristic)
- `open`: normalized gripper opening in [0, 1]
- `in_goal`: cube XY inside goal rectangle (boolean)
- `v_c = ||cube linear velocity||`

Smooth gates (avoid hard discontinuities)

Use gates like:

- `g_grasp = 1 if grasp else 0` (ok as boolean)
- `g_lift = clamp((h - 0.03) / 0.03, 0, 1)`
  (≈0 until cube is clearly off the table)
- `g_near_goal = exp(-d_goal_xy / 0.15)` (softly increases near goal)

These gates make the reward “turn on” gradually.

Rewards (dense, bounded, and aligned with your 4-step behavior)

1) Reach cube_2 (always active)

Give a reward that increases as you get closer.

A very stable choice is an exponential kernel:

$$r_\text{reach} = \exp(-d_\text{reach}/\sigma_\text{reach})$$

Use `σ_reach = 0.08` (8 cm) to start.

- **weight** `w_reach = +2.0`

This alone fixes your biggest bootstrapping problem.

---

2) Pregrasp “close when ready” (teaches grasp attempts)

Reward closing *only when the gripper is near the cube*, otherwise you risk “always close.”

Example:

- `near = exp(-d_reach / 0.05)` (close-range gate)
- `r_close = near * (1 - open)` (more reward when closed near cube)
- **weight** `w_close = +0.5`

This helps PPO discover grasping much sooner.

---

3) Grasp holding reward (small, not huge)

You want grasp to be “worth it,” but not the final objective.

- `r_grasp_hold = grasp` (per-step)
- **weight** `w_grasp = +0.5` (per-step)

If you can implement an **event** reward (grasp transitions 0→1), add:

- `r_grasp_event = 1 on grasp_start`
- **weight** `+5.0` (one-time)

Event reward is better than per-step, but per-step is fine.

---

4) Lift to carry height (your “lift a little bit more”)

Reward height, but only when grasped.

A bounded shaping:

rlift=clip(h/hcarry,0,1)r_\text{lift} = \text{clip}(h / h_\text{carry}, 0, 1)rlift=clip(h/hcarry,0,1)

- `r_lift = grasp * clip(h / 0.12, 0, 1)`
- **weight** `w_lift = +4.0`

Add a “don’t drag” penalty while grasped:

- `r_drag = - grasp * exp(-h / 0.02)`
  (big penalty if height is near 0 while grasped)

- **weight** `w_drag = +1.0` (since `r_drag` is negative)

---

5) Carry to goal XY (only once lifted)

Key to prevent “pushing on table”: gate by lift.

rcarry=glift·exp(−dgoal,xy/σgoal)r_\text{carry} = g_\text{lift} \cdot \exp(-d_\text{goal,xy}/\sigma_\text{goal})rcarry=glift·exp(−dgoal,xy/σgoal)

- `σ_goal = 0.15`
- `r_carry = g_lift * exp(-d_goal_xy / 0.15)`
- **weight** `w_carry = +6.0`

This gives strong dense guidance after the cube is off the table.

---

6) Lower only when near goal (prevents dropping elsewhere)

When close to goal, reward reducing height toward `h_place`.

rlower=glift·gnear_goal·exp(−|h−hplace|/σz)r_\text{lower} = g_\text{lift} \cdot g_\text{near_goal} \cdot \exp(-|h - h_\text{place}|/\sigma_z)rlower=glift·gnear_goal·exp(−|h−hplace|/σz)

- `σ_z = 0.02`
- **weight** `w_lower = +3.0`

This builds the “lower gently near the goal” behavior.

---

7) Gentle placement / anti-drop shaping

To stop “drop from high ground,” penalize releasing while cube is high, and penalize high downward velocity near placement.

(a) Penalize opening when cube is high and not in goal

- `bad_release = (open > 0.8) & (h > 0.06) & (~in_goal)`
- `r_bad_release = -1 * bad_release.float()`
- **weight** `w_bad_release = +5.0` (strong)

(b) Penalize high downward velocity near table (slam)

When in goal and within lowering phase:

- `slam = relu(-v_z - 0.2)` (threshold downward speed)
- `r_slam = - g_near_goal * relu(-v_c.z - 0.2)`
- **weight** `w_slam = +2.0`

(c) Reward being stable before release

- `stable = exp(-v_c / 0.25)`
- `r_stable = in_goal.float() * exp(-abs(h - h_place)/0.02) * stable`
- **weight** `w_stable = +2.0`

This strongly encourages “set down, settle, then release.”

---

8) Release reward (explicit!)

You need to explicitly make releasing optimal.

Define “ready-to-release”:

- `ready = in_goal & (abs(h - h_place) < 0.015) & (v_c < 0.05)`

Reward opening *when ready*:

- `r_release = ready.float() * open`
- **weight** `w_release = +4.0`

And/or an event bonus on “release happens while ready”:

- **event bonus** `+25` (recommended)

---

9) Success bonus + success definition (simplify)

Your success definition is too strict.

Recommended success condition (final-state only):

- cube_2 in goal rectangle
- cube_2 height near table (`|h - h_place| < 0.015`)
- cube_2 stable (`v_c < 0.05` and maybe angular vel small)
- gripper open (`open > 0.8`)
- optionally: held for **N consecutive steps** (e.g., 10) to avoid bounce

Success bonus: `+50` (dt-cancel if you use dt-scaling elsewhere)

Remove “was lifted above 0.08 at least once” from success.
If it ends correctly placed and released, it *necessarily* had to lift or slide; the behavior you actually care about is the final placement.

Penalties (keep them, but make them “policy-friendly”)

Time penalty

Unless you are 100% sure about your dt scaling, your `-0.5` is suspicious.

- Start with **`-0.01` per step** (or `-0.01 * dt` if you dt-scale all rewards)

Goal of time penalty is just: “don’t waste steps,” not “make the whole return negative.”

Action smoothness (usually better than joint-limit penalties early)

These are very effective for “gentle place”:

- `r_action_l2 = -||a||^2` with weight `0.001` to `0.01`
- `r_action_rate = -||a_t - a_{t-1}||^2` with weight `0.01` to `0.1`

Joint vel / effort soft limits

Keep, but small:

- joint vel: `-0.05` (not `-0.15`) initially
- effort: `-0.02` initially

“Avoid other cubes” — change what you penalize

Do NOT penalize “being close to other cubes.”
That can make the optimal strategy “never go near any cube.”

Instead penalize **moving** them:

- store `p_i_init` at reset for cube_1 and cube_3
- penalty: `sum_i relu(||p_i - p_i_init|| - 0.02)` (2 cm slack)
- weight: `-1.0` to `-3.0` (dt-scaled)

If you still need “other cubes not in goal,” make it a *small penalty*, not a success constraint:

- `r_other_in_goal = -1 * (# other cubes inside goal)`
- weight: `-5.0` event-like (or per-step small)

Reward System B: “Potential-based progress rewards” (less reward hacking, often better PPO stability)

If you can store previous-step values per env, do this. It prevents the agent from “parking” to farm dense reward.

Instead of rewarding `exp(-d)` directly, reward **improvement**:

- `r_reach_prog = (d_reach_prev - d_reach_curr)`
- `r_goal_prog = (d_goal_prev - d_goal_curr)` gated by lift
- `r_height_prog = (h_curr - h_prev)` gated by grasp

Then clamp them to avoid spikes.

This structure is extremely robust because:

- doing nothing gives ~0 progress reward
- moving away gives negative reward
- moving toward gives positive reward

You still keep the same late-stage terms (stable + release + success bonus).

If you can’t easily store prev values in Isaac Lab reward terms, stick with System A.

Reward System C: “Curriculum (most reliable with vision)”

If you want *maximum robustness* for visuomotor manipulation, curriculum beats any clever reward.

Train the **same policy** but change environment/reset distributions over time:

1. **Stage 1 (grasp only)**
    - cube_2 spawned close to gripper
    - goal ignored
    - success: grasp + lift 5 cm
2. **Stage 2 (lift + carry)**
    - cube_2 still near
    - goal closer
3. **Stage 3 (full pick and place)**

    - full randomization

You can keep System A rewards, but curriculum makes “first success” happen orders of magnitude sooner.

About “avoid link contact with table/cubes” *without* a contact sensor

You can’t perfectly penalize *contacts* without some source of contact/collision info. If you truly don’t want contact sensors, your options are:

Option 1 (best): Collision filtering (prevent contact physically)

Configure collisions so that:

- only fingertips collide with cubes
- arm links do **not** collide with table/cubes

This is robust and cheap, and you don’t need contact sensors.
Tradeoff: can allow unrealistic interpenetration for disabled links (depending on setup).

Option 2: Geometry-based “clearance” penalties (no contact data required)

Penalize being *too close* to the table plane using link poses:

- For each “forbidden” link (forearm, wrist, hand, etc.), compute its `z_link`
- penalty: `relu((table_z + margin) - z_link)`
- sum over links

This discourages scraping the table without a contact sensor.

Similarly for cubes (approximate):

- penalize if forbidden link origins are within `r_safe` of non-target cubes
- or penalize displacement of other cubes (recommended)

Important: Don’t apply this to fingertips, or you’ll fight the grasp.

Option 3: Keep contact sensor but fix filtering (honestly: usually the right move)

You already pay the big cost (RGBD rendering). Contact sensors are not the bottleneck. Most "contact penalty breaks learning" cases are just **mis-filtered** (penalizing fingertip–target cube contact).

If your real issue is "don't touch table with forearm," keep contact sensing but filter:

- allow fingertip ↔ target cube
- penalize non-fingertip ↔ table/cubes

Minimal changes to your current system that will likely unblock learning

If you don't want to rewrite everything, do these first:

1. **Replace "reach via grasp detector"** with a true distance shaping to cube_2.
2. **Remove grasp gating** on reach and (partially) on carry. Use lift-height gating instead.
3. **Add explicit release reward** (and/or penalize high release).
4. **Change avoid-other-cubes** from "proximity penalty" → "displacement penalty."
5. **Simplify success** to final placement + release + stability. Remove "lifted above 0.08 at least once."
6. **Verify camera batching** (`rgbd.shape[0] == num_envs` and images differ across envs).

If you want, I can give you a drop-in Isaac Lab reward config

Answer these and I'll translate System A into concrete reward term functions + weights in the exact style of the Franka stack task you forked:

1. `num_envs` you train with (e.g., 64/256/1024)?
2. Your action space: IK delta pose? joint targets? (sounds like IK delta pose)
3. What is your control timestep `dt` and episode length (steps)?

4. How are you detecting `grasp` right now (gripper width heuristic vs contact-based)?

With those, I can propose weights that are scaled correctly for your dt and won't get drowned by penalties.