Appendix

A. Limitations and Future Work

Within the Stochastic Interpolant framework, we adopt linear interpolation with ODE integration. While this approach allows sampling diverse trajectories through different noise initializations, we have not yet explored alternative interpolation schedules or mechanisms to explicitly control which trajectory mode is generated for ambiguous tasks.

Furthermore, we focus on quasi-static tasks in this work; temporal normalization is a deliberate choice to learn a speed-invariant geometric trace prior that transfers across embodiments. Consequently, applying our method to highly dynamic environments may pose challenges. Future research can be extended to also predict a temporal scale alongside the normalized trace, which directly recovers velocities while retaining cross-embodiment alignment.

The quality of demonstration data varies. A portion of our source videos contains inefficient or corrective motions—where operators make exploratory movements or errors before completing tasks—introducing suboptimal supervision signals. We implemented additional filtering steps to clean the dataset, though some noisy demonstrations remain.

Moreover, TraceGen's zero-shot generation ability, while promising, is not yet fully reliable under novel embodiments or unseen environments, occasionally yielding plausible but physically infeasible trajectories. Additionally, for fine-grained manipulation tasks, the generated trajectories may lack sufficient detail for the robot to execute precise manipulation actions. Scaling to internet-scale demonstration datasets, combined with improved data filtering mechanisms, could address these issues. Finally, extending beyond human-like robot arms to very different robot types would test the limits of trace-space abstraction. Despite these challenges, we believe TraceGen's efficiency and generality represent a meaningful step toward practical cross-embodiment manipulation systems.

B. Prompts for Task Instruction Generation

To obtain consistent and diverse task instructions from video segments in TraceForge, we use a vision-language model (VLM) to transform representative video frames into three complementary instruction styles. As described in the main manuscript, each event chunk is paired with (i) a concise imperative command, (ii) a stepwise manipulation instruction, and (iii) a natural, human-like request. When human-written instructions are already available, we preserve them and augment them with these additional variants. Otherwise, we sample frames from the beginning, middle, and end of the chunk and prompt the VLM to generate instructions following the structured specification below.

Sample Prompt for Instruction Generation

You are an expert image analyzer. You will receive a sequence of frames from a single video episode that records a simple manipulation task. The frames are ordered chronologically from initial state to final state.

GOAL Infer the most likely task being performed in this episode and return the OUTPUT FORMAT below.

TASK INSTRUCTIONS

- IMPORTANT: Do **NOT** generate a descriptive sentence like "The agent is trying to grab the sink faucet."
- Instead, generate instructions as if a human is directly commanding a robot or agent to make this situation happen.
- Provide exactly three instructions, one for each category:
  - `instruction_1`: Direct, simple imperative command e.g., "Turn on the faucet."
  - `instruction_2`: Step-by-step explicit manipulation e.g., "Move your gripper toward the faucet handle."
  - `instruction_3`: Natural human-like request e.g., "Can you turn on the sink for me?"
- All instructions must explicitly state **what** object is manipulated and **how**.

Instruction length rules

- `instruction_1`: $\leq$ 10 words
- `instruction_2`: $\leq$ 20 words
- `instruction_3`: $\leq$ 15 words

The operator can be a human, robot, or tool.

OUTPUT (JSON ONLY)

```
{
  "instruction_1":
      "<=10 words
      imperative command>",
  "instruction_2":
      "<=20 words
      stepwise/explicit command>",
  "instruction_3":
      "<=15 words
      natural human-like request>"
}
```

POLICIES

- Do NOT invent objects that are not visible.
- Prefer specific names if clear (banana), else generic ones (container).

- All strings must be in English.
- Return only valid JSON—no markdown or extra text.

C. 3D Trace Extraction Accuracy of TraceForge

As discussed in the main text, TraceForge provides the large-scale 3D motion supervision used to train TraceGen. To ensure that this supervision is reliable, we report a quantitative sanity check of TraceForge's 3D trace extraction accuracy.

We evaluate whether TraceForge recovers 3D trajectories that faithfully reflect real robot motion by comparing its predicted traces with ground-truth end-effector trajectories obtained via forward kinematics across nine teleoperated episodes. Since TraceForge represents motion as a 20×20 grid of point trajectories, we identify, for each episode, the predicted point whose 2D projection is closest to the end-effector in the first frame and treat its 3D path as the corresponding predicted trace.

Across episodes (13–24.5 seconds each, with an average displacement of 70.96 cm), TraceForge achieves sub–2.3 cm endpoint error on all axes (Table 3). These results indicate that the TraceForge extraction pipeline produces centimeter-level motion accuracy, providing reliable supervision for training TraceGen.

Table 3. Absolute endpoint error along the x, y, z axes in camera coordinate between predicted and ground-truth trajectories.

Error (cm)	x	y	z
Mean	1.66	1.79	2.26
Std	0.82	1.82	2.69

D. Model Training Details

This section provides comprehensive details on TraceGen's training configuration, complementing the architecture overview in the main manuscript. TraceGen employs a multi-encoder architecture with DINOv3 and SigLIP for RGB feature extraction, a depth encoder with a learnable stem adapter for geometric information, and T5 for text encoding.

D.1. Encoder Freezing Strategy

To leverage pretrained representations efficiently, we keep all encoders (DINOv3, SigLIP, T5) frozen throughout training and train only the fusion layer and decoder. This design choice follows Prismatic VLM [19], which demonstrated that finetuning visual backbones significantly degrades performance on vision-language tasks. Their analysis revealed that updating pretrained vision encoders during task-specific finetuning leads to catastrophic forgetting of the rich visual priors learned during large-scale pretraining.

By maintaining frozen encoders, TraceGen retains:

- **Visual features from DINOv3**, trained via self-supervised learning on large-scale natural images, providing strong geometric and semantic understanding
- **Vision-language alignment from SigLIP**, enabling effective conditioning on text instructions
- **Linguistic representations from T5**, capturing task semantics and manipulation goals

The trainable fusion layer and flow-based decoder learn to combine and map these frozen representations to the 3D trace prediction task. This approach substantially reduces the number of trainable parameters, accelerates training, and improves generalization to unseen manipulation scenarios.

E. Evaluations

E.1. Real world evaluation setup

We evaluate whether 3D traces predicted by TraceGen enable effective robot manipulation. Experiments are conducted on a Franka Research 3 robot across four tasks:

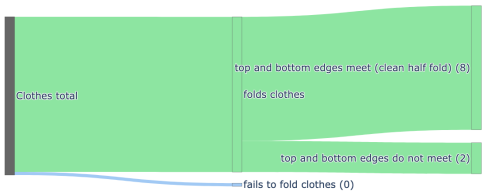
- folding a garment (Clothes)
- inserting a tennis ball into a box (Ball)
- sweeping trash into a dustpan with a brush (Brush)
- placing a LEGO block in the purple region (Block)

Given a single RGB-D observation and a text instruction, TraceGen predicts trajectories using 100-step ODE integration with classifier-free guidance (guidance scale 2) for *Brush* and *Clothes*, and without guidance for *Block* and *Ball*. To convert these scene-level traces into executable actions, we identify the subset of keypoints from the predicted 20×20 grid that correspond to the robot gripper. We then calculate the average 3D displacement $(\Delta x, \Delta y, \Delta z)$ of these gripper keypoints. After transforming this averaged translational motion from the camera frame to the robot base frame via $T_{\text{base}\leftarrow\text{cam}_{\text{ref}}}$, we apply inverse kinematics to obtain the final joint-space commands. For the gripper control, we explicitly trigger open/close actions based on successful grasp conditions. For all methods, the predicted z values are rescaled to match the measured depth maps prior to execution.

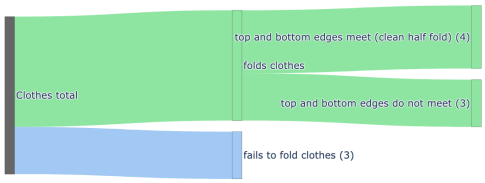
E.2. Depth rescaling

To align the predicted trace depths with the original sensor depth, we apply a depth-rescaling procedure. Similar to NovaFlow, which computes a single scaling factor based on the median depth of the initial ground-truth map, we also begin by estimating a global scaling factor between the predicted depth and the sensor depth.

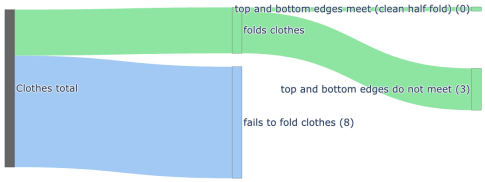
However, unlike the environments used in NovaFlow,



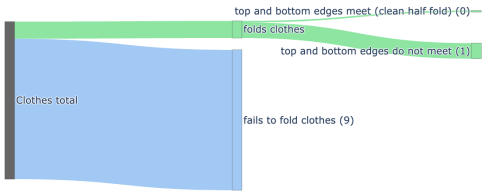
(a) TraceGen (Warmed up with robot demo)



(b) TraceGen (Warmed up with human demo)

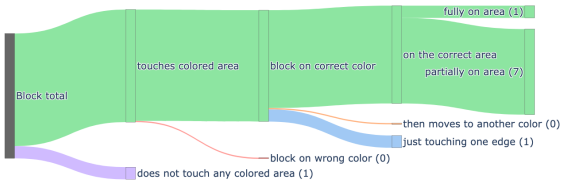


(c) NovaFlow (Veo3.1)

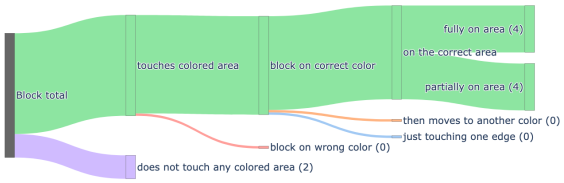


(d) NovaFlow (Wan2.2)

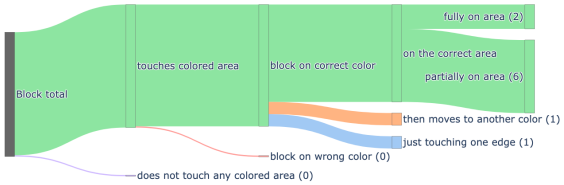
Figure 9. Failure-mode breakdown for the *Clothes* task.



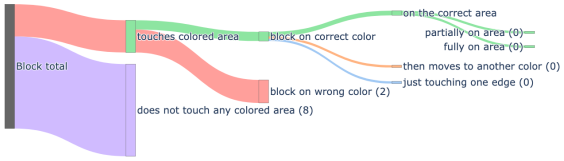
(a) TraceGen (Warmed up with robot demo)



(b) TraceGen (Warmed up with human demo)



(c) NovaFlow (Veo3.1)



(d) NovaFlow (Wan2.2)

Figure 10. Failure-mode breakdown for the *Block* task.

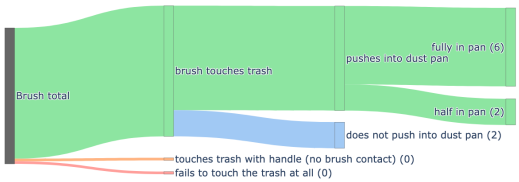
our settings exhibit much larger variations and more frequent movements along the depth axis. We observed that using a single median-based scalar often leads to substantial depth estimation errors in such scenarios.

To address this, instead of relying on a single global statistic, we compute a pixel-wise depth rescaling map by directly comparing the predicted and sensor depth maps across all pixels. We then apply a Gaussian blur to this map to obtain a smooth depth-rescaling field, and multiply this smoothed map with the predicted 3D trace to correct its z-values.

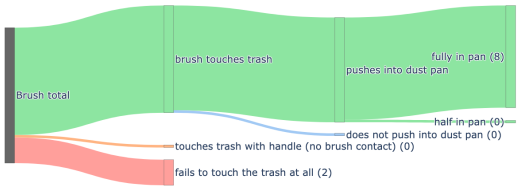
E.3. Implementation details for the baselines

3DFlowaction The official 3DFlowAction implementation relies on a filtering pipeline to extract object masks. This pipeline combines a language-conditioned object detector with a heuristic process designed to remove the robot gripper. However, when only a single image is provided as input, the detector often fails to identify the target object reliably or to filter out the robot end-effector. As a result, the predicted masks frequently include parts of the robot itself.

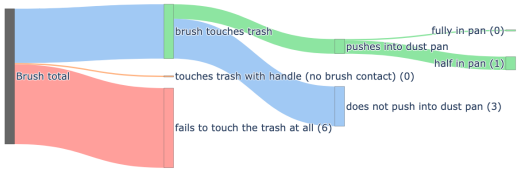
To mitigate this issue, and consistent with the official implementation, we provide a short sequence of images containing minimal robot motion. These slight temporal changes help the filtering pipeline correctly identify the robot gripper, allowing the final filtered region to closely match the ground-truth object mask. To ensure the generated object mask aligned with the expected robot movement, we manually checked whether the detected bounding box is aligned with the ground truth bounding box of the



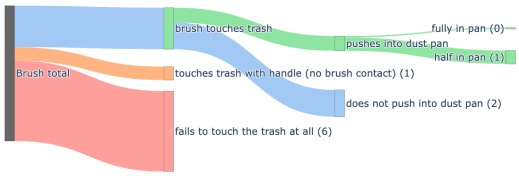
(a) TraceGen (Warmed up with robot demo)



(b) TraceGen (Warmed up with human demo)

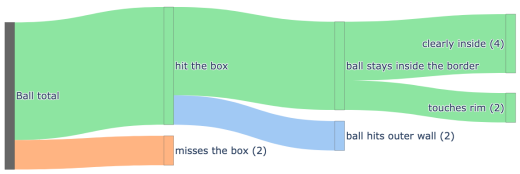


(c) NovaFlow (Veo3.1)

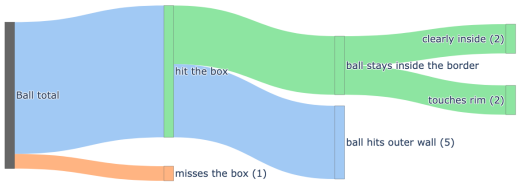


(d) NovaFlow (Wan2.2)

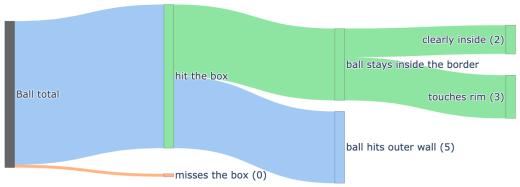
Figure 11. Failure-mode breakdown for the *Brush* task.



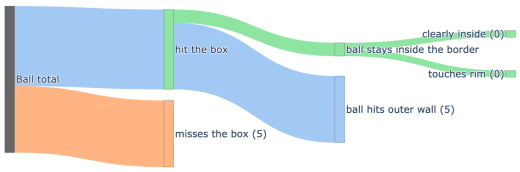
(a) TraceGen (Warmed up with robot demo)



(b) TraceGen (Warmed up with human demo)



(c) NovaFlow (Veo3.1)



(d) NovaFlow (Wan2.2)

Figure 12. Failure-mode breakdown for the *Ball* task.

target objects.

NovaFlow The original NovaFlow pipeline includes a grasp-proposal module and a trajectory-planning module. Since our tasks do not involve grasping, we remove the grasp-proposal stage and initialize the robot in a state where it is already holding the object, ensuring a fair comparison.

In addition, the official NovaFlow implementation uses MegaSaM [28] with TAPIP3D for camera pose estimation and depth prediction. For fair comparison, we replace the MegaSaM component with the same fine-tuned VGGT [42] depth and pose predictor from SpatialTrackerV2 [44] that we use in TraceForge. This VGGT-based predictor achieves similar accuracy while providing substantially faster inference by avoiding expensive 3D optimization. Also, for each NovaFlow, we use the following prompt:

- Veo3.1 (Clothes): The robot smoothly picks up the pants leg and folds the garment in half.
- Veo3.1 (Brush): The robot arm moves the broom toward the yellow trash, sweeps it forward, and guides it into the dustpan.
- Veo3.1 (Block): In the picture you see robot, blue cube, red paper, and purple paper. The blue cube is right underneath the robot arm. These are the only things that you need to pay attention. The robot arm grabs the blue cube and put it on the top of the purple paper.
- Veo3.1 (Ball): The robot arm in the image moves to grab the tennis ball and put it into the box in the image.
- Wan2.2 (Clothes): The robot's end effector grasps the black pants, positions them flat, then folds them in half

to create a compact folded shape.

- Wan2.2 (Brush): The robot's end effector grips the broom handle and sweeps the yellow trash into the dustpan with deliberate strokes.
- Wan2.2 (Block): The robot's end effector grasps the LEGO block, lifts it upward, moves it above the purple notebook, then lowers it onto the notebook center.
- Wan2.2 (Ball): The robot's end effector grasps the tennis ball, lifts it upward, then moves it horizontally toward the box and lowers it inside.

NovaFlow on Human-Hand Videos. Since NovaFlow originally utilizes generated object trajectories to transfer across embodiments, prompting it to generate robot grippers can sometimes lead to hallucination artifacts. To ensure a comprehensive evaluation in its intended regime, we also evaluated NovaFlow by generating human-hand videos and extracting object-only traces for execution. The results of this evaluation are presented in Table 4.

Table 4. NovaFlow success rate (Human-Hand Videos)

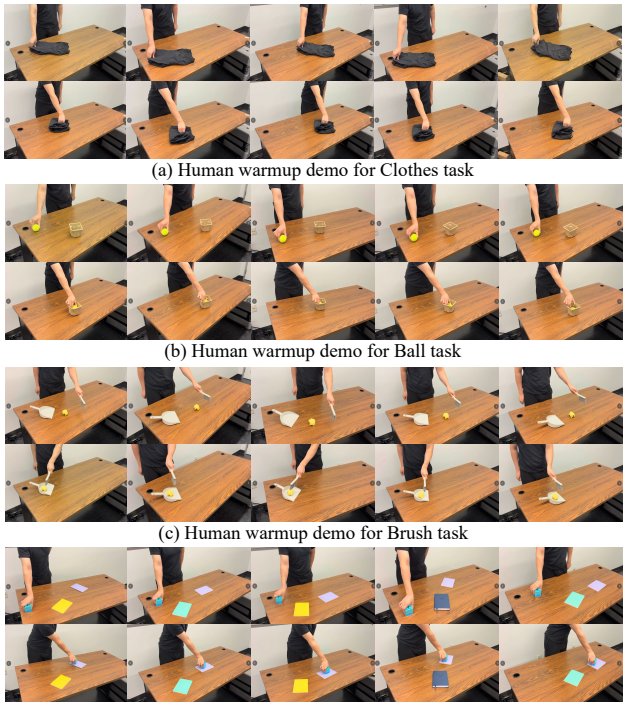
Video Model	Brush	Block	Ball	Clothes	Average
NovaFlow (Veo3.1)	0/5	1/5	4/5	2/5	35%
NovaFlow (Wan2.2)	1/5	0/5	0/5	1/5	10%

AVDC For AVDC, we follow the exact evaluation setup used in the NovaFlow paper. Specifically, we isolate the video-generation component from AVDC and combine it with the same 3D point-tracking and depth-estimation modules used in both TraceForge and NovaFlow, ensuring consistency across all baselines.

Inference latency measurement. All baseline runtimes were measured on an NVIDIA RTX A5000 except Wan 2.2 and Veo 3.1. Wan 2.2 could not fit on a single A5000, so we enabled inference using multiple GPUs—an unavoidable choice that in fact favors the baseline. Veo 3.1 is closed-source, and its latency is reported based on average API response time, which again places the baseline at an advantage.

F. Failure modes analysis

To better understand the behavior of each method beyond success rates, we provide a detailed failure-mode analysis across all four tasks and four model configurations: (i) **TraceGen** with robot-domain warmup, (ii) **TraceGen** with human video warmup, (iii) **NovaFlow** (Veo3.1), and (iv) **NovaFlow** (Wan2.2). For each combination of task and method, we collect every executed trial and categorize the outcome into fine-grained success and failure types based on object interaction quality and task completion criteria.



(a) Human warmup demo for Clothes task

(b) Human warmup demo for Ball task

(c) Human warmup demo for Brush task

(d) Human warmup demo for Block task

Figure 13. Human warmup demonstrations for all four tasks, showing first (top) and final (bottom) frames of each handheld video.

We visualize these distributions in Sankey diagrams, which reveal how trials progress from the total pool of attempts (left) into distinct outcome modes (right). This representation highlights the long-tail structure of failure patterns—showing whether errors arise early (e.g., incorrect approach) or late in the trajectory (e.g., partial completion, drift during final alignment).

F.1. Warmup data

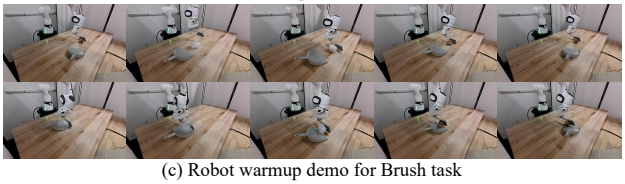
In this section, we visualize all warmup demonstrations used in our experiments. As described in the main text, TraceGen is adapted to each task using a lightweight warmup stage, which serves to translate the embodiment-agnostic 3D traces into the action space of the target robot or tasks.

We consider two warmup regimes:

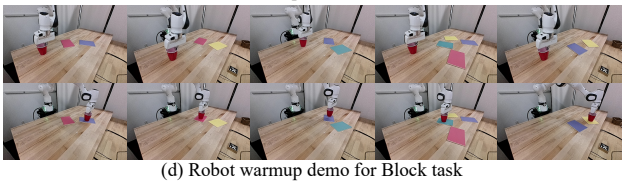
- **Robot→Robot (same-embodiment warmup).** Five in-domain robot demonstrations are provided for each task. These clips differ from the evaluation setting in object layout and initial robot pose, ensuring that warmup does not simply memorize target configurations.
- **Human→Robot (cross-embodiment warmup).** Five uncalibrated human videos (3–4 seconds each) are captured per task using a handheld phone. These clips differ substantially from the robot setting in background, light-

(a) Robot warmup demo for Clothes task

(b) Robot warmup demo for Ball task



(c) Robot warmup demo for Brush task



(d) Robot warmup demo for Block task

Figure 14. Robot warmup demonstrations for all four tasks. Top row shows the first frame of each demo; bottom row shows the final frame.

ing, embodiment, and object placement.

Each figure below shows all warmup demonstrations for the four tasks (*Clothes*, *Block*, *Brush*, *Ball*). For each demo, the **top row displays the first frame** and the **bottom row displays the final frame**.

F.2. Long horizon experiments

To assess whether TraceGen's predicted trace can be composed into longer multi-step behaviors, we evaluate the model on a long-horizon *Sorting* task. The goal is to separate blocks from white trash items: each block must be placed in a designated green region, and each trash item must be placed in the red region. We collect five human teleoperation demonstrations of the full sorting process and segment them into four primitive subtasks, which serve as warmup data for TraceGen.

The sorting procedure consists of the following four consecutive subtasks:

1. Place the left trash on the red paper.
2. Place the pink LEGO block on the green paper.
3. Place the blue LEGO block on the green paper.
4. Place the right trash on the red paper.

Completing all four subtasks in sequence constitutes a successful sorting episode.

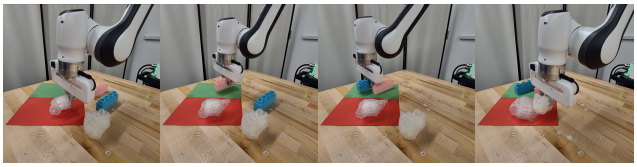


Figure 15. Visualization of the long-horizon Sorting task, showing the four sequential placement subtasks from left to right.

Use of scripted grasping. Because TraceGen models only the 3D trace component of manipulation and does not include an external grasping module, we assume access to a pre-defined scripted policy for picking up each object. This policy moves the robot to a preset grasping pose, enabling the placing skill generated by TraceGen to begin from a consistent home configuration.

Results. We compare TraceGen initialized with pretraining from the TraceForge-123k dataset ("Warmed up from TraceGen") against a *From Scratch* model trained only on the four warmup segments. Table 5 reports per-step success rates across 10 rollouts. While the pretrained model occasionally fails the first trash placement, it maintains high performance on all subsequent subtasks. In contrast, the scratch model exhibits compounding errors over time, with success rates degrading markedly in the later steps.

Table 5. Long-horizon Sorting task: per-subtask success rates (left to right indicates temporal order).

Model	Left Trash →	Pink Block →	Blue Block →	Right Trash
Warmed up from TraceGen	0.8	0.8	0.8	0.8
From Scratch	1.0	0.8	0.5	0.4

Overall, these long-horizon results show that TraceGen's pretrained motion priors enable stable composition of primitive placing behaviors, mitigating error accumulation across sequential subtasks. Although the model was not explicitly optimized for extended planning, its compact 3D trace representation supports reliable stitching of skills over longer task horizons.