Fachhochschule Nordwestschweiz
Hochschule für Informatik

P9 Mars Rover Autonomy

Autonomous manipulation and workflow automation for the FHNW Mars Rover

FHNW University of Applied Sciences and Arts Northwestern Switzerland

School of Engineering

Advisor: Prof. Dr. Christoph Stamm
Author: Sandro Covo

August 25, 2026

Acknowledgements

I want to thank the current and former members of the FHNW Rover Team, whose work on this rover and on the ones before it made this project possible.

The team would not exist without the support of the programme heads at the Schools of Engineering and Computer Science: Prof. Dr. Arne Wahlen (Mechanical Engineering), Prof. Dr. Sebastian Gaulocher (Electrical and Information Technology), and, in Computer Science, Barbara Scheuner and Christoph Denzler, who held the post before its current head, Dr. Michael Faes.

Finally, I thank my advisor, Prof. Dr. Christoph Stamm. He has supervised all my projects on the rover: from the P6 bachelor's thesis through P7 and P8 to this one. He has also backed the Rover Team since its early days.

Contents

List of Figures

List of Tables

List of Abbreviations

ACT Action Chunking Transformer

AI artificial intelligence

API application programming interface

BF16 brain floating point, 16 bit (table 3.1)

CAD computer-aided design

CI/CD continuous integration and deployment

CLI command-line interface

CPU central processing unit

DDS Data Distribution Service

DiT Diffusion Transformer

DLT Direct Linear Transform

DoF degrees of freedom

ERC European Rover Challenge

FAST Frequency-space Action Sequence Tokenization

FDCC Forward Dynamics Compliance Control

FHNW University of Applied Sciences and Arts Northwestern Switzerland

FP16 floating point, 16 bit (table 3.1)

FP32 floating point, 32 bit (table 3.1)

GPU graphics processing unit

GRU gated recurrent unit

GUI graphical user interface

HPC high-performance computing

IK inverse kinematics

INT8 integer, 8 bit (table 3.1)

IPPE Infinitesimal Plane-based Pose Estimation

JPEG Joint Photographic Experts Group, an image format

JSON JavaScript Object Notation

JSONL JSON Lines

KDL Kinematics and Dynamics Library

LLM large language model

LoRA Low-Rank Adaptation

MCP Model Context Protocol

ONNX Open Neural Network Exchange

PnP Perspective-n-Point

PPO Proximal Policy Optimization

QoS quality of service

RECAP RL with Experience and Corrections via Advantage-conditioned Policies

REST Representational State Transfer

RMS root mean square

ROS2 Robot Operating System 2

RTC real-time chunking

SLURM Simple Linux Utility for Resource Management, a cluster workload manager

SoC state of charge

SQPNP Sequential Quadratic Programming Perspective-n-Point solver

TF32 TensorFloat-32 (table 3.1)

ToF time-of-flight

TOML Tom's Obvious Minimal Language

TTL time to live

URDF Unified Robot Description Format

VLA Vision-Language-Action

VLM Vision-Language Model

YAML YAML Ain't Markup Language

Abstract

The FHNW Rover Team competes in the European Rover Challenge. One of the tasks in this challenge is operating a maintenance panel. Building on a ROS2 stack that already ran per-task imitation policies, this work extends the rover's autonomy in five areas, over a reworked manipulator control and safety layer. A single language-conditioned SmolVLA policy replaces the per-task models and is post-trained with RECAP (RL with Experience and Corrections via Advantage-conditioned Policies), a reinforcement-learning recipe that raises average success across three LIBERO suites from 62.9 % to 68.1 %, against 55.3 % for the same architecture trained conventionally. The workflow tool n8n is bridged to ROS2, so an operator composes task sequences from topics, services and actions without writing code. An on-device Qwen3.5-0.8B agent, fine-tuned on its own executions, calls tools on 93 % of 43 held-out prompts, against a cloud configuration's 92 %. TensorRT deployment and single-step distillation cut one action chunk from 1357 ms to 330 ms on the rover's Jetson Orin, leaving 21 ms of the 50 ms control cycle free. The in-gripper time-of-flight grid enters the policy's state through one configuration entry; it is cheap enough to keep without a measured gain. The classical approach phase, reimplemented in C++, drives the gripper to a panel switch to a median error of 18.1 mm against a required 25 mm, on a registration solved from an earlier recording as it is on the rover. Every autonomous controller runs behind an operator heartbeat and a latching software e-stop. The pipeline is demonstrated end to end on the arm. The deployed policy turns a rotary panel switch on roughly four attempts in five, below a teleoperating human but repeatable until it succeeds; lever switches, whose motion first tensions and then snaps over, are harder and carry fewer demonstrations. Plug insertion is the one substep still open, because an operator cannot teleoperate it well enough to demonstrate it, so there is nothing to imitate; a purpose-built tool now exists but has not yet been used to record it.

Chapter 1

Introduction

The European Rover Challenge (ERC)[1] is one of the largest robotics competitions in Europe, held every year in Poland. Student teams from universities around the world build a rover and compete in tasks inspired by real Martian missions. Tasks include autonomous navigation over Mars-like terrain, the collection of probes, stones, and regolith samples, and the operation of a maintenance panel. Teams operate the rover without direct line of sight and are encouraged to perform tasks autonomously.

In 2026, the FHNW Rover Team competes with its newest rover, Barbara [23]. It keeps the previous year's drivetrain and adds a six-degree-of-freedom manipulator carrying a tool changer with a camera on either side, a chassis-mounted camera observing the workspace, a time-of-flight sensor inside the gripper, and a compute stack built around two NVIDIA Jetson Orin NX units on a custom carrier board.

1.1 Motivation

The manipulator is teleoperated. The operator commands it in Cartesian space with a SpaceMouse and inverse kinematics, relying on a digital twin that shows the arm's current pose together with the camera feeds. Commands cross a Wi-Fi link whose limited bandwidth, dropped frames, and latency all degrade an operator's performance on the task [13]. The ERC rewards autonomous control of the manipulator with bonus points, and onboard autonomy removes the link delay from the control loop.

Some ERC tasks also require strict adherence to an order of operations. The sampling task, for example, requires that images be taken before and after the rover manipulates its surroundings. Under the pressure of the competition,

[1] `https://roverchallenge.eu/`

such steps are easily forgotten. Preparing a task as a digital workflow enforces the order the rules require and makes the protocol easy to follow.

The aim of this project is to implement the building blocks for autonomous task completion at the ERC, with the maintenance task as the main focus: a static panel of lever switches, rotary switches, and sockets that must be operated in a prescribed order (section 2.1.1). Points are awarded for completing the steps, and bonus points if parts of the task are done autonomously. Most other ERC tasks award points for technical excellence rather than for autonomy specifically, but automating some of the steps usually earns part of that credit, and it reduces the operators' workload.

Apart from the competition, the same autonomy stack could be useful in the team's own workshop, where the rover could assist in building its successor, for example by clearing the 3D printer's build plate between prints or fetching tools.

1.2 Prior Work

This project builds directly on a previous project [17], which laid the foundation for deep-learning-based control of the rover.

P8 split control into an approach phase and a manipulation phase. In the approach phase, a classical controller aligns the gripper with the target: ArUco markers [24, 46] on the gripper and, for the maintenance task, on the panel are detected in the camera frame, and an Extended Kalman Filter [43] fuses those occlusion-prone detections into a stable pose estimate. In the manipulation phase, a learned policy performs the contact-rich part. That policy was an Action Chunking Transformer (ACT) [61], which predicts a chunk of actions from a single observation. This architecture cannot be conditioned on language, so a separate policy has to be trained per task. P8 also produced the software foundation extended here: an adapter bridging the LeRobot framework [11] with the Robot Operating System 2 (ROS2) [38], a dataset recorder, a policy controller, and a task-progress regressor, developed largely on the SO101 [33], a Hugging Face leader–follower bench arm.

1.3 Research Questions

This project is guided by three research questions:

- **Task automation and reinforcement learning.** How can the maintenance task and further ERC tasks be automated, and how can

reinforcement learning be used to train more capable and robust manipulation policies?

- **Marker-free depth-based control.** How can the gripper be positioned autonomously and precisely using depth information when no fiducial markers are available?
- **Safe handling of errors and plan changes.** How can errors, unknown states, and changes of plan be handled safely and purposefully during autonomous operation?

1.4 Contributions

Five contributions centre on a language-conditioned manipulation policy and the software that trains, deploys, and runs it, together with the visual servoing that places the gripper for it. The manipulator control and safety gating all five run on is described last, as the supporting work it is rather than as a contribution of its own.

1.4.1 Vision-language-action policies with reinforcement learning

A single SmolVLA [51] policy is trained to replace the per-task ACT models of P8. It is conditioned on a natural-language task description, extended with RECAP [2], an advantage-conditioned reinforcement-learning recipe. On the LIBERO benchmark [36] this raises average success from 62.9 % for the base SmolVLA policy to 68.1 %. The policy is also SnapFlow distilled [37] to improve latency. That figure comes from one full pass around the RECAP loop on LIBERO, rollout collection included; the rover lineage skips the rollout stage, so both of its RECAP stages train on demonstrations (section A.2). The LIBERO suite was used to examine the quantitative improvement to SmolVLA, while the rover deployment shows that the resulting policy trains, deploys, and runs on real hardware (section 5.7). This answers the first research question and bears on the third: the architecture is re-tasked by changing the instruction rather than by loading another model (section 5.1).

1.4.2 Workflow automation and agentic control

The open-source automation tool n8n [1] is integrated with ROS2 over a rosbridge WebSocket bridge [18]. n8n gives operators a no-code way to compose task sequences from ROS topics, services, and actions. ERC tasks

are expressed as workflows that enforce the required order of operations. The arm's tool change is also implemented as a workflow: it drives the passive coupling on the rack through joint configurations that were taught once and stored in a table (section 5.5.3). A large language model (LLM) agent is built inside n8n and handles free-text questions and open-ended inspection. It responds by calling the ROS2 tools built here, so a question such as "how is the battery doing?" is answered by a sequence of tool calls. To run it onboard, offline and at no per-call cost, a small vision-capable model (Qwen3.5-0.8B [45]) is fine-tuned on recorded executions of the same agent driven by a cloud-based model. Because n8n bridges ROS2 to a large range of external tools, the same integration is useful well beyond the competition. This work writes some of the task results to Asana, the team's project management platform; a messaging system such as Slack can be connected the same way. Order enforcement and human-in-the-loop confirmation connect this to the third research question as well as the first.

1.4.3 Real-time deployment on the Jetson Orin

To achieve faster inference of the policy on the rover's Jetson Orin NX, the SmolVLA policy is compiled to TensorRT [54]. To do that, it is split into a Vision-Language Model (VLM) prefix and a flow-matching [35] action expert, and the two run at different precisions: converting the whole graph to half precision (FP16) collapses the policy's actions on trained weights, while TensorRT's reduced-mantissa TF32 mode for the prefix (section 4.7.4) does not. In that configuration a chunk takes 434 ms with the full ten-step sampler and 330 ms with SnapFlow single-step distillation, a speed-up of 4.1× over the uncompiled baseline; the engine therefore sustains about 3 Hz of re-planning. The engines are integrated into the live policy controller, selected by configuration. The conversion was implemented using an agentic AI with a goal and tooling to assess its current implementation for errors and accuracy.

1.4.4 Depth sensing

The 8×8 time-of-flight sensor in the gripper is routed into the policy's state input as an additional observation, through a single configuration entry: the state vector grows by those 64 cells and nothing else about the policy changes. If no board markers are available for the visual servoing phase, the policy must perform approach and manipulation. The sensor integration supplies the policy with information the cameras cannot provide and costs almost nothing. Whether the policy is better for it was not measured: the grid

is kept on the cost argument rather than on a demonstrated gain, and the ablation is left to future work (sections 5.4 and 6.3.3).

1.4.5 Visual servoing

A classical approach phase is kept alongside the policy. Not every ERC subtask is equally constrained by language. "Pick up the stone", from the sampling task, names its target, so a subtask like it could be given to the policy alone, working from camera images and the in-gripper depth grid with no fiducial and no alignment phase in the loop; that path is built but untested (section 5.4). "Rotate the third switch from the left" is a more complex sentence and requires deterministic output, something a VLM cannot provide. The approach phase exists for precise placement against repeated, ambiguous targets, and it gives the policy a consistent starting pose in a known frame.

That phase is inherited from P8, where it localises the board with ArUco markers seen by a base-mounted camera. P9 implements the whole perception and control chain in C++, for the margin that leaves on the onboard compute while a policy runs alongside it (section 5.2). The improved inverse kinematics of the new manipulator makes it possible to rely on it for the approach. The markers on the gripper are only used for initial registration of the transform between the camera and the frame of the manipulator. This also makes it possible to align the gripper with targets that fall outside the camera's field of view.

1.4.6 Manipulator control and safety gating

The six-degree-of-freedom manipulator is driven with FZI's Cartesian controllers (Forward Dynamics Compliance Control) [47]. This work added a per-joint weighting to the solver that sets how much each axis contributes to a Cartesian motion, which damps the forearm roll so it no longer absorbs wrist rotations. A MuJoCo [56] model of the manipulator was also built and integrated with ROS2, for smoke tests and as a target for developing the ArUco alignment. It was not used for policy training.

Both autonomous control modes (policy and visual servoing) are gated to ensure safe operation. A heartbeat from the operator station must be present, so a lost link or a crashed operator interface halts autonomous motion without requiring operator action; and a latching software e-stop aborts the running policy or alignment goal on demand and refuses new ones until it is reset. With the rover's hardware stop and the cancellable phase actions they form four layers of stopping that differ in what they leave running and how they

resume (section 2.1.4). That is the interruption half of the third research question; the recovery half sits in the workflow layer above.

1.5 Document Structure

Chapter 2 presents the system design and the main architectural decisions. Chapter 3 gives the theoretical background of the methods used. Chapter 4 describes the implementation of the five contributions and the manipulator work beneath them. Chapter 5 presents the experimental results, and chapter 6 summarises what was achieved, states the limitations, and outlines future work. The appendices carry the material the body refers to but does not print: the training runs behind both policy lineages and the agent model (chapter A), the agent transcripts (chapter B), and the workflow canvases too wide for the page (chapter C).

Chapter 2

System Design

This chapter gives an overview of the system that runs on the rover to solve the maintenance task and perform other autonomous operations. First the requirements and system constraints are presented, then the chosen solution, and finally the hardware and software.

2.1 Requirements

The requirements of this project are derived from the rules of the ERC, the available hardware, and safety considerations.

2.1.1 Maintenance panel

The maintenance task is the main focus of this project. It requires operating on a static panel (fig. 2.1): setting a main switch, flipping four of several lever switches, turning three of five rotary switches, inserting a plug into one of two IEC 320 C14 sockets, turning two rotary power switches, and finally grasping and placing a plate on an electromagnet [21]. The elements are repeated and sit close together, and the plug demands a precise insertion. The models, the exact dimensions, and the placement of the panel and its elements are not published in advance. The approach phase has to place the gripper close to the target element in the panel plane, so the policy has no ambiguity about which switch to operate. That sets the tolerance at 25 mm: from a gripper standing that close, the element to be turned or flipped is unambiguous in the gripper cameras' view, and the policy is left to manipulate one switch rather than to choose among the neighbouring ones. ArUco markers on the panel provide the reference poses for the alignment phase.

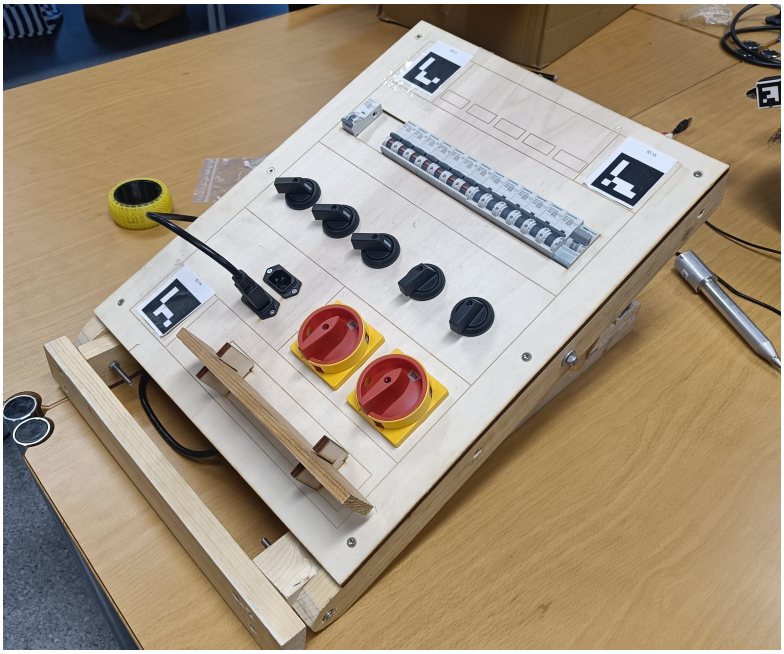


Figure 2.1: Replica of the maintenance panel used at the European Rover Challenge 2025/2026.

2.1.2 Performance

For autonomous control, the arm must be commanded at 20 Hz, the frame rate of the cameras and the rate the demonstrations are recorded at (section 2.8), and the stack must run on the rover's own compute, which is an NVIDIA Jetson Orin NX with 16 GB of unified memory. Inference does not need to run at 20 Hz, since action chunking makes it possible to generate a batch of actions with one observation.

2.1.3 Safety

A human operator must always be able to disable autonomous control. If the connection to the rover is lost, autonomous motion must stop in under a second to prevent unobserved actions that could damage the rover or its surroundings. This requirement drives the operator-heartbeat deadman described in section 2.5.1.

Autonomous in this context means a controller that decides its own commands, like the learned policy or the alignment controller. A move to a joint configuration stored beforehand is not autonomous in this sense. It is fully determined before it starts and runs through the manipulator stack

rather than through either controller. Such a movement is therefore not gated by the safety mechanisms here.

The ERC rules require a beacon to indicate autonomous operation, and autonomous motion may begin only after a 5 s delay. Autonomous control may also fail mid-task, so it must be able to retry an operation and, if it keeps failing, hand back to a human operator and later resume.

2.1.4 Layers of stopping

The rover carries a hardware emergency stop that cuts power to everything on board, the compute included [23]. It is the final safety layer and the only one still working when the software is unresponsive. It causes the arm to lose holding torque, and the autonomy stack has to be restarted afterwards.

Above it sits a software e-stop, a latching abort. Any message on the `/e_stop` topic aborts the active policy and alignment goals, and new autonomous goals are rejected until a reset is published on `/e_stop/reset`. The middleware for these topics, services, and actions is introduced in section 2.4.3. The arm holds its position under its controllers, so no holding torque is lost.

The operator-heartbeat sits above that (section 2.5.1). Autonomous commanding halts in the absence of the signal, so a lost link or a crashed operator interface both stop it. At the top, the `run_policy` and `align_to_target` actions can be cancelled, which ends the current phase cleanly and returns control to the orchestration layer. Figure 2.2 sets out the triggers and recovery paths of all four.

Recovery from the hardware stop is a reboot. The manipulator starts up read-only, enforced by the hardware interface, so an operator has to enable writes before anything moves. The joint zeros are stored and survive the power cut; what a restart needs is the coarse homing routine that resolves the multi-turn ambiguity on the joints that have it, run through the hardware interface of section 4.10.

2.2 Solution

ERC tasks are broken down here into substeps, such as turning a single switch on the panel, and a substep normally runs in two phases. The *approach* brings the gripper close to the region to be manipulated, driven by deterministic algorithms built on marker-based localisation and the manipulator's kinematics. The *manipulation* then performs the action at that position. A substep may iterate between the two, for example to pick up the metallic rod at one

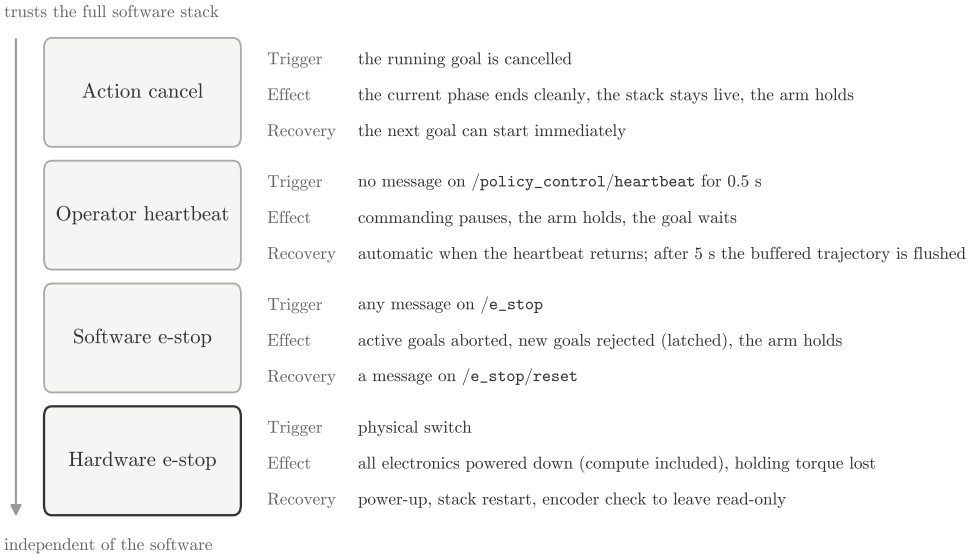


Figure 2.2: The four stopping layers, ordered by how much of the system they trust. The upper three hold the arm in position under its controllers and differ in how they resume; only the hardware stop cuts power, and it cuts it to the entire rover, compute included.

place and mount it at another. The split is inherited from the previous work; P9 adds a language-conditioned policy in the manipulation phase and the tooling around it, and improves the alignment controller.

2.2.1 Phase 1 – Classical alignment

The approach phase is driven by two sets of ArUco markers. The gripper markers fix the camera-to-base transform, solved offline from a dedicated recording that moves the wrist through a range of attitudes (section 4.2.2). An Extended Kalman Filter can then track the gripper pose through that transform, fusing the marker measurement with a forward-kinematics estimate, so the estimate survives the poses where no marker is usable. The rover runs the registration alone: it needs no marker in view to start a task, so once the transform is stored the gripper is carried by forward kinematics and the live fusion stays available rather than deployed (section 5.2.1). The panel markers give the target frame: their board coordinates are known, so one pose estimate fixes the panel (section 4.2.3). A second Extended Kalman Filter averages that pose, treating the panel as static (section 4.2.2). A detector node locates the individual switches and sockets on the panel in the frame of the panel. With the known camera-to-base and board-to-camera transforms,

these detected objects can be approached by the gripper using a classical control loop.

2.2.2 Phase 2 – Policy node

The manipulation phase is executed by a deep-learning policy. It receives the task as a textual instruction together with the camera images and the manipulator's joint state; depth is additionally provided by the time-of-flight sensor in the palm of the gripper (section 2.3.3). The policy runs in open loop: one observation yields a whole chunk of actions, which the controller executes and replenishes with a fresh prediction once fewer than a configured number of them are left in the queue (section 2.9). When to stop is decided by a separate progress model (section 2.7).

2.2.3 Command space: joints or end effector

The approach phase commands the *end effector's Cartesian pose*: its goal is a pose relative to the panel, so it is expressed as a Cartesian target and uses the inverse kinematics to resolve it into joint positions (section 3.1).

Cartesian control uses FZI's Cartesian motion controller, with a per-joint weighting added to the solver to tune how much each axis contributes to a Cartesian motion. It damps the forearm-roll axis, which otherwise swings widely for a small tool rotation (section 4.10).

In contrast, the manipulation phase commands *joints* directly. This follows the LeRobot default and the shape of the recorded demonstrations. It also keeps the inverse-kinematics solver out of the learning loop: the policy's output reaches the hardware directly, so its behaviour does not depend on how the solver resolves redundancy or singularities. The cost is that the policy is tied to this arm's mechanical construction.

2.2.4 Workflow automation

To compose these substeps into a complete task, the open-source workflow-automation tool n8n is used. For that, a custom integration connecting n8n over a rosbridge WebSocket is implemented. Each ERC task is built into a workflow of connected blocks that call ROS services and actions in the order the rules demand, with other external tools and flow logic. The same bridge lets ROS2 reach the external services n8n integrates, such as an issue tracker or a messaging system. The bridge and node package are detailed in section 4.9.

2.2.5 Agentic control

n8n also provides a feature that allows setting up an LLM-backed agent with access to the same n8n blocks and ROS2 nodes. It handles open-ended requests from a chat interface, reading rover state and calling ROS2 services. It is mainly used for diagnosis and not to execute actions on the rover. The package implements a topic-namespace allowlist and a read-only credential mode, which narrow the services and topics an attached tool may reach (section 4.9.7). The agent works with either a cloud or a locally served model. The locally served model is fine-tuned to improve its performance for the specific tasks on the rover.

Carrying binary results through the tool path required a change to n8n itself, since the agent implementation of n8n could attach an image only once, at the start of a conversation. The change was contributed upstream as a pull request; with it, a node's binary output reaches the model as a multimodal content block (section 4.9.5).

2.3 Hardware

The full mechanical and electrical design of the rover is documented in the team's ERC final report [23]. The most important parts of the hardware for autonomy are detailed here.

2.3.1 Rover manipulator

The rover carries a custom-built six-degree-of-freedom manipulator. Joints 4–6 run through a Bowden pulley system, which keeps mass off the differential wrist. The arm ends in a tool changer, so different tools can be mounted. ArUco markers on the gripper are visible to the base camera.

2.3.2 Cameras

Three cameras are mounted on the manipulator. Two sit on either side of the tool changer, looking down on the gripper and the workpiece. The third is on the chassis and covers the area directly in front of the rover, where the maintenance panel is placed or stones and probes are picked up.

2.3.3 Gripper depth sensor

An ST VL53L8CX [53] multi-zone time-of-flight (ToF) sensor sits in the palm of the gripper. It is run in its 8×8 zone mode, so it returns a grid of 64

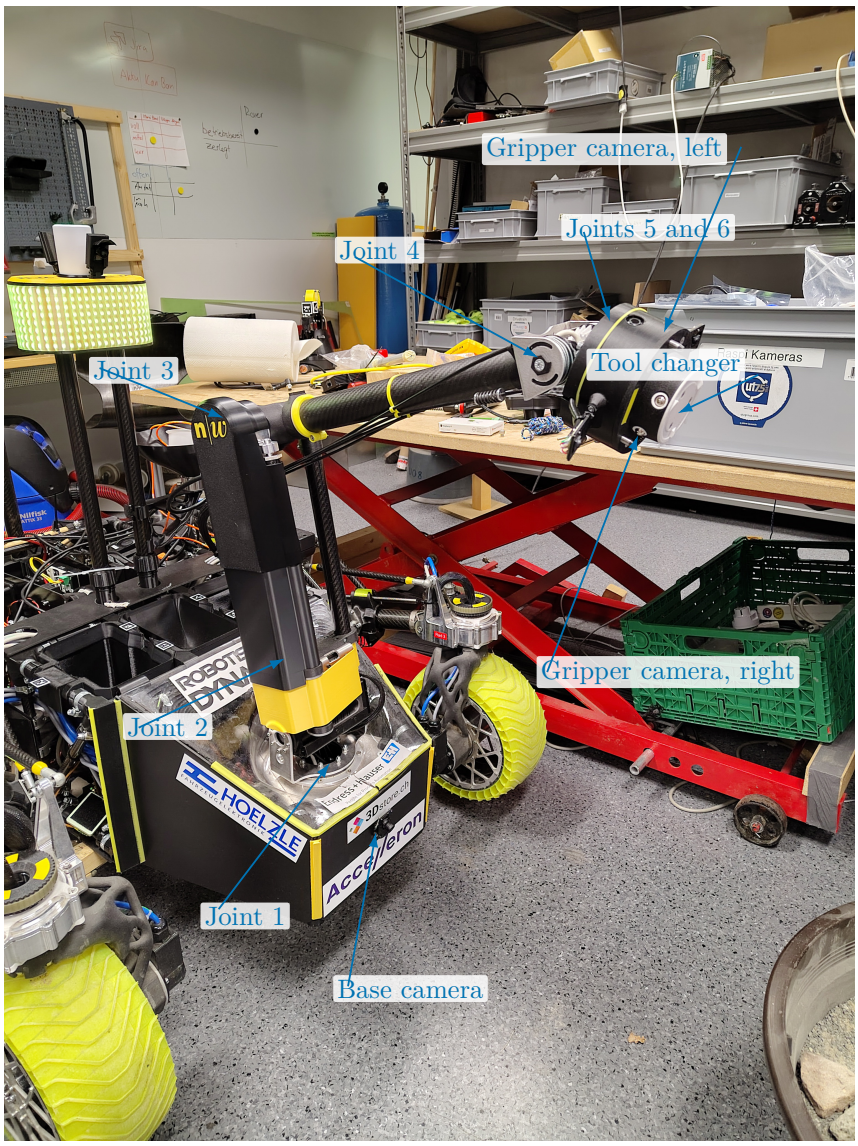


Figure 2.3: The manipulator mounted on the rover “Barbara”, with no tool fitted. Joints 1 to 3 carry the arm, joint 4 rolls the forearm about its own long axis, and joints 5 and 6 are the pitch and roll of the wrist, driven by two motors through a differential transmission. The arm ends in the tool changer, with a camera on either side of it. The base camera that watches the gripper and the panel (section 4.2) is mounted on the chassis below.

per-cell distances in millimetres across a square field of view of 45° on a side and 65° on the diagonal, and it is configured for the part's full ranging span, 4 m per zone. The LeRobot–ROS2 adapter consumes the array directly, maps the raw millimetres into $(0, 1]$, and routes it into the policy's state input (sections 4.4.1 and 4.8).

On the rover, the sensor publishes at 5 Hz, a quarter of the frame rate, so each reading is held across four frames of a recording. All 64 cells are passed to the policy; none are pooled or discarded.

2.3.4 Compute

The autonomous stack runs onboard on two NVIDIA Jetson Orin NX 16 GB units, one in the drivetrain and one in the manipulator.

2.4 Software Architecture

The autonomous components are separate ROS2 packages that run alongside the rover's existing nodes, reading and writing the topics named in their configuration (section 2.9). On the rover, they are deployed in a single Docker container, started with the NVIDIA container runtime so the Jetson's GPU is available to the policy from inside it.

2.4.1 Deep-learning framework

Policies are built with LeRobot [11], a PyTorch [3] framework for robot learning that provides implementations of common policy architectures and a dataset format for robotic data. The deployed policy is based on SmolVLA and is post-trained with RECAP [2], an advantage-conditioned reinforcement-learning recipe; Knowledge Insulation [20] and SnapFlow distillation (chapters 3 and 4) are also implemented for SmolVLA, to improve learning and inference speed. An adapter bridges LeRobot to ROS2, converting between ROS messages and the tensors the policy expects.

2.4.2 AI-assisted development

The software described in this report was developed with AI coding assistants throughout. In some components, an agent wrote the bulk of the code under direction, above all the n8n node package (section 4.9), while elsewhere the assistance was ordinary tooling: refactoring, code documentation, and quality

control on code the author wrote. The Index of Assistive Technologies lists the systems used and the scope of their use.

The TensorRT export pipeline was produced by giving an agent a goal, a validation oracle, and the hardware to test against, and letting it iterate until the oracle passed. The process is described in section 4.7.5 alongside the pipeline.

2.4.3 Robot Operating System

The middleware is ROS2 Jazzy Jalisco [38], and it is used for all the implemented nodes since this is the current stack of the Rover Team.

2.4.3.1 Messages and topics

Sensor and command data use the standard ROS2 messages: `sensor_msgs/Image` and `CompressedImage` for cameras, `sensor_msgs/JointState` for the manipulator state, and `std_msgs/Float64MultiArray` for the joint commands the policy publishes. For the gripper, a separate `std_msgs/Float32` is used for the state and command. The gripper is the one command that sits outside the joint array, on its own `std_msgs/Float32`. This can be configured and adapted to most robotics systems, as long as they rely on the commonly used topics (section 2.9). P9 adds two custom interfaces into `lerobot_ros`: `TaskProgress` (`string policy_name`, `float32 progress`) reports task completion, and `PolicyStatus` latches the state of the policy controller (section 2.5.2).

2.4.3.2 Services and actions

The OpenCV [10] panel detector serves `detect_board_objects` and `select_board_object` (section 2.6). Long-running, cancellable operations use ROS2 actions: `run_policy` (section 2.5.1) executes a policy while streaming progress feedback, and the relative controller exposes `align_to_target` for board-relative alignment.

2.5 Control Flow

Figure 2.4 shows the manipulator stack with the four paths that control the arm. Manual control runs from the SpaceMouse through `twist_to_pose`, which integrates the operator's twist into a pose target and bounds it so it cannot drift away from a blocked arm (section 4.10). The relative controller and the policy controller drive the two autonomous paths, the

phases of section 2.2.1; the policy controller reads the cameras, the joint state, and the ToF grid through one adapter configuration (section 4.4).

Manual control and the approach phase meet on the same topic, `target_frame`, so the alignment drives the arm through the same interface a human does. The policy commands joint positions directly, on `position_controller/commands`. The two topics reach the hardware through different `ros2_control` controllers, only one of which may be active at once, so a policy run has to acquire and release its controller (section 4.3).

The fourth path is orchestration: n8n starts a phase through the `run_policy` and `align_to_target` actions and gets progress back while it runs (section 2.2.4).

The safety gating of section 2.1.4 lies across all of it. The operator heartbeat is published by the SpaceMouse node while its `auto` parameter is true. That parameter is toggled either from the dashboard or with a button on the SpaceMouse. Taking the arm back to manual control stops the heartbeat.

2.5.1 Policy execution and operator heartbeat

A policy is executed through the `run_policy` action. Its goal carries the policy and the natural-language task, and it streams back progress, elapsed time, and status while it runs. It ends with a result of `completed`, `timeout`, `cancelled`, or `e-stop`.

The heartbeat that gates the run arrives on `/policy_control/heartbeat`. If none arrives within a configurable timeout (default 0.5 s), the controller stops publishing commands and the arm holds position while the goal stays active (section 4.3).

2.5.2 Autonomous system state

The policy controller publishes its current state on a latched `PolicyStatus` topic, so a subscriber that connects late still sees the message. It reports the running policy in `active_policy`, which is empty under manual control, the loaded set in `available_policies`, and the task, the progress, and the liveness of the heartbeat. The `ros2_control` switch of section 4.3 decides which controller may command the hardware. The ERC beacon and its 5 s start delay (section 2.1.3) are implemented in the workflow layer: entering autonomous mode sets the beacon to blue, waits the required 5 s, and only then enables autonomous commanding.

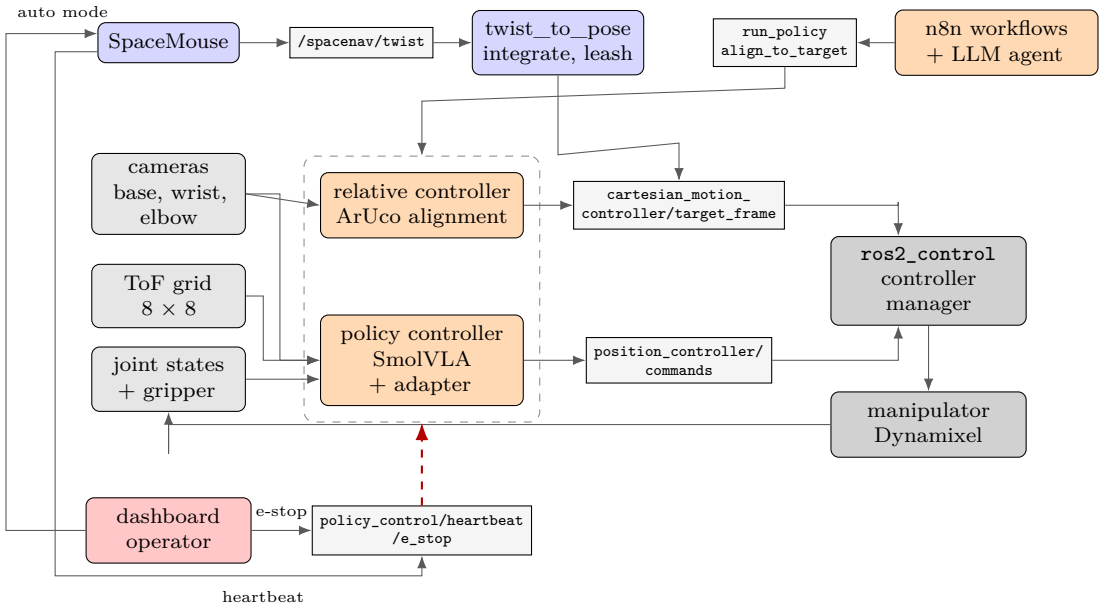


Figure 2.4: The manipulator stack. Rounded boxes are nodes, sharp boxes are topics and actions; blue is the manual path, orange the autonomous paths and the orchestration that starts them, grey the sensors and the hardware interface. Each side of the dashed autonomy block carries one role: observations enter from the left, goals from the top, the safety gating from the bottom, and commands leave to the right. The two command topics reach the arm through different `ros2_control` controllers, of which only one is active at a time.

2.6 Nodes

The classical perception and alignment nodes live in `barbara_autonomy` (Python), with a C++ mirror `barbara_autonomy_cpp` that reimplements the performance-critical ones for the onboard compute budget. The deep-learning nodes live in `lerobot_ros`.

In `barbara_autonomy`:

- **`aruco_detector_node`** — detects ArUco markers in the camera images and publishes their poses.
- **`pose_filter_node`** — fuses the marker detections with an Extended Kalman Filter into a stable gripper/panel pose.
- **`relative_controller_node`** — servos the gripper to a board-relative target using the filtered pose, exposed as the `align_to_target` action.
- **`board_object_detector_node`** — detects panel switches and sockets with OpenCV and serves `detect_board_objects` and `select_board_object`.

In `lerobot_ros`:

- **`policy_controller`** — runs the LeRobot policy behind the `run_policy` action, with heartbeat gating and progress reporting.
- **`dataset_recorder`** — records episodes from live ROS2 topics into LeRobot datasets.
- **`episode_tracker`** — estimates task completion with a learned progress model (section 2.7).
- **`bag_to_dataset`** — converts recorded ROS2 bags into LeRobot datasets.
- **`annotation_server`** — serves a GUI for annotating recorded datasets.

2.7 Progress Tracker

The `episode_tracker` estimates how far the current task has progressed and publishes it as a `TaskProgress` message. It is surfaced to the operator and stops the policy: crossing a completion threshold (default 0.9) ends the `run_policy` action, and a missing model falls back to a timeout. The tracker was reworked from the single-frame regressor implemented in P8: it reads a strided window of recent frames, so the estimate is based on about 1.5 s of motion rather than on a single observation (section 4.5).

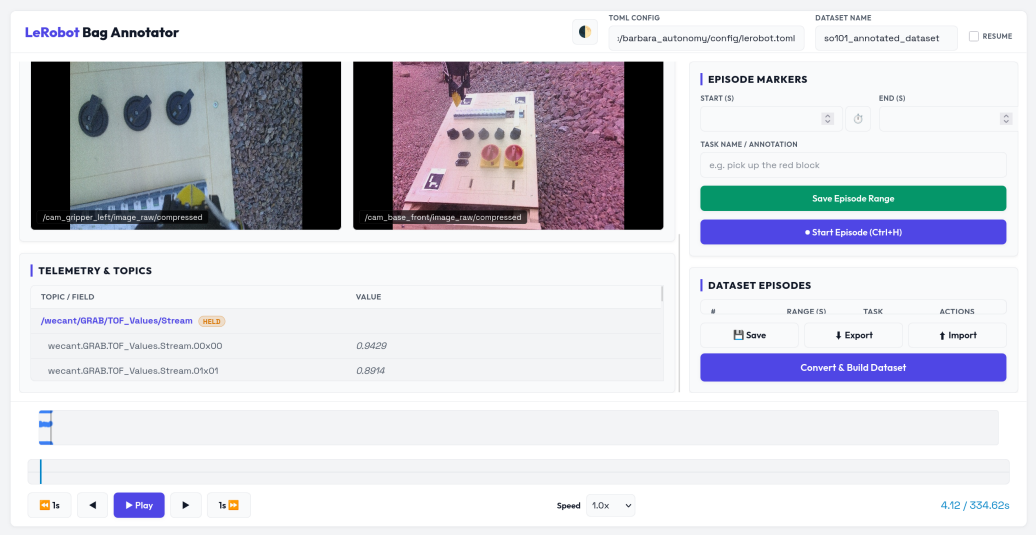


Figure 2.5: The annotation GUI replaying a bag. The configured camera streams are shown on the left and the remaining topics as telemetry below them; the ToF values carry a *held* tag because the last message is repeated until a newer one arrives. Episode boundaries are marked on the timeline at the bottom and given a task description on the right.

2.8 Dataset Creation

Training data is collected as plain ROS2 bags and converted into a *LeRobotDataset* offline.

The episode boundaries and tasks are entered through the annotation GUI, served by the `annotation_server` node (fig. 2.5). It visualizes the bag frame by frame, according to the configured frequency of the dataset. The operator marks the start and end of an episode and gives each a task description; everything outside a marked segment is discarded, which removes the reset and idle phases between attempts (section 4.4.4). The annotated episodes can be exported and imported, so the same bag can be converted into different datasets with different sets of available features.

The `bag_to_dataset` tool resamples the asynchronous bag into the dataset's fixed frame rate and episodes (sections 3.4 and 4.4.3).

Table 2.1 lists the manipulator datasets recorded this way and published on Hugging Face.[1] Each frame carries three camera streams and one 83-dimensional state vector: the 64 cells of the ToF grid (section 2.3.3), the six joint positions, their velocities and efforts, and the gripper aperture.

[1] `https://huggingface.co/fhnwrover`

The action is the six commanded joint positions plus the gripper command. Observation and action therefore overlap in the six joint positions, which makes the dashed and solid traces in fig. 2.6 follow each other.

Table 2.1: The manipulator datasets published under the `fhnwrover` account, all at 20 Hz. The three per-task sets carry one instruction each. `manibar-erc_various` carries four in one dataset, the three above plus "rotate the black switch to the left", so the black switch is demonstrated in both directions and two of the four instructions turn a switch left.

dataset	instruction	episodes	frames
`manibar-erc-black_switch`	rotate the black switch to the right	83	26 634
`manibar-erc-red_switch`	rotate the red switch to the left	24	9730
`manibar-erc-fi_switch`	flip the switch	11	3300
`manibar-erc_various`	four instructions, see caption	120	40 432

Figure 2.6 shows what comes out of the conversion, on one episode of `manibar-erc-black_switch`.

2.9 System Configuration

For configuration, the nodes rely on ROS2 parameters. They can be set as an argument to the node for local testing, or configured through a YAML file for deployment.

The LeRobot–ROS2 adapter is configured with a TOML (Tom's Obvious Minimal Language) file that maps ROS topics to policy inputs and outputs. Listing 1 shows an excerpt of the configuration the manipulator runs: each topic is a key giving its path, its message type, and the tensor it maps to. On the rover, both files, the adapter's TOML and a YAML of the ROS2 parameters, are deployed in a single container started from a launch file. The full configuration is kept in the rover-config repository.[2]

Several policies can be configured in one file (listing 2), and the operator needs to specify which one to use by name when starting a `run_policy` action; the set is published as `available_policies` and the running one as `active_policy`.

`action_queue_size` is the number of queued actions still available in the queue before the controller starts the next inference, so at 20 Hz the forward

[2] `https://gitlab.fhnw.ch/fhnw-rover/rover-config/-/tree/main/config/autonomy`

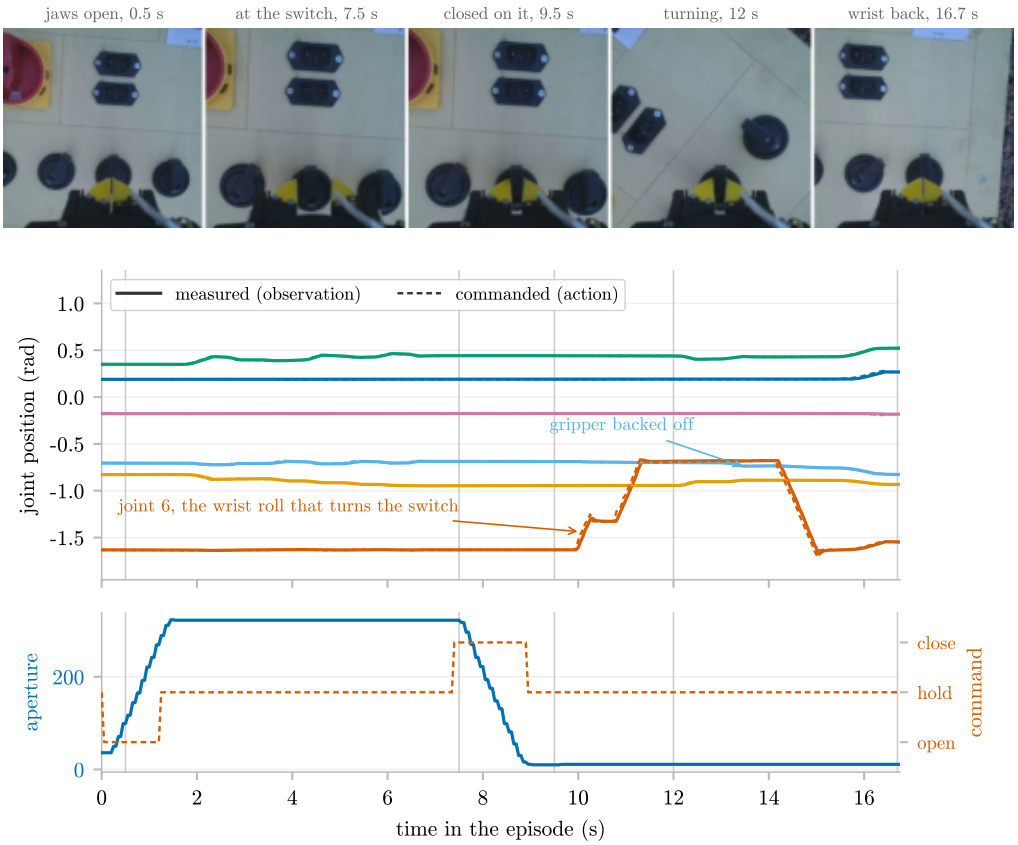


Figure 2.6: One episode of the panel-switch dataset. Top: five frames of the left gripper camera, at the marked instants. Middle: the six joint positions, measured (solid) against commanded (dashed) in the same colour; the vertical lines mark the camera frames above. Bottom: the gripper, with the measured aperture in raw units on the left axis (0 closed, 300 open) and the commanded jaw motion on the right, which is a discrete open, hold or close command. The demonstration opens the jaws, holds them open while the arm settles at the switch, closes them at 7.5 s, and turns the switch with the wrist roll. It then backs the gripper off the switch, at 13.5 s, before rolling the wrist back to its starting orientation.

pass gets that many 50 ms cycles before the queue is empty. It also sets the re-planning rate, because the controller predicts every `n_action_steps` minus `action_queue_size` actions. `n_action_steps` counts the actions of a chunk the controller consumes, which can be lower than the actions the policy predicts: ACT predicts 100 and the controller uses the first 50.

SmolVLA is deployed with a queue size of 40. That is 2 s of budget for the 330 ms chunk of the SnapFlow engine and a re-plan every ten actions, so 2 Hz. ACT fallback sits at 25. Under-budgeting does not fail outright; instead the queue drains and the arm stalls mid-chunk, so the entry has to be revisited whenever the inference path changes. Predicted action chunks are combined with the remainder of the queue using blending between the old and the new actions.

`limits` clamp predicted actions to a range, so a mis-predicted action cannot command a value outside it, and `round_values` snap them to a set of allowed values (section 4.4.1). A policy emits floating-point values, so a downstream node that expects a discrete input would receive invalid values without the snap. The gripper command in listing 1 is one such output: a velocity command that opens (+1), holds (0), or closes (−1) the jaws.

Observations are reshaped by a separate mechanism, `transform`, which applies a monotonic elementwise map to the values of a message. It exists because normalisation in the training pipeline is affine, so it can shift and scale a sensor's range but not redistribute it. The adapter implements two such maps, `exp_decay` and `reciprocal`. The ToF grid is the case they were added for. The deployed configuration uses `exp_decay`: it gives the first few centimetres, the part that decides a grasp, far more of the numeric range than a linear map over the sensor's four-metre span would (sections 4.4.1 and 4.8).

```toml
fps = 20

[topics."/cam_gripper_left/image_raw/compressed"]
msg_type = "CompressedImage"
qos      = { reliability = "best_effort", depth = 10, history =
↪  "keep_last" }
height   = 360
width    = 640
key      = "gripper_left"
tag      = "observation"

[topics."/wecant/GRAB/TOF_Values/Stream"]
msg_type  = "Int16MultiArray"
key       = "tof"
tag       = "observation"
transform = { type = "exp_decay", scale = 800.0 }
names     = ["00x00", "01x01", "..."]   # one entry per cell,
↪  64 in total

[topics."/position_controller/commands"]
msg_type = "Float64MultiArray"
tag      = "action"
names    = ["joint_1", "joint_2", "...", "joint_6"]

[topics."/wecant/GRAB/Grip_Man/Set"]
msg_type     = "Float32"
tag          = "action"
round_values = [-1.0, 0.0, 1.0]
```

Listing 1: Excerpt of the manipulator configuration, mapping ROS topics to policy inputs and outputs. Topic lists are abbreviated and quality of service (QoS) is omitted where it is the default. The policy section of the same file follows in listing 2.

```toml
[policies."manibar-smolvla-recap-snapflow"]
# fhnwrover/smolvla_recap_manibar-erc_various
pretrained_name_or_path =
↪  "/data/models/smolvla_recap_snapflow_manibar"
device                  = "cuda"
use_trt                 = true
trt_engine_dir          =
↪  "/data/models/smolvla_recap_snapflow_manibar/trt"
policy_config           = { n_action_steps = 50 }
max_episode_length_s    = 30.0
action_queue_size       = 40
action_smoothing_beta   = 0.5

[policies."rotate-black-switch"]
# fhnwrover/manibar-act-erc-black_switch
pretrained_name_or_path =
↪  "/data/models/manibar-act-erc-black_switch"
device                  = "cuda"
policy_config           = { n_action_steps = 50 }
max_episode_length_s    = 10.0
action_queue_size       = 25
action_smoothing_beta   = 0.5
```

Listing 2: The policy section of the same configuration (listing 1), declaring the two policies the arm runs. `manibar-smolvla-recap-snapflow` is the SmolVLA policy with RECAP post-training and SnapFlow distillation, served from its TensorRT engines (section 4.7); `rotate-black-switch` is the single-task ACT policy.

Chapter 3

Theory

This chapter contains the theory behind the implemented contributions.

The classical approach phase rests on inverse kinematics and Cartesian control (section 3.1) and on the visual servoing that supplies the target: fiducial markers, the eye-to-hand configuration, and the Kalman filter that fuses the two pose sources (section 3.2).

Sections 3.3 and 3.4 introduce the framework and the dataset format used for the deep-learning-based policies. Section 3.5 covers SmolVLA and its flow-matching action expert together with the RECAP recipe, covering advantage conditioning, the critic, classifier-free guidance, self-improvement and knowledge insulation, and SnapFlow distillation that raises the rate at which it can re-plan. Section 3.6 covers the inference optimisation, and sections 3.7 to 3.9 the simulator, the benchmark and the depth sensor.

Parts of the orchestration layer above both phases are in section 3.10: agents, tool use, and where a deterministic workflow is preferable to an agent.

3.1 Inverse Kinematics and Cartesian Control

The arm can be commanded in joint space, as target joint angles, or in Cartesian space, as a target pose of the end effector. To go from a target expressed in Cartesian coordinates, inverse kinematics (IK) is required. The approach phase commands Cartesian poses, while the manipulation policy commands joints directly (section 2.2.3).

The rover manipulator uses the Forward Dynamics Compliance Control (FDCC) solver [47]. FDCC models the arm as a virtual dynamic system. The Cartesian pose error is treated as a force f applied at the end effector, and the joint accelerations follow from the equation of motion

$$\ddot{q} = H^{-1} J^{\top} f, \tag{3.1}$$

where J is the manipulator Jacobian and H a virtual joint-space inertia matrix. Integrating these accelerations with a forward-Euler step drives the joints so the end effector moves toward the target. Because the system is simulated rather than physical, H need not be the true inertia; it is a tuning parameter.

The solver is adapted for this manipulator. The first adaptation weights the joints through H, whose diagonal entries hold one value per joint: a larger value makes that joint resist acceleration and contribute less motion under the same Cartesian force, which stops a single joint from dominating a motion the remaining joints could perform equally well. The second adds a per-joint spring and damper to the integrated acceleration rather than acting through H, pulling a joint back towards zero. The affected axis and the parameter values are given in section 4.10.

3.2 Visual Servoing

Kinematics-based control is confined to the manipulator's own frame. A maintenance task is not specified there: the switch to be operated is a position on the panel. *Visual servoing* supplies the relation between the manipulator's frame and the board's frame by putting a camera in the loop [30, 12]. Tool and target are measured in a single frame, so a target read off the panel becomes one the arm can be commanded with.

This work uses *position-based* visual servoing, which reconstructs a full 6-DoF pose from the image and defines the error in Cartesian space. The fiducial markers below give a metric pose, and the resulting Cartesian error is handed to the same Cartesian controller as any other target (section 3.1). This needs a calibrated camera and a known marker geometry. In exchange the error comes out in the units the task is specified in, millimetres of misalignment to a switch, rather than in pixels.

3.2.1 Fiducial markers and pose estimation

The visual feature used throughout is the ArUco marker [24, 46]: a square binary pattern whose black-and-white grid encodes an identifier together with error-correcting redundancy, drawn from a fixed dictionary. A detector recovers the candidate quadrilaterals in an image and reads each one's bit pattern against the dictionary, which both identifies the marker and rejects false positives.

The marker is planar with a known side length, so its four detected corners give four 2D–3D correspondences. That is enough to recover the marker's

pose relative to the camera, by solving a Perspective-n-Point (PnP) problem against the camera's intrinsic calibration. Several markers of known relative layout can be solved jointly as one rigid body, which over-determines the problem and makes it more robust.

The dominant error mode of a single small marker detection is orientation, not position. Corner localisation is accurate to a fraction of a pixel, but inferring the out-of-plane rotation from a near-fronto-parallel square is ill-conditioned, so small corner noise produces large tilt errors. It also leaves an ambiguity: two distinct poses explain the observation almost equally well [50]. For a single square marker the plane-specific IPPE (Infinitesimal Plane-based Pose Estimation) method is used, which returns both candidate poses with their reprojection errors instead of silently picking one [16]. When the two errors are close the reprojection cannot choose between them, so the filter keeps the candidate that agrees with its forward-kinematic orientation prediction (section 4.2.2). Markers placed in different planes constrain the rotation jointly, so no single ill-conditioned square decides it, as the paired gripper markers of section 4.2.1 do.

3.2.2 Eye-to-hand configuration

The configuration used here is eye-to-hand: markers are fixed to the gripper jaws and a base-mounted camera sees the gripper and the target in the same frame. Localising both bodies in one image gives the transform between the camera and the arm base without a separate calibration rig.

The base camera is far from the gripper and must cover the whole workspace, so the gripper markers are small in the image and their orientation estimate is correspondingly noisy. The arm also occludes its own markers at the poses where precision matters most, close to the panel.

Where the target is fixed in the world and only the tool must be placed, the two sources complement each other. Forward kinematics is available every cycle and free of occlusion; the marker observation is intermittent, but independent of the arm. The filter below fuses them.

3.2.3 Fusing pose measurements with a Kalman filter

The filter is an Extended Kalman Filter, carried over from P8 and reimplemented in C++ for P9 (section 4.2.2). A Kalman filter maintains a state estimate together with a covariance that expresses how much it is trusted [43, 6]. Prediction advances the state and inflates the covariance; update folds in a measurement, weighted against the prediction by their relative covariances, and shrinks it again. Where a process or measurement model is nonlinear, the

extended variant linearises it about the current estimate; here the nonlinearity sits outside the recursion instead, which lets the update collapse to the linear form below.

The tracked state is the pose of one object, the gripper or the panel, in the camera frame:

$$x = \begin{bmatrix} p \\ q \end{bmatrix} \in \mathbb{R}^7, \qquad p \in \mathbb{R}^3, \quad q = [q_x, q_y, q_z, q_w]. \tag{3.2}$$

T_{XY} denotes the pose of frame Y expressed in frame X, with C for the camera, B for the arm base, T for the tool centre, O for a tracked object and M for a marker. The prediction step of the gripper filter is the kinematic pose of the tool in the camera frame,

$$T_{CT} = T_{BC}^{-1}\, T_{BT}, \qquad x^- = \mathrm{vec}(T_{CT}), \qquad P^- = P + Q, \tag{3.3}$$

where T_{BT} comes from the forward kinematics and T_{BC} is the camera pose in the base frame. The panel filter tracks a static object, so its prediction leaves the state untouched and only inflates the covariance, $x^- = x$ and $P^- = P + Q$. The prediction runs on a fixed timer at 50 Hz, independently of when measurements arrive.

The gripper prediction discards the previous state, so as far as the covariance is concerned the state transition is the identity. Entering the kinematic pose as the process model is a deliberate simplification: it lets the filter keep running whenever no marker is visible.

A measurement is a marker detection, which gives the marker pose T_{CM} in the camera frame. Since the marker's offset T_{OM} in the object frame is known from the survey, the detection is first turned into a measurement of the object origin itself,

$$T_{CO} = T_{CM}\, T_{OM}^{-1}, \qquad z = \mathrm{vec}(T_{CO}). \tag{3.4}$$

This composition is the only nonlinear part of the filter. Otherwise the measurement is the state, so the measurement Jacobian is the identity and the usual update collapses to

$$y = z - x^-, \qquad S = P^- + R, \qquad K = P^- S^{-1}, \tag{3.5}$$

$$x = x^- + Ky, \qquad P = (I - K)\, P^-. \tag{3.6}$$

The orientation is carried as a quaternion. A quaternion and its negation describe the same rotation, so the measured quaternion is sign-aligned with the predicted one before the innovation is formed: $q^z \leftarrow -q^z$ whenever

$(q^z)^\top q^- < 0$, writing q^z for the orientation part of the measurement z and q^- for that of the prediction x^-. Otherwise the filter would see a large error where there is none. The update is then a linear combination of two unit quaternions, whose result is not itself a unit quaternion, so it is renormalised after every update. Treating the four components as independent scalars is an approximation, but the orientation changes little between updates, so the interpolation stays close to the sphere.

A blurred frame or a marker seen at a grazing angle produces a detection that is geometrically valid but wrong, and the innovation y makes it recognisable. The Kalman gain would absorb part of it into the state, so such a measurement is rejected outright when $\|y\|$ exceeds a threshold, and the filter proceeds as if it had never arrived. The threshold is on the norm of the innovation rather than on its Mahalanobis distance, so the gate never widens as the covariance grows. That is harmless for the gripper filter, which re-seeds from forward kinematics on every tick. For the board filter a persistently rejected measurement stays rejected, so a bad initial pose has to be cleared by restarting the filter rather than waited out.

3.3 Deep Learning for Robot Control

The policies in this work are built with LeRobot [11], a robot-learning framework on top of PyTorch [3]. It implements common policy architectures, the training and evaluation scripts around them, and a dataset format for robot data, so a policy can be exchanged without touching the data pipeline.

3.4 The LeRobot Dataset Format

A *LeRobotDataset* is a collection of episodes, and an episode is a sequence of frames recorded at a fixed frame rate. One frame holds the camera images, the robot state and the commanded action for a single instant, together with its episode and frame index, timestamp and task. The task is a natural-language string, which a vision-language-action policy consumes as its instruction (section 3.5). Non-image data is stored in Parquet files, while the images of each camera are encoded as a video stream and decoded on access. Video encoding keeps a dataset small enough to move between machines, at the cost of decoding during training.

A ROS2 bag has no fixed frame rate. Every topic arrives at its own rate and with its own jitter, so messages belonging to one instant are never exactly aligned. Building a dataset from bag data therefore resamples: at each frame

time the most recent message of every topic is taken, and the original arrival times are discarded. Timing accuracy has to be enforced while recording, since resampling cannot recover it afterwards.

3.5 Vision-Language-Action Models

A Vision-Language-Action (VLA) model is a policy architecture that maps camera images, a natural-language instruction, and the robot's proprioceptive state to robot actions. Most VLAs are built on a pretrained Vision-Language Model, which brings visual and semantic knowledge from internet-scale pre-training that robot demonstrations alone could not supply. The actions can be emitted as discrete tokens by the language model itself, or generated by a separate action expert conditioned on the VLM's internal features.

Because the instruction is part of the input, one set of weights can perform different tasks, which the single-task ACT baseline (section 1.2) cannot. Like ACT, most VLAs predict a *chunk* of future actions from a single observation, which amortises inference over several control steps. Chunking generally leads to smoother actions, since the model can predict a trajectory instead of a single action.

3.5.1 SmolVLA and the flow-matching action expert

This work uses SmolVLA [51], a compact VLA designed to be trained on a single GPU and to run on consumer hardware. Its backbone is SmolVLM-2 [39], a VLM that combines a SigLIP [60] vision encoder with the SmolLM2 [8] language model. Only the first half of the language model's layers is used, which halves the compute at a small cost in accuracy. Together with the action expert described below, the model has about 450 million parameters, roughly a tenth of the size of other common VLAs, like $\pi_{0.5}$ [31].

All inputs are mapped into one token sequence for the VLM: the camera images pass through the vision encoder, the instruction is tokenised as text, and the proprioceptive state is projected by a single linear layer into one state token, padded to a fixed maximum width. The policy has no per-sensor structure and never sees which dimensions of the state come from which sensor. On the rover the manipulator's joint state (position, velocity and torque) and the 8×8 ToF depth grid (section 4.8) are concatenated into one flat vector before the projection. The token sequence is shown on the left of fig. 3.5.

That maximum width, `max_state_dim`, is fixed when the projection layer is created, so it is chosen at pre-training for the widest state the model will

ever be given: 128 here, against the 83-dimensional rover state of 64 ToF cells, six joint positions with their velocities and efforts, and the gripper aperture (section A.2).

The action chunk is generated by the *action expert*, a smaller transformer of about 100 million parameters that attends to the VLM's features through interleaved cross-attention and self-attention layers. The expert is trained with flow matching [35]: instead of predicting the chunk directly, it learns a velocity field that transports samples from a noise distribution to samples from the action distribution. For a demonstrated chunk A, a noise sample $\epsilon \sim \mathcal{N}(0, I)$, and the interpolation

$$A^\tau = \tau A + (1 - \tau)\,\epsilon, \qquad \tau \in [0, 1], \tag{3.7}$$

the expert v_θ is optimised to predict the constant velocity of the straight path from the noise sample to the chunk, by minimising

$$\mathcal{L}(\theta) = \mathbb{E}_{\tau,\epsilon}\left[\, \| v_\theta(A^\tau, o_t) - (A - \epsilon) \|^2 \,\right], \tag{3.8}$$

where o_t is the observation encoded by the VLM. At inference an action chunk is generated by drawing a noise sample and integrating the learned field from the noise towards the action distribution, with ten Euler steps in SmolVLA's default sampler. LeRobot stores the negated field and runs its time variable from 1 down to 0, which is the same trajectory under the opposite sign convention.

A mean-squared-error head regressing the chunk directly would predict the average of all action chunks plausible for an observation. Where the demonstrations contain several valid ways to perform a motion, that average can itself be invalid. Sampling from the velocity field reproduces the distinct behaviours instead of their mean, as fig. 3.1 illustrates on a two-dimensional example with two behaviour modes.

SmolVLA is trained purely by imitation, so it can do no better than the behaviour it was shown.

3.5.2 Photometric augmentation

An imitation policy may explain the demonstrated actions by anything constant across the demonstrations, and the appearance of the training scene is constant in them: the objects in it and the lighting they were recorded under.

Against this, only a *photometric* transform is safe for a policy that maps images to actions: it changes appearance alone and leaves features where they were, so the correspondence between what the image shows and where the

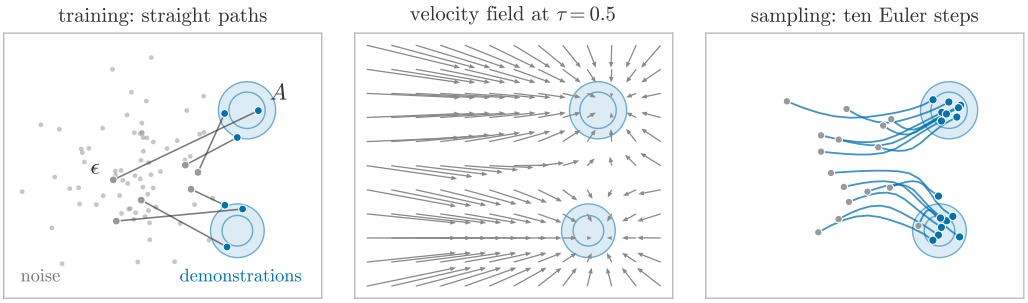


Figure 3.1: Flow matching in a two-dimensional action space with two modes of demonstrated behaviour (blue). Left: training pairs a demonstrated chunk A with a noise sample ϵ; the regression target for the expert is the constant velocity of the straight path between them. Middle: the resulting velocity field, shown at $\tau = 0.5$, points from the noise region towards the action distribution. Right: at inference, noise samples are transported along the field with ten Euler steps; the paths bend into one of the two modes instead of averaging them.

arm has to go is untouched; a geometric one (a crop, a flip, a rotation) moves the target in the image without moving it in the action space, and breaks that correspondence, so they are not used.

The transforms are brightness and contrast jitter over $[0.75, 1.25]$ and $[0.6, 1.4]$, saturation over $[0.8, 1.2]$, hue over $[-0.05, 0.05]$ and sharpness over $[0.6, 1.4]$. The ranges are those of LeRobot's multi-task Diffusion Transformer (DiT) LIBERO recipe rather than values tuned here, so runs stay comparable with published ones.

3.5.3 Advantage conditioning with RECAP

Imitation learning treats every recorded action as a target, but demonstrated actions are not equally good: some move efficiently toward the goal, others are hesitant or mistaken. RECAP [2] keeps all of the data and instead tells the policy which actions are good, by conditioning it on a discrete advantage token. The labelling used here is binary, with one token for each class (`<advantage_positive>` / `<advantage_negative>`). The model then learns to produce good actions when conditioned positive and poor ones when conditioned negative. At inference the condition is fixed to positive. Figure 3.2 sets the mechanism out end to end.

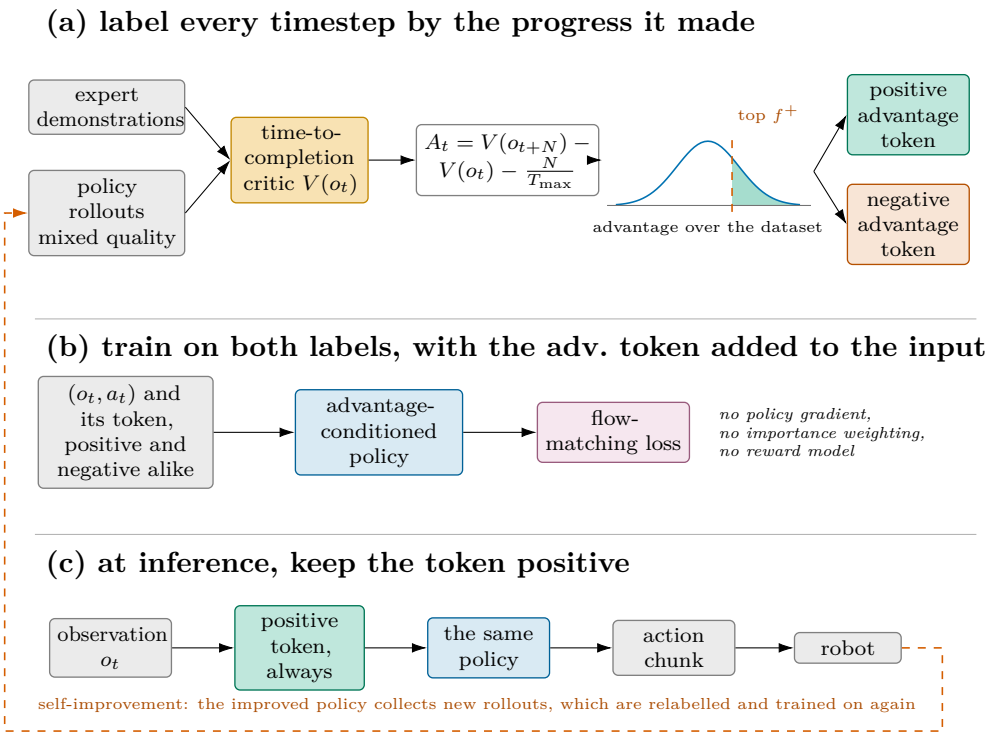


Figure 3.2: Advantage conditioning, the mechanism behind RECAP. Grey marks data, orange the critic, blue the policy, green the positive advantage label and vermillion the negative one. (a) Expert demonstrations and the policy's own rollouts are pooled together. The critic of section 3.5.4 scores every frame, the N-step advantage A_t says whether that step advanced the task faster than the baseline rate, and a percentile threshold, taken over that task's successful frames, splits the timesteps in two: the top f^+ fraction of those is labelled `<advantage_positive>`, and every frame below the threshold, failures included, `<advantage_negative>`. How selective the labelling is depends on the threshold, which is why f^+ is a tuned quantity (section 5.1.1). (b) The label is appended to the observation as one more token, so training is ordinary conditional imitation over the whole pooled dataset. Negative samples are trained on rather than discarded: the positive condition is defined against them. (c) At inference the token is fixed to positive, and the policy generates from the behaviour mode it has learned to associate with progress. Rollouts collected by the improved policy re-enter (a), which is the self-improvement loop implemented in fig. 4.3.

This conditioning turns imitation learning into a simple form of reinforcement learning. The labels carry a reward signal into the training, and because the negative behaviour is learned under a token that inference never sets, the resulting policy can perform better than the average of its training data. Unlike policy-gradient algorithms such as Proximal Policy Optimization (PPO) [49] or REINFORCE [58], the objective stays the supervised imitation loss; only the conditioning input is new, so the method works offline, on data recorded earlier.

3.5.4 Time-to-completion critic

The required advantage labels come from a critic, which in reinforcement-learning terms is a value function. From the task description, camera images and sensor state at a frame it predicts the number of frames remaining until the task is complete. Rather than regressing that time as a single scalar, the critic predicts a categorical distribution over discretised time bins [7] and is trained with a classification loss, following the classification-over-regression argument that stabilises value learning from [22].

The critic is trained for each phase of the RECAP recipe. The first pass runs on the expert demonstrations, where every episode reaches the goal and a frame's time to completion is the number of frames left in it. Such a critic has never seen a failure, and on the policy's own rollouts it is over-optimistic: it scores failed episodes close to successful ones, so the advantages drawn from it cannot separate the two. That is the reason for additional passes over the rollout data. The critic is fine-tuned on the collected rollouts with a pessimistic target for the failed episodes: a large constant is added to their remaining time to completion, which after normalisation clamps their return target to the bottom of the value support, -1. The frames of a failed episode then carry strongly negative advantages (fig. 4.3).

The critic is evaluated on every frame of every episode of the dataset. One step of progress is worth a fixed fraction of the longest episode recorded for that task, $r_t = -1/T_{\max}$, so the value $V(o_t)$ is the negative of the expected remaining fraction of that horizon and lies in $[-1, 0]$. The advantage of the actions taken at frame t is then the N-step temporal-difference estimate

$$A_t = \sum_{t'=t}^{t+N-1} r_{t'} + V(o_{t+N}) - V(o_t) = V(o_{t+N}) - V(o_t) - \frac{N}{T_{\max}}. \tag{3.9}$$

This compares the progress actually made over N steps against the critic's expectation. The advantage is zero where the episode advanced exactly as predicted, positive where the actions brought the task closer to completion

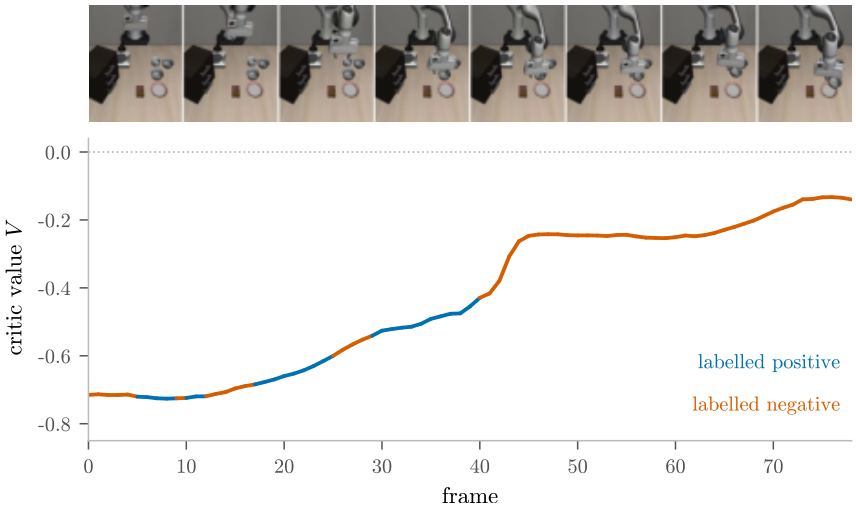


Figure 3.3: The critic's predicted value over one successful rollout episode of the `libero_spatial` task "pick up the black bowl between the plate and the ramekin and place it on the plate", with eight frames of that same episode above it, each sitting over the moment it was taken. The curve is coloured by the resulting advantage label ($N = 50$, against the threshold taken as the 70th percentile of the advantage distribution over the successful episodes of this task. The value climbs from -0.72 to -0.14 as the arm reaches, grasps and transports the bowl, and the positive labels fall in that stretch. After frame 40 it steps up once and then flattens: the task is effectively decided, little progress per step remains, and the whole tail is labelled negative even though the episode runs on for another 39 frames.

than N typical steps would have, and negative where time was wasted. Past the end of an episode the future terms drop out, leaving the Monte-Carlo advantage $R_t - V(o_t)$, in which the pessimistic return of a failed episode holds R_t at -1. A threshold on the advantage yields the binary label, set per task as a percentile of the advantage distribution over that task's successful frames, and precomputed for the whole dataset.

The critic is a training-time component: it exists to produce those labels and is never deployed. The task-progress signal the rover uses to determine the completion of a task comes from the separate and more compact tracker of section 2.7, which is trained in a similar way, hence its categorical head following the same classification-over-regression argument, but it serves a different purpose. Figure 3.3 shows the critic's output over one episode with the labels that follow from it.

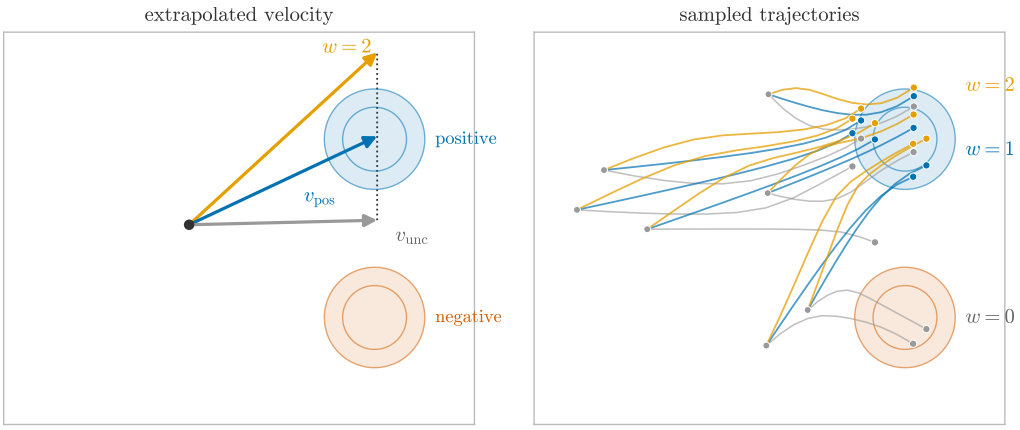


Figure 3.4: Classifier-free guidance in the example action space, with a positively and a negatively labelled behaviour mode. Left: at one point of the integration, the guided velocity extrapolates from the unconditioned velocity v_{unc} past the positive-conditioned velocity v_{pos}. Right: sampled trajectories; without conditioning ($w = 0$) both modes are reached, conditioning on positive ($w = 1$) selects the positive mode, and $w = 2$ pushes the actions further away from the negative behaviour.

3.5.5 Classifier-free guidance

Conditioning on a discrete label also enables classifier-free guidance [27]. During training the advantage token is dropped with a fixed probability, so the same network additionally learns an unconditioned policy. At inference the observation is passed through the model twice per integration step, once conditioned positive and once without the token. This yields two velocity predictions, v_{pos} and v_{unc}, which are combined into the velocity that is actually integrated:

$$v = v_{\text{unc}} + w\,(v_{\text{pos}} - v_{\text{unc}}). \tag{3.10}$$

At the guidance weight $w = 1$ this recovers the positive-conditioned policy exactly, and a single forward pass runs. Any weight other than 0 or 1 costs a second pass through both the backbone and the expert. Larger weights extrapolate along the direction from the unconditioned to the positive field, pushing the generated actions beyond the positive behaviour present in the data (fig. 3.4); the actions become more decisive, and past some weight they overshoot.

3.5.6 Rollout and self-improvement

RECAP closes the loop by letting the policy improve from its own experience. Its rollouts are recorded, successes and failures alike, labelled by a critic trained on them, and used to fine-tune the policy further. The policy thereby sees its own off-distribution states with correct labels, which pure imitation never supplies. The fine-tuning also keeps the original demonstration data in the mixture; without it the policy forgets the skills its rollouts never prompted.

3.5.7 Knowledge insulation

A policy that adds an untrained head to a pretrained VLM couples a backbone that carries the knowledge the architecture was chosen for to an action expert that starts from random initialisation. Since the expert cross-attends to the backbone's features, its flow-matching gradients propagate back into the backbone. Early in training these gradients are large and noisy, and they degrade the pretrained representations, which costs both training speed and the generalisation inherited from the VLM [20]. Freezing the backbone would prevent it from learning anything about the task, and lowering the learning rate enough to protect it would also slow training of the expert.

The resolution is to isolate the backbone from the expert and to add a second training signal for the backbone that does not come from the expert. FAST (Frequency-space Action Sequence Tokenization) [44] supplies it by turning a continuous action chunk into a short sequence of discrete tokens. Within a chunk consecutive actions are strongly correlated, so discretising every value separately would produce long sequences of tokens that each carry almost no information; FAST transforms the chunk into the frequency domain using Discrete Cosine Transform first, so that a few coefficients carry it. The resulting frequency components are turned into a frequency matrix, which is then flattened and encoded with Byte Pair Encoding. The resulting encoding is turned into tokens, which are then added to the vocabulary of the backbone.

The backbone is trained with an autoregressive next-token loss on the FAST tokens of the action chunk, exactly as it was pretrained on text, while the action expert keeps the flow-matching loss. A stop-gradient between the two keeps the expert's gradients out of the pretrained weights, so the expert can train at a normal learning rate. The FAST tokens are only a training signal; at inference the actions come from the flow-matching expert as before. Figure 3.5 collects the policy, its two training signals and the critic that labels its data into one diagram.

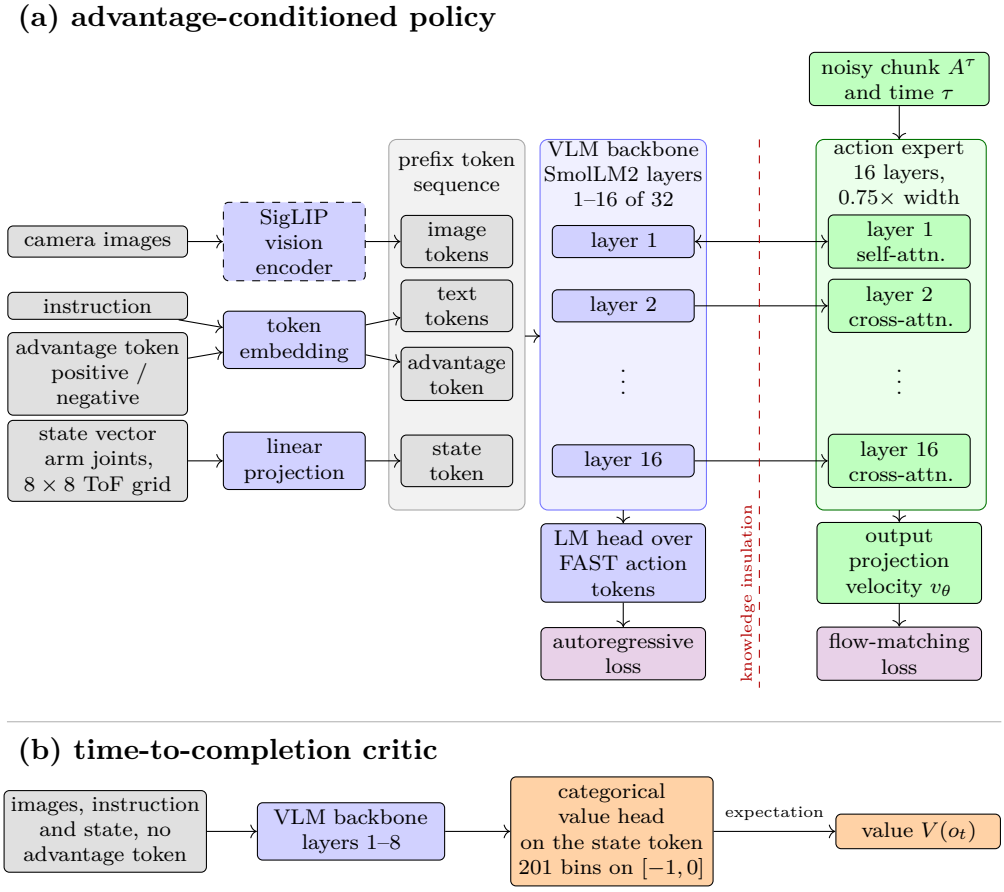


Figure 3.5: The advantage-conditioned policy and the critic that labels its training data. Grey marks inputs and tokens, blue the VLM backbone, green the flow-matching action expert, violet the training losses, and orange the parts that only the critic has. (a) The dashed outline of the SigLIP encoder marks the only part that stays frozen. The expert holds one layer per backbone layer and alternates between two kinds of attention. A double-headed arrow is a layer where the prefix and the action chunk attend jointly. A single arrow is a layer where the expert attends to the backbone's keys and values only. The red dashed line is the stop-gradient of knowledge insulation. Activations cross it, gradients of the flow-matching loss do not, so the backbone is trained only by the autoregressive loss on the FAST tokens of the chunk. (b) The critic reuses the same architecture with the backbone truncated (to eight layers on LIBERO, twelve for the rover lineage) and without the action expert. It reads the backbone output at the state token and predicts a distribution over 201 time bins, whose expectation is the value $V(o_t)$ of section 3.5.4.

3.5.8 SnapFlow: distilling the sampler to a single step

Generating an action chunk from a flow-matching policy means integrating the learned velocity field. Every integration step is a full forward pass through the action expert. On the models SnapFlow reports, the sampler is around 80 % of end-to-end latency [37]; on the engines deployed here the prefix dominates and the ten steps are closer to a quarter (section 5.3). Simply running fewer steps degrades the actions, because a single coarse Euler step along a curved trajectory lands away from the endpoint the fine integration would have reached.

SnapFlow removes this cost by distilling the sampler into a single step, with the model acting as its own teacher. One part of the training mixture is standard flow matching, which keeps the learned velocity field intact. The other teaches the shortcut: the model's own field is evaluated at the noise sample, one Euler step is taken to the middle of the trajectory, and the field is evaluated again. The average of the two velocities becomes the target for a single direct jump, since that average approximates the mean velocity along the whole path. Whether the network should predict the local velocity or the shortcut is signalled through an additional target-time input, encoded by a small zero-initialised multilayer perceptron added to the expert's fused action and time embedding. Its zero initialisation means training starts from a network that behaves exactly like the multi-step policy. Figure 3.6 draws the shortcut on one trajectory.

The student is the same network, so no second model is deployed and the RECAP pipeline above is untouched. The multi-step path is still available. In the SnapFlow paper's evaluation the one-step model matches the success rate of the ten-step sampler on the LIBERO benchmark (section 3.8). The latency gain and the accuracy on the present policy are reported in sections 5.1 and 5.3.

A distillation does not need to target the raw velocity field. *Guidance baking* targets the *guided* one instead, the blend of the positive-conditioned and the unconditioned field at a chosen weight ($w = 1.5$ here), so that a single unguided pass reproduces behaviour that would otherwise cost the second pass classifier-free guidance requires (section 3.5.5).

3.6 Inference Optimisation on the Edge

The policies in this chapter have to run on the rover's Jetson Orin NX (chapter 2). It has limited compute and memory bandwidth, and shares both with the rest of the stack. The control loop needs one action per cycle from

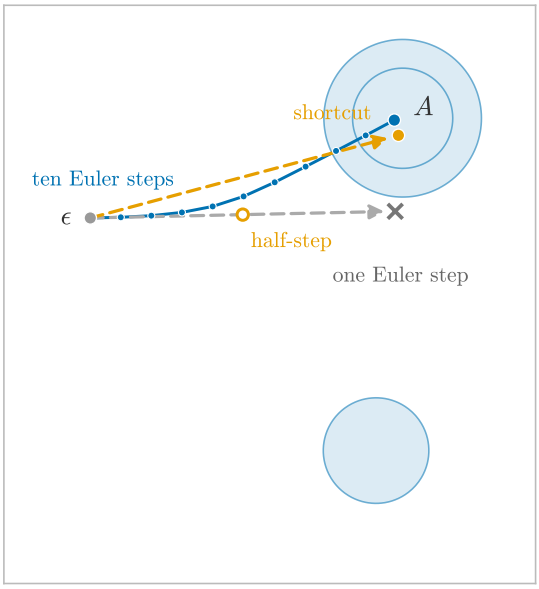


Figure 3.6: The SnapFlow shortcut on one trajectory of the example field from fig. 3.1. The deployed sampler follows the curved path with ten Euler steps (blue). A single Euler step along the initial velocity misses the endpoint (grey cross). Averaging the initial velocity with one evaluated half a step later gives a single jump (orange) that lands close to the endpoint A of the full integration.

the policy. Action chunking already achieves this, since a chunk is replayed over its actions; compilation and reduced precision provide additional margin and a higher rate at which the policy can re-plan (section 5.3).

A policy served directly from PyTorch dispatches each layer separately and writes every intermediate result to memory for the next one to read back. Compiling the network ahead of time for the hardware it runs on removes both steps.

ONNX (Open Neural Network Exchange) [41] is a framework-independent format that stores the weights together with the architecture as a static graph of tensor operations. Being static, it has no representation for repeated or conditional computation: the integration loop of the flow-matching sampler, the second unconditioned pass for classifier-free guidance (section 3.5.5), and the handling of the VLM's key–value cache.

The SmolVLA policy is exported as two graphs, with a small amount of Python code between them carrying the control flow. The *prefix* graph is the expensive VLM pass over images, instruction and state; it fills the key–value cache and runs once per observation, or twice under guidance, since the advantage token sits inside the prefix and the two conditionals cannot share a cache. The *suffix* graph is velocity evaluation of the action expert, run as often as the sampler needs, ten times for the regular policy or once with the distilled sampler of section 3.5.8. The cache passes between them as plain tensors.

The ONNX export is generic, done once, and portable across GPUs. The hardware-specific part is done by TensorRT [54], NVIDIA's inference compiler, which builds the graph into an *engine* for the GPU that is present. The builder fuses chains of operations into single kernels, so that a matrix multiplication, its bias and the following activation become one launch. It also times candidate implementations of each layer on the actual device, keeps the fastest, and plans a fixed memory layout for the activations [40]. That tuning ties the engine to one GPU architecture, so it is rebuilt on the target device.

A floating-point number is a sign, an exponent and a mantissa. The exponent bits fix the *range*, the largest and smallest magnitudes representable at all, and the mantissa bits fix the *precision* within that range, the number of significant digits. Table 3.1 lists the formats TensorRT can build with. The two properties vary independently: FP16 and TF32 carry the same ten mantissa bits and differ in range and in where the rounding is applied, and BF16 was designed for training precisely by keeping FP32's range and giving up mantissa instead.

Inference at batch size one is bound by memory bandwidth, so storing weights and activations in FP16 instead of FP32 roughly halves the memory

Table 3.1: The numerical formats a TensorRT engine can be built with. TF32 is not a storage format: tensors stay in FP32 memory and only the inputs to a tensor-core matrix multiply are truncated, with the product accumulated back into FP32, so a TF32 engine keeps an FP32 datapath everywhere except inside the multiply. FP16 replaces the datapath as well, which is why it changes the range the intermediate activations have to live in and TF32 does not.

format	sign	exponent	mantissa	largest magnitude	scope
FP32	1	8	23	$\sim 3.4 \times 10^{38}$	storage and compute; the reference
TF32	1	8	10	$\sim 3.4 \times 10^{38}$	tensor-core multiply inputs only
BF16	1	8	7	$\sim 3.4 \times 10^{38}$	storage and compute; a training format
FP16	1	5	10	65 504	storage and compute
INT8	8-bit integer with a per-tensor scale			scale dependent	storage and compute

traffic, and on tensor cores the compute time as well. TF32 saves no memory traffic, since the tensors are still FP32, but the matrix multiplications it moves onto tensor cores are where most of the time goes on this hardware.

The accuracy of the conversion depends on the model, and on which of the two properties it gives up. A trained transformer develops rare outlier activation channels whose magnitude sits orders of magnitude above the rest. A format with a smaller exponent has to represent those in the same range as everything else, and spends its representable magnitudes on them at the expense of the rest; this is the phenomenon that motivates the outlier handling in quantisation work [19]. Losing mantissa bits alone is a far milder change, because the outliers stay representable and only lose significant digits. An engine needs to be compared numerically against the full-precision reference with trained weights. On our policy the vision-language prefix runs at TF32. Whole-graph FP16 is rejected, while the action expert can be halved outright under the multi-step sampler (section 4.7.4). With the distilled sampler of section 3.5.8 the expert runs once per chunk, so its precision is worth only a few milliseconds either way and the deployed student keeps it at FP32.

3.7 Simulation

The post-training of section 3.5.3 needs hundreds of episodes per configuration and a repeatable evaluation on top of them, which is a volume of rollouts the real arm could not supply during the time of this project.

A MuJoCo [56] scene of the rover exists, with the manipulator, gripper, ERC panel and the ArUco markers in their real positions, integrated with ROS2 and used for smoke tests and coarse marker alignment. No training environment was built around it.

3.8 The LIBERO Benchmark

LIBERO [36] is a MuJoCo-backed suite of language-conditioned manipulation tasks on a simulated 7-DoF arm, designed to measure how well a policy transfers knowledge across tasks rather than how well it fits one. Each task pairs a scene with a natural-language instruction and a set of human demonstrations. An episode counts as a success only if the task's goal predicate holds at termination.

A configuration is evaluated over 50 episodes on each of a suite's ten tasks, so 500 episodes per suite, and the reported rate is the share of those that succeeded. Every advantage-conditioned configuration is swept over the guidance weights of section 3.5.5 and reported at its best one (section 5.1.5). The convention in this benchmark, and the one the SmolVLA paper follows, is to re-plan on every simulator step rather than to replay an action chunk.

The suite is split into groups that vary exactly one factor at a time. `libero_spatial` varies object placement with fixed objects, `libero_object` varies the objects with a fixed layout, `libero_goal` varies the instruction with both fixed, and `libero_10` combines them into long-horizon tasks. Because the policy is language-conditioned and trained on all groups at once, a per-group success rate separates *where* advantage conditioning helps from whether the average moved.

3.9 Depth Sensing

The gripper's 8×8 ToF sensor reports a grid of distances, and the policy receives it in the state vector of section 3.5.1. A fixed exponential maps each distance to a closeness value, which resolves the last few centimetres, the part that decides a grasp, several times more finely than a linear map over the sensor's range would (section 4.4.1). Nothing in the stack computes a distance to the target from the grid. The cells enter the state vector as 64

more numbers and what the policy makes of them is learned, as it is for the images, though by a different route: the images pass through the vision encoder while the grid is projected with the rest of the state into a single token (section 3.5.1).

A stereo pair on the arm and the ToF grid measure different things. A stereo pair on the arm fixes a target pose metrically from a standstill, because a wide baseline steepens the intersection angle of the two rays, but it is not hardware-synchronised and therefore cannot sit in a control loop. The ToF grid is the opposite: coarse, but continuous and near-field, and it looks exactly where the gripper occludes the stereo pair's overlap. The stereo calibration is in place but unused, because the tasks that would have needed it are driven from taught joint positions instead (section 3.10.4).

3.10 Agentic AI and Workflow Automation

The autonomy stack is orchestrated by two complementary mechanisms built on the same ROS2 nodes: deterministic workflows and an LLM agent. n8n [1] is the harness for both, holding the tool definitions, running the agent loop and connecting the model to the rover. The integration itself is described in section 4.9.

3.10.1 Tool use

A language model produces only text, so it acts on the world through *tool use*, also called function calling: it is given a set of tools, each described by a name and a JSON schema of arguments. It responds with a structured call that a runtime executes, returning the result for the next step. Learning to call tools can be self-supervised [48]. Interleaving such calls with reasoning lets the model plan and act in a loop [59]. Here the tools are the ROS2 primitives (reading a topic, calling a service, starting an action), so that what the rover can do becomes the agent's action space. Perception reaches the model the same way, as tool *results*: an object list from a detector, or a camera frame carried alongside the text as a multimodal block (section 4.9.5).

3.10.2 Exposing tools: function calling, MCP, and skills

Several conventions connect an agent to a system's capabilities: direct function calling, the Model Context Protocol (MCP) [4] as a standardised client–server interface for tools and context, and "skills" as packaged, reusable procedures. MCP and skills were considered; the ROS2 tools are exposed through n8n

instead: the same node can be dragged onto a canvas by a human or called as a tool by an agent, so a single integration serves both. The tools are a package of custom n8n nodes that talk to the running ROS2 graph over a rosbridge websocket (section 4.9).

3.10.3 Agents versus deterministic workflows

An LLM agent is flexible but non-deterministic: it may choose a different sequence of calls on each run. A workflow is fixed and repeatable. Agents therefore take the work where flexibility helps and a wrong step is low-risk: inspection and diagnostics, such as reading several ROS topics, enabling or disabling subsystems, pinging motors, or capturing photos. Deterministic workflows take everything that drives the hardware, moving the base, drilling or operating the panel, where the order of operations and safety must be guaranteed. Unsafe steps are additionally gated on a human confirmation: the workflow pauses until an operator approves.

An agent is bounded first by the set of tools it is given. What an attached tool may do can be narrowed further by restricting the namespace of allowed topics, actions and services. Underneath, all autonomous motion stays under the operator-heartbeat deadman and the software e-stop of section 2.1.4.

3.10.4 Taught positions

A joint configuration is taught once by driving the arm there by hand, stored under a name, and later sent back as a joint target. A task is then a sequence of names, with no pose controller, no camera and no inverse kinematics involved.

The kinematic model is never computed, so the reach-dependent bias of the pose chain (section 5.2.1) does not enter the result. The outcome rests on joint repeatability instead, and repeating a joint target is more accurate than reaching a commanded Cartesian pose.

Taught positions are only correct while the target sits where it sat during teaching, so the method is restricted to objects that are fixed to the rover and a mechanical change invalidates the stored positions.

Where the arm engages a mechanism, the remaining error is absorbed by the mechanism itself: a coupling with a lead-in chamfer converts a small misalignment into a guided one, which lowers the accuracy the software must reach. The engagement is constrained to one axis, which the workflow stores as a short chain of taught positions. Section 5.5.3 describes a tool change built this way.

3.10.5 Small on-device models: vision, function calling, and fine-tuning

The agent's model is exchangeable behind n8n's model interface. In the cloud configuration a commercial frontier model is used, Gemini 3.5 Flash [25] or Claude Sonnet 4.6 [5]. Running the agent onboard instead makes it work offline and removes the per-call cost. It needs a model that fits the rover's compute alongside everything else, exposes native *function calling* in the format the runtime expects (section 3.10.1), and reads images, because the rover's camera frames come back as tool results. One network then covers both the topic-query and the image-reading halves of a diagnostic request. The model chosen, the serving stack and the endpoint it exposes are described in section 4.9.10.

Without task-specific training, small models call tools far less reliably than large cloud models: they often answer in prose instead of acting. Specialising a model on API and tool-call examples improves its call accuracy [42], and even very small models reach competitive tool-calling pass rates after targeted supervised fine-tuning [32]. Retraining all of a model's weights is expensive and risks degrading its pretrained abilities, so Low-Rank Adaptation (LoRA) [29] is used instead. It freezes the base weights and learns a small set of low-rank update matrices, which for the deployed model trains 1.25 % of the parameters (section A.3). Run on the agent's own recorded executions, this teaches the model the rover's specific set of tools, with the aim of approaching the cloud model's call reliability.

Loss and next-token accuracy on held-out trajectories show only that the model imitates the recorded behaviour, not that the tasks succeed. Judging a run asks instead whether the right tools were called and whether the answer is grounded in their results. A stronger model can assess that as a judge [63], or a human can, against per-prompt pass criteria (section 5.5.7).

3.10.6 Memory

State that has to outlive a run needs somewhere else to live. Blackboards [26] and behaviour trees [15] are the established structures for this in robot mission control. The n8n data tables of section 4.9.6 are used as a blackboard in that sense: a small keyed store that workflows and the agent read and write between executions, holding taught positions, task state and the locks that keep two actuating workflows apart. What is not adopted is the behaviour tree, since the ordering it would encode is already the shape of the workflow itself.

Chapter 4

Implementation

This chapter describes what is implemented and why. The classical approach phase comes first: marker layout, the pooled pose fit, the camera-to-base registration, and the alignment controller (section 4.2). Then come the controller switching and the operator heartbeat and software e-stop underneath them (section 4.3), as well as the LeRobot–ROS2 adapter with the progress tracker that stops a policy (sections 4.4 and 4.5).

These contributions then take one section each: the SmolVLA policy and the RECAP training pipeline (section 4.6), the TensorRT deployment on the Jetson (section 4.7), the depth grid (section 4.8), and the n8n workflow and agent layer (section 4.9). The manipulator's inverse kinematics closes the chapter (section 4.10).

4.1 Development Setup

The ROS2 stack runs in containers during development and on the rover. The main repository carries a `.devcontainer` definition, so an editor opens into the same ROS2 Jazzy image the rover runs and a checkout builds the same way on a workstation and on the Jetson.

Rover Autonomy (`ros-fhnw-autonomy`)[1] is the main repository. It holds the classical vision and control of section 4.2 and pulls in two submodules: `smolvla-rl`[2] carries the SmolVLA and RECAP training code of section 4.6, and the *LeRobot ROS2 Adapter* (`lerobot_ros`)[3] bridges LeRobot and ROS2 (section 4.4). It follows the same continuous integration and deployment (CI/CD) conventions as the other ROS2-based repositories of the team, which

[1] `https://gitlab.fhnw.ch/fhnw-rover/autonomy/ros-fhnw-autonomy`
[2] `https://gitlab.fhnw.ch/fhnw-rover/autonomy/smolvla-rl`
[3] `https://gitlab.fhnw.ch/fhnw-rover/autonomy/lerobot_ros`

build, test, and deploy the images directly to the rover. Some jobs of the pipeline are marked as optional, as they are compute heavy and running them on each push is deemed too expensive, especially since the images are built on an NVIDIA Jetson Orin NX, for platform compatibility reasons.

The workflow layer is a Node.js package rather than a ROS2 one and is kept separate. `n8n-nodes-ros2`[4] holds the custom n8n nodes of section 4.9 together with the Docker setup for the container that runs them, and `n8n-agent-finetune`[5] the collection, grading and training pipeline for the on-device agent model (section 4.9.10).

The `lerobot_ros` adapter and the classical vision part come from P8 and were extended for P9; the other three repositories are new. The manipulator's own package (`ros-fhnw-barbara-manipulator`, section 4.10) belongs to the rover's low-level software rather than to the autonomy stack.

4.2 ArUco-based Control

The approach phase brings the gripper close to the target with classical, deterministic vision and a control loop. One base camera sees both sets of markers, and the phase is built on that. The markers on the jaws give the gripper in the camera frame, and against the arm's forward kinematics they fix the camera-to-base transform, so the camera is registered without a separate calibration rig (section 4.2.2). The markers on the panel give the board in the camera frame through a Perspective-n-Point (PnP) solution. Composing the two puts the gripper and every target on the panel in one common frame, and the control law then drives the gripper to a pose expressed in board coordinates (section 4.2.3).

The two halves are treated differently over time. The board is static, so its pose is tracked by vision throughout. The gripper is not: two of its markers have to be visible at once for a trustworthy pose, and at the pose the arm works in they rarely are, so the camera-to-base transform is solved once and forward kinematics carries the gripper from there. The idea is inherited from P8 (section 1.2) and reimplemented with changes in C++ (`barbara_autonomy_cpp`) for P9 to maintain the control rate on the onboard compute.

[4] `https://gitlab.fhnw.ch/fhnw-rover/autonomy/n8n-nodes-ros2`
[5] `https://gitlab.fhnw.ch/fhnw-rover/autonomy/n8n-agent-finetune`

Table 4.1: The two operational marker sets. The size is the edge length of the black square, which the detector returns and the object points must use. A set is identified by dictionary and id range together.

set	dictionary	size (mm)	ids	frame
maintenance panel	`DICT_ARUCO_ORIGINAL`	47	11, 13, 14	board
gripper jaws	`DICT_5X5_250`	25	86–89	tool centre

4.2.1 Markers and marker poses

Two marker sets are in use (table 4.1), in different dictionaries so a detection cannot be attributed to the wrong set. Each detector is nevertheless configured with a dictionary and an id range together, since ids alone are not unique across this robot's tasks.

Panel markers are given as coordinates in the board frame and solved jointly as one board. The pose is accepted only when all three are detected, since a board solved from two is biased by 18 mm to 64 mm; with all three it is stable to 1.1 mm and 0.15°.

The gripper carries four markers, two on each jaw. The two markers of a jaw meet at 90°, one facing along the direction the jaw opens in and one transversely, and the two jaws are mirrored through the tool axis (fig. 4.1), so a camera at roughly 45° to a jaw sees both at a workable angle. The markers sit on the jaws, so the surveyed offsets describe the tool only while the jaws are shut. Opening them drags the reported pose with the marker, which no filter can distinguish from the tool having moved, so the gripper filter gates on the grip signal and drops every sighting taken with the jaws open.

Camera intrinsics. Intrinsics are calibrated from a ChArUco board [24], a 5×7 grid of 38.6 mm squares with 19.3 mm embedded markers, which share `DICT_5X5_250` with the gripper set and are separated from it by id range. The base-camera fit over 206 frames reached a mean reprojection error of 0.246 pixels, with a worst corner at 1.62.

A single square marker has four coplanar corners, so its PnP solution is not unique (section 3.2.1): two poses reproject almost identically, the reprojection error cannot separate them, and the wrong branch is roughly 130° out in orientation, which the lever arm to the tool frame translates into centimetres of position error (fig. 4.2).

The remedy is to stop solving single markers. Whenever two markers of the same tool appear in one frame, their corners go into a single `solvePnP` whose object points are expressed in the tool frame. Eight points with a

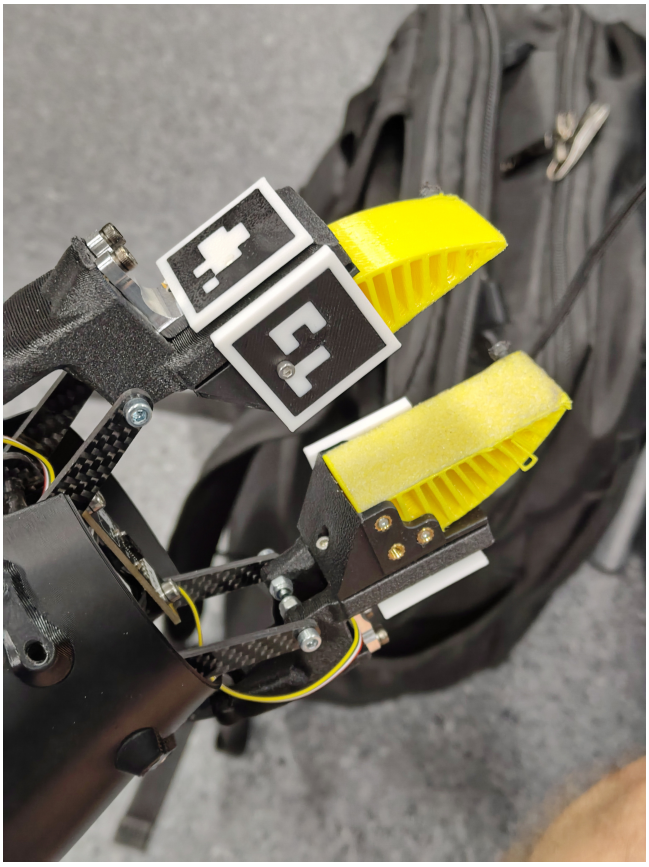


Figure 4.1: One jaw of the gripper, carrying two of the four markers. The two meet at 90°, so a camera roughly 45° off the jaw sees both at a workable angle and neither of them head on. The opposite jaw is built the same way with the two remaining ids, mirrored through the tool axis.

known relative geometry, non-coplanar by 12.5 mm root mean square (RMS), admit no second solution, and the solver returns the tool pose directly with no per-marker offset step afterwards, which is what the panel already gets from its three markers. SQPNP [55] is used rather than OpenCV's [10] default iterative solver, which seeds from a Direct Linear Transform (DLT) estimate that is poorly conditioned on a point set this compact at this range; the two differ only in the tail, at 4.0° against 17.5°.

Pooling needs two markers, so a lone one is kept usable: the detector publishes both solutions and the filter picks between them by attitude against its forward-kinematic prediction, never by position, and drops the sighting altogether when even the better branch is more than 45° away (section 4.2.2).

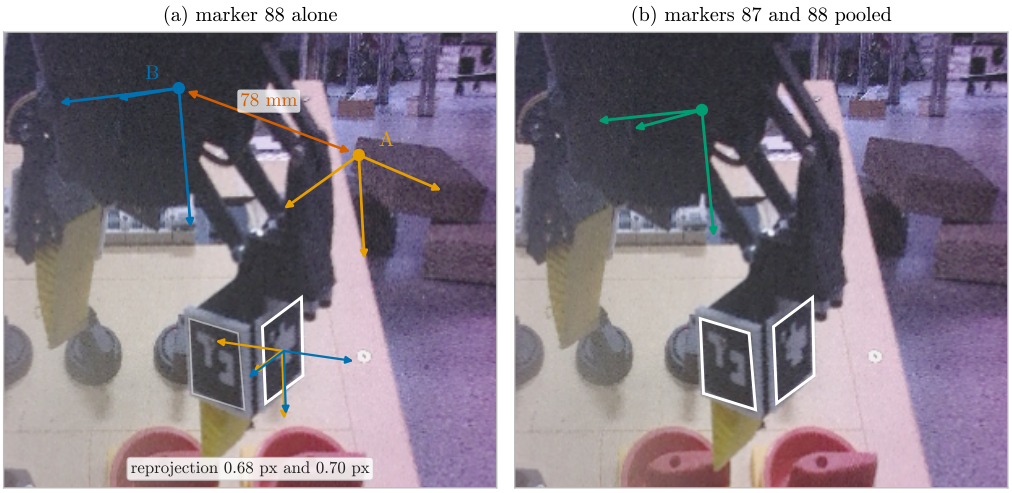


Figure 4.2: The ambiguity and its fix, on one frame of the base camera, which is mounted rotated and turned upright here for display. (a) Marker 88 solved on its own admits two poses. The short arrows at the marker are the two marker orientations, 129.4° apart; the long arrows are the two tool frames that follow from them through the surveyed marker offset. They reproject to 0.68 and 0.70 pixels, so the reprojection error does not choose between them, and the one OpenCV reports as better is the one 65 mm away from the truth. (b) The corners of both markers pooled into one fit. Drawn from the deployed detector and the deployed pooled fit.

Markers already consumed by a pooled fit are not published individually, so no sighting is fused twice.

The marker offsets themselves have to be surveyed. The markers are mounted by hand, so the CAD nominal is several millimetres off and each offset from the tool-centre frame is fitted on the assembled hardware. The chain runs from the camera through a fixed calibration panel and the arm base to the tool frame and on to the marker, over a recording of wrist-roll sweeps at otherwise fixed arm poses; forward-kinematic error is nearly constant within one sweep, so each gets its own panel-to-base transform.

The fit is constrained rather than free, because twenty-four free marker parameters are spent on the out-of-plane tilt, the direction the ambiguity above lives in. The deployed model takes ten shared shape parameters and four in-plane rolls from the mechanical design instead: all four markers lie in one plane perpendicular to the tool axis, the rear marker of a jaw is centred on it, the side marker is offset transversely, and the two jaws are mirrored through the tool axis. The offset along the tool axis is measured rather than

fitted, since the recorded poses do not constrain it: 76.9 mm from the flange to the marker centroid and a further 50 mm to the fingertips, putting the tip at the 126.9 mm the targeting figures of section 5.2.1 are computed with.

4.2.2 Registration and pose filtering

Camera and arm base are both bolted to the rover, so the transform between them is a physical constant and the deployed system treats it as one: it is solved offline from a recording, loaded as a file, and the filter's online update is switched off. What makes that affordable is the fitted kinematic model of section 4.10.2, which carries the tool accurately enough to be trusted between sightings, so no marker has to be visible once the arm is working.

The recording is a deliberate wrist excitation in front of the camera, and three filters decide which of its frames enter the fit: closed-gripper frames only, pooled fits only, and the excitation segment only. The last matters most, taking the disagreement between recordings from 118.9 mm to 18.6 mm. The excitation needs wrist tilt rather than duration, since rolling about the tool axis leaves the axial offset in the null space of the fit, and a registration from 86 frames transfers better than one from 667. Registrations solved from independent recordings agree to 18.6 mm to 26.1 mm and 2.0° to 3.1°, which is what allows one to be stored and reused; re-registration is triggered by mechanics, when the camera or the arm is knocked or remounted, and not by driving.

Both poses are then filtered. Two instances of the same node (`pose_filter_node`) run, one per tracked object, both predicting on a 20 ms timer and differing only in the prediction step (section 3.2.3). The gripper filter fuses the ArUco measurement of the jaw markers, pooled or single (section 4.2.1), with a forward-kinematics estimate from the joint states. The same detection and pooled fit feed the offline registration, which is what the rover deploys; the online fusion below is implemented and available, and is not in the deployed path. Its prediction replaces the state outright with the kinematic pose of the tool, which keeps the estimate available when no marker is seen, and a measurement too far from that prediction is rejected rather than fused. The board filter has no kinematic source, so its prediction only inflates the covariance and it averages over roughly fifty measurements, which a filter tracking a static object can afford. The known switch and socket coordinates then place every target relative to the filtered board pose.

Both filters broadcast their result as a transform, and the alignment controller servos on those two frames. The camera-to-base transform is a third quantity the node can refine online; that update is disabled and the transform loaded as a constant (section 4.2.2).

4.2.3 Target selection and alignment

A dedicated detector (`board_object_detector_node`) locates the panel's interactive elements with classical OpenCV colour thresholding and contour detection, since the elements are simple, coloured, and seen against a known background. It is specific to the ERC maintenance panel, recognising the element types by a slug (`fi-switch`, `black-switch`, `red-switch`, `socket`). The detector node provides a ROS2 service for retrieving the detected positions using these slugs. The positions come out in the board frame recovered in section 4.2.2, so a detected element is directly usable as the goal pose for the alignment controller below. The workflow layer reaches both services over the rosbridge (section 4.9) and can feed the output of the detector to the aligner.

With the board and gripper poses in one frame, the relative controller (`relative_controller_node`) servos the gripper to a target pose expressed relative to the board, as a proportional velocity controller at 20 Hz. Position error is scaled by a gain into a clamped linear velocity. Orientation is held constant until the position error falls below 3 cm, at which point the orientation controller engages with its own clamp, so the gripper approaches in a near-straight line and reorients only near the end, avoiding sweeping motions that could catch on the panel. The correction is computed in the gripper frame and transformed into the base frame. Alignment completes when position and orientation are both within tolerance, 5 mm and 0.05 rad, at a gain of 0.8 on both and clamps of $0.06\,\mathrm{m\,s^{-1}}$ and $0.4\,\mathrm{rad\,s^{-1}}$.

The controller does not send a velocity to the arm. It publishes a Cartesian target pose to the motion controller of section 4.10, and each cycle's step accumulates onto the previous target rather than onto the measured pose, so that a lagging arm widens the gap the controller pulls against instead of holding it fixed at one step. The accumulated target is leashed to 8 cm and 1 rad from the measured pose, the same leash the teleoperation path puts on its integrated target.

Alignment is exposed as the `align_to_target` action, whose goal names the target frame, the target pose within it and the source frame that is servoed. The feedback streams the remaining distance and the full pose error, so the workflow layer can watch an approach and branch on how it ended (section 4.9.8).

4.2.4 Implementation notes

P8's own operator web interface was not rebuilt, because tools that exist for other reasons cover the same ground: the team dashboard for watching and for

the software e-stop, over `rosbridge` [18], n8n for commanding (section 4.9), and the annotation GUI for recording (section 4.4.4).

The stack has to run within the 20 Hz control budget on the onboard compute, so the performance-critical path was moved out of Python. The ArUco detection, the two Kalman pose filters, the relative controller and the panel object detector were reimplemented in C++ (`barbara_autonomy_cpp`). The ArUco detection was previously two node instances that each decoded the same camera frame. It was merged into one node that decodes and converts to greyscale once per frame and shares the result, so only the dictionary-specific marker detection still runs twice. Incoming camera images are decoded on the GPU where hardware JPEG decoding is available (NVJPEG on the Jetson, through `torchvision`'s `decode_jpeg` on CUDA), with a CPU fallback otherwise.

Most of the code in this section was written by an AI coding agent under the author's direction.[6] That worked because the agent was given recorded rosbags to test against instead of the robot, so it could check its own attempts, as in the export pipeline of section 4.7.5.

Every validation script in the calibration toolkit drives the deployed nodes' own logic, configured from the file the rover loads, over a recorded bag, so a change to the detector, the pooled fit, or the filter is evaluated by re-running the same recordings, and each bag is processed twice in one run, with the change and without, so the two sides of a comparison share frames, detections and solver settings. Trying each change on the hardware instead is slow, occupies a machine the whole team shares, and confounds the measurement, since the arm's state changes between sessions. Because a bag only answers what its content covers, the recorder scores the number of stations, the spread of wrist attitudes and the two-marker rate live, and reports when the coverage is sufficient.

4.3 Controller Switching and the Operator Heartbeat

Both autonomy phases drive the same arm, and neither may do so unsupervised or simultaneously. Three mechanisms enforce that: a heartbeat the operator has to keep publishing for any autonomy to actuate, a latching software e-stop that aborts what is running, and an exclusive claim on the `ros2_control` controller a phase commands through. The first two implement the layered stopping requirement of section 2.1.4; the third is what

[6] Claude Code; see the Index of Assistive Technologies.

makes the hand-off from approach to manipulation a property of the actions themselves rather than of an external sequencer.

4.3.1 Operator heartbeat

The autonomy controllers actuate only while an operator heartbeat is present, the policy controller and the relative controller alike; a predefined joint move through `move_to_joint_positions` is not gated, for the reason given in section 2.1.3. The operator publishes on `/policy_control/heartbeat`, and a controller that has not seen a message for 0.5 s stops publishing commands, so the arm holds position. The goal stays active and reports that it is waiting, which distinguishes a paused run from a failed one. After a longer gap of 5 s the buffered trajectory is flushed, so that on resume the controller re-predicts from the current observation rather than replaying stale actions. Both timeouts are configuration, and setting the first to zero disables the deadman for bench testing. This is the mechanism behind the sub-second disable requirement of section 2.1.3.

4.3.2 Software e-stop

Alongside the heartbeat the autonomy nodes share a latching software e-stop, implemented once and used by both controllers. Any message on `/e_stop` trips the latch: the active goal is aborted, the node stops publishing so the arm holds position, and new goals are rejected until `/e_stop/reset` clears it. The latch is what makes it an e-stop rather than a pause, and it is checked before the heartbeat, so a trip aborts the goal even while the controller is already paused waiting for the operator. Blocking joint moves through the `move_to_joint_positions` service are the one motion this cannot interrupt (section 2.1.3) and remain covered by the hardware stop. The latch semantics and the goal rejection are covered by unit tests in both packages.

4.3.3 Controller switching

The two phases command the arm through different `ros2_control` controllers: the relative controller publishes a Cartesian pose to the Cartesian motion controller, whereas the policy controller writes joint position commands to the position controller. Only one may be active at a time, so handing the arm from the approach to the manipulation phase means activating one controller and deactivating the other. This is done in code rather than by an external orchestrator: a shared helper, one implementation per language, acquires the controller a node needs when its action starts and releases it

when the action ends, deactivating a configured list of mutually exclusive ones in the process. The hand-off is therefore implicit rather than sequenced from outside, and a node cannot command the arm without having claimed its controller first: `align_to_target` releases the Cartesian controller when it finishes, and `run_policy` activates the position controller when it starts. The policy controller also publishes its state as a latched `PolicyStatus` message (section 2.5.2) so the operator interface and workflow layer can see which policy, if any, is in control.

4.4 LeRobot Adapter for ROS2

The adapter connects the LeRobot framework of section 3.3 to ROS2. It subscribes to the topic set declared in the TOML configuration (section 2.9), holds the latest message of every topic, and assembles one frame from them on a fixed-rate timer, which is the resampling the two data models require (section 3.4). A topic that has not published yet contributes a zero tensor of the right shape, so a frame always has the same layout.

Each message type has its own converter to a tensor, declared once and looked up by type. Non-image topics are concatenated into one vector and split into observation and action by a per-topic tag from the configuration; images stay separate features. The same frame assembler serves the recorder, the offline conversion, and the policy controller, so a dataset and a live observation have identical layout by construction. That property is the point of the design: nothing has to be kept in step manually between recording and inference.

The adapter is inherited P8 infrastructure (section 1.2). The P9 additions are the value constraints and input transforms described next, the offline conversion path (section 4.4.3), and the annotation GUI (section 4.4.4).

4.4.1 Value constraints and input transforms

The `limits` and `round_values` options of section 2.9 constrain a predicted action on its way back into ROS messages. Every element is clamped into its allowed range and then snapped to the nearest permitted level, in that order, so rounding cannot move a value back out of the safe range. This is where a policy's output is made safe to actuate, and it is configuration rather than code: a joint can be given its own range without the rest of the vector being spelled out.

A `transform` entry additionally applies a monotonic elementwise map as a message is converted. Two are implemented, both turning a distance into

a closeness: an exponential decay $\exp(-d/s)$, bounded in $(0, 1]$ and equal to 1 at contact, and a reciprocal $s/(d + \varepsilon)$, which resolves the near field more aggressively but is unbounded as d approaches zero. The ToF grid is the case they were added for (section 4.8), and the deployed configuration uses the exponential at $s = 800\,\text{mm}$. What a transform does is redistribute the value range, which the training pipeline's affine normalisation cannot: over the sensor's four-metre span the exponential gives the first five centimetres about five times the numeric range, and that is the part which decides a grasp. It is applied inside the conversion itself, so every consumer of a frame sees the same values without opting in. Transforms are rejected on action topics, since they have no exact inverse for publishing back to ROS.

4.4.2 Running a policy

A policy is executed through the `run_policy` action (section 2.5.1), whose goal names the loaded policy and the task string. Inference and actuation are decoupled by a shared action queue: one side runs the policy and appends the predicted chunk, the other publishes one action per control period. The policy therefore does not have to keep up with the command rate, only with the rate at which the queue drains, which is what makes a chunked policy deployable at 20 Hz on this hardware (section 5.3). Where a new chunk overlaps actions still queued, the two are blended with a weight that decays across the chunk (the P8 mechanism, kept unchanged), which keeps the arm from jumping when a fresh prediction disagrees with the one in flight. Constraints are applied on the way out (section 4.4.1).

The queue is also where the two stop conditions differ. Nothing is published while the e-stop is latched or the heartbeat is stale (section 4.3), and the queue is not advanced either, since that would drop the head of the trajectory and resume a step further along. A tripped e-stop then clears the queue outright, because an e-stop is an abort, while a stale heartbeat keeps it, so a short lapse continues where it stopped. The same distinction is made at goal level: the e-stop aborts the goal, a stale heartbeat leaves it running and merely reports the wait.

A goal ends as `completed` when progress crosses the completion threshold, as `timeout` at the configured episode length, or as `cancelled` by the operator. On every exit the controller stops publishing, clears the queue so a later run cannot replay stale actions, and releases the controller it acquired at the start (section 4.3). It publishes no terminal command: the actions are joint positions, so a zero command would order a jump to the origin instead of a stop, and leaving the position controller at its last setpoint holds the arm where it is.

4.4.3 Offline dataset creation from rosbags

Besides recording a *LeRobotDataset* live, the adapter can build one offline from an existing ROS2 bag. This decouples data collection from dataset creation: raw bags are recorded in the field with minimal overhead and converted afterwards, and bags recorded for other purposes can be reused. The tool takes the same TOML file the nodes use and deserialises with the same converters as the live path, so a bag is read as the running system would have read it and datasets created either way are consistent (section 3.4). Frames are emitted on the target clock from whatever messages are currently held, so a slower topic is repeated and a faster one decimated, and a gap needs no special handling because the previous message is held for longer.

Episodes come from a control topic carrying the task name: an episode starts when a name is published, ends when the topic goes empty or names an excluded idle or reset state, and is split when the name changes. A bag without such a topic can be converted as a single episode. Conversion runs faster than real time and is deterministic, since it never plays the bag back through ROS2.

4.4.4 Annotation GUI

Bags recorded outside an autonomous run carry no task topic, so their episode boundaries have to be set by hand, and the `annotation_server` node serves a browser GUI for that (fig. 2.5).[7] It wraps the same configuration and converters as the conversion tool, so what the operator sees in the browser is what the dataset will contain. Seeking returns the camera images at the requested instant together with the held value of every other topic, which makes it useful for inspecting a bag as well. The marked segments are stored next to the bag and handed to the conversion.

4.5 Task Progress Regressor

A policy never signals that it is finished, so something outside it has to decide when the switch has been turned and the arm should stop. P8 answered this with a progress regressor, and P9 keeps the idea and replaces the model.

The estimator predicts how far through the task the episode is, as a number in $[0, 1]$. Each image of a window of recent frames goes through a MobileNetV3-Small [28] backbone, frozen at its ImageNet weights, and

[7] This tool was written by AI coding agents under the author's direction; see the Index of Assistive Technologies.

one projection, shared across the cameras, reduces each camera's features to a small vector. Those are concatenated with the robot state and the last action, both normalised by statistics stored on the model itself, since the raw channels span about five orders of magnitude between joint efforts and ToF distances. A ToF cell can sit at one reading through a whole training set, and such zero-variance channels are excluded from normalisation rather than divided by the usual small floor, which would blow the channel up the moment it moves at inference. The sequence runs through a gated recurrent unit (GRU) and a linear head reads the final hidden state.

The head does not regress a scalar. It predicts a distribution over 101 progress bins, trained with HL-Gauss targets [22], and progress is read back as the expectation of that distribution. Progress is not monotonic, since a grasped object can be dropped, and a distribution can represent that ambiguity where a single regressed number has to average it away. This is the same argument that puts a categorical head on the RECAP critic (section 3.5.4).

The window is strided: four frames spaced ten apart span 1.5 s at no extra decoding cost, where four consecutive frames at 20 Hz would reach back 150 ms, too little to see a task progress or regress. The stride belongs to the trained model rather than to the call site, since feeding such a model consecutive frames hands the recurrent layer a tenth of the history it was trained on.

At runtime the tracker is loaded by the policy controller alongside the policy it belongs to and fed from the same converted frames, so it costs no second subscription. Its window never reaches back into the previous episode, and its estimate is published alongside the rest of the controller state (section 2.5.2). Crossing the configured completion threshold, 0.9 by default, ends the `run_policy` goal as completed. A policy configured without a tracker is warned about at startup, because the timeout is then its only termination condition. Like the policy, the tracker can be exported to TensorRT and loaded from an engine instead (section 4.7).

4.6 SmolVLA and Reinforcement Learning

Where the previous project trained a separate ACT policy per task, this work integrates SmolVLA [51], a Vision-Language-Action model that is conditioned on a natural-language task description and can therefore address multiple tasks with a single set of weights. It loads through the same policy-controller interface as ACT, with the policy type selected via configuration, which keeps the deployment path unchanged.

4.6.1 Policy implementation

The policy subclasses LeRobot's stock SmolVLA rather than reimplementing it (fig. 3.5), so it loads and runs through the same path as any other LeRobot policy and the deployment side needs no special case. The flow-matching sampler, the joint VLM–expert backbone, and the image and state preprocessing are inherited unchanged. Three things are added:

- **Advantage conditioning.** A positive and a negative advantage token are added to the VLM vocabulary and appended to the language tokens, so the same weights can be asked for good or for bad behaviour at inference.
- **FAST tokenizer and Knowledge Insulation.** FAST action tokens are mapped into the VLM vocabulary for the autoregressive loss, and Knowledge Insulation (section 3.5.7) is enforced by cutting the gradient path, so the flow-matching loss never reaches the VLM.
- **SnapFlow head.** A zero-initialised target-time head is added to the suffix, so one network can run either the multi-step sampler or a single denoising step (section 4.6.3).

The critic is a separate subclass of the same base. It reuses the VLM–expert backbone with the VLM frozen and adds a categorical value head over time-to-completion, whose predicted $V(o_t)$ the advantage computation builds on. It is a training-time component and is never loaded on the rover, where the progress signal comes from the separate tracker of section 4.5. The concepts behind these components are covered in section 3.5.

4.6.2 Training pipeline

Training runs as a sequence of separate stages rather than one end-to-end loop, following the RECAP phases of section 3.5.3. Each stage is one script, run by hand and passing its output to the next as files on disk, which is what lets a stage be re-run or replaced without touching the others. Figure 4.3 shows the stages and what each one carries. Four of them train a network:

- **Critic training** by supervised regression of time-to-completion. It runs twice. The first pass fits the critic to the expert demonstrations and supplies the labels the base policy is pre-trained on. The second runs against the rollout dataset, pushing every failed episode's return target to the bottom of the value support. Without it, the critic, having seen only successes, scores a failed rollout almost as highly as a successful one,

and the advantages computed from it cannot tell them apart. The fine-tuned critic is the one whose values everything downstream is labelled with. The rover lineage runs the first pass only, since it collects no rollouts to fine-tune on, so the policy deployed on the arm is conditioned on labels from a critic that has only ever seen demonstrations.

- **Advantage labelling** runs the fine-tuned critic over the dataset and computes an N-step temporal-difference advantage for every frame ($A_t = (R_t - V(o_t)) - (R_{t+N} - V(o_{t+N}))$, horizon 50). A per-task threshold at a chosen percentile then splits the frames into positive and negative, by default keeping the top 30 %. The labels are written to disk, so training does not need the critic to run.

- **Policy fine-tuning** on the labelled data, optimising the flow-matching and FAST autoregressive losses together with Knowledge Insulation in place. It covers both the imitation and the rollout phase, and can mix demonstration batches into rollout data to avoid forgetting.

- **Distillation** of the converged multi-step policy into the single-step SnapFlow variant (section 4.6.3).

Rollout episodes for the self-improvement phase of section 3.5.6 are recorded by a separate evaluation script. It supports human intervention through a gamepad or keyboard, but the runs reported in section 5.1 do not use it: every rollout was collected autonomously and labelled by the critic alone.

The architecture is the same for LIBERO and the rover: the 450M-parameter SmolVLA with a 16-layer VLM backbone, and a critic sharing that architecture with the backbone truncated to eight layers on LIBERO and twelve for the rover lineage, predicting the remaining time to completion over 201 bins. Advantages use a temporal-difference horizon of 50 frames on LIBERO and 20 on the rover lineage, and the advantage token is dropped for 30 % of the samples so that an unconditional predictor is learned alongside the two conditionals, which classifier-free guidance needs at inference.

The LIBERO configuration of section 5.1 pre-trains the base policy for 250k steps on the LIBERO expert demonstrations, with knowledge insulation, FAST co-training and advantage conditioning already in place; each fine-tuning run takes 20k steps and distillation 15k. The policy deployed on the rover is not pre-trained on rover data: it is pre-trained for 120k steps on the `berkeley_autolab_ur5` manipulation dataset [14] and then fine-tuned for 30k steps on the recorded dataset, each stage with its own critic (section A.2). The flow-matching loss is weighted 4.0 against 1.0 on the autoregressive one at pre-training, where the flow-matching expert starts from a random initialisation,

and the two are weighted equally at LIBERO fine-tuning; the rover fine-tune keeps the 4.0-to-1.0 weighting. The photometric augmentation of section 3.5.2 is a training flag, off for the LIBERO configurations of section 5.1 so that they are affected identically and on for the rover training since it recovered the object suite (section 5.1.4) and should make the policy more robust to changing environments. Training runs on the FHNW high-performance computing (HPC) cluster under SLURM (Simple Linux Utility for Resource Management): about 18 h on two NVIDIA H200 GPUs for pre-training and 90 min on one for a fine-tuning run. The commands for every stage are in section A.1.

4.6.3 SnapFlow distillation

The flow-matching action expert generates an action chunk by integrating an ordinary differential equation over several denoising steps, and running the full multi-step sampler at every inference is a significant part of the latency on the target hardware. As an optional final stage of the pipeline, the multi-step sampler is distilled into a single-step generator, which the runtime loads as a policy of its own. Because the policy emits a chunk of actions, the 20 Hz command rate is already met by replaying that chunk; the single step buys a higher re-planning rate, so the policy reacts sooner to a change in the scene. The theoretical background is introduced in section 3.5.8; the latency benefit is quantified in section 5.3 and the effect on task success rate in section 5.1.

The student is distilled from the ten-step teacher for 15k steps, on the same data mixture the teacher was fine-tuned on: critic-labelled rollouts with demonstration batches at a ratio of 0.5. The rollout frames keep their critic labels, since forcing them all positive would make the flow-matching term imitate advantage-negative actions under the positive token. Training combines that flow-matching term, anchored on the training data, with a distillation term that matches the single-step jump to the teacher's full integration, and the student is initialised from the teacher so distillation starts from its behaviour. Every distilled policy is trained for 15k steps because distilling longer hurts: extending an early variant to 120k steps kept lowering the distillation loss while closed-loop success dropped by about ten points. The student is validated closed loop, by running it in the benchmark against the teacher (section 5.1).

4.6.4 Simulation environment

Reinforcement Learning is carried out in simulation, for the reason given in section 3.7: the post-training needs hundreds of episodes per configuration

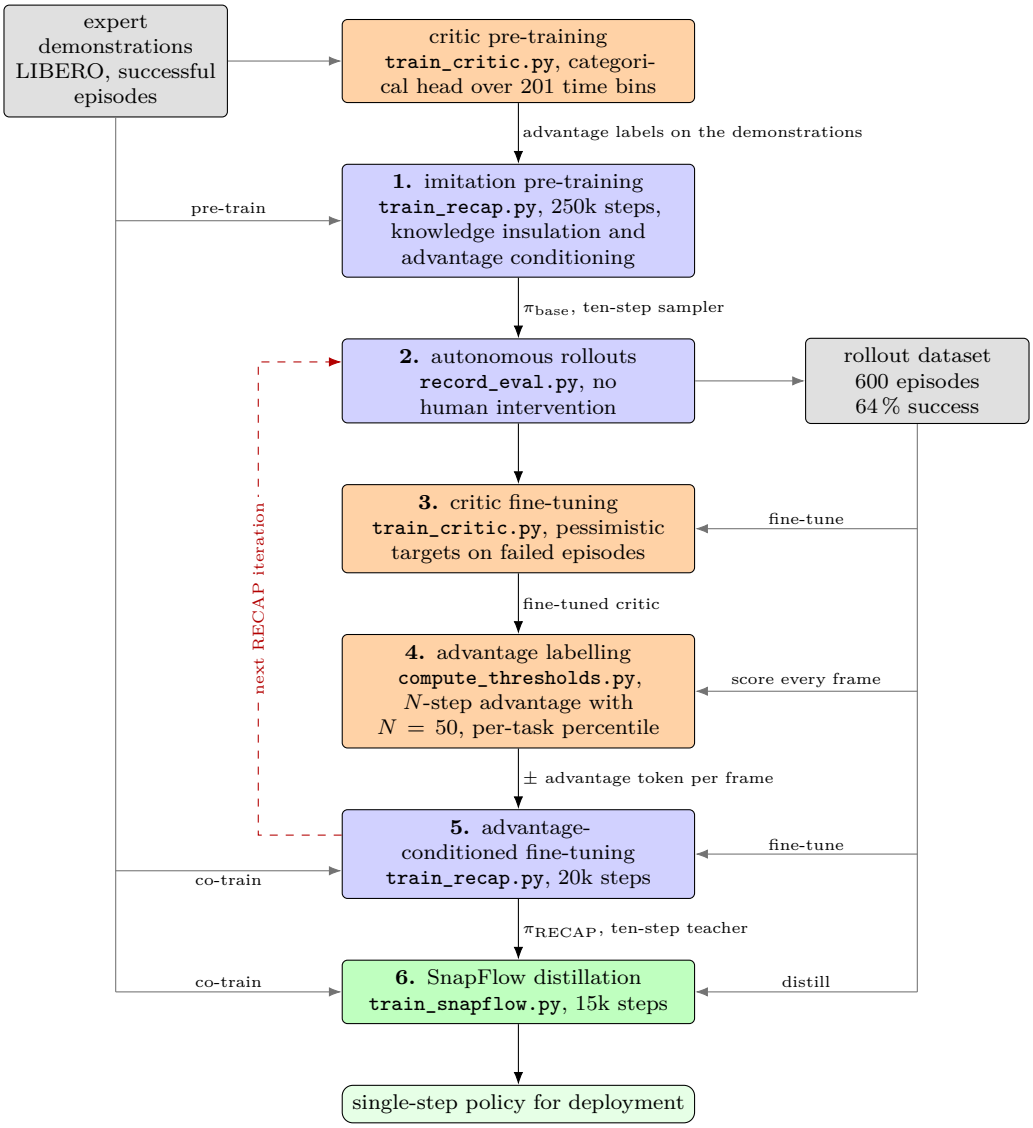


Figure 4.3: The RECAP training pipeline as it is implemented in the `smolvla-rl` package. Blue marks the stages that train the policy, orange the stages that train or apply the critic, green the distillation, and grey the two datasets. Each stage is one script, named inside its box, and the label on an arrow names what that arrow carries. The critic pre-training at the top is not numbered, because it only supplies the labels that stage 1 conditions on. Expert demonstrations re-enter stages 5 and 6 as co-training batches, in which every frame is labelled positive. The red dashed arrow is the self-improvement loop of section 3.5.3: the fine-tuned policy collects new rollouts and stages 2 to 5 can repeat. One pass around it was run, on LIBERO; the rover chain never ran the rollout stage at all (section A.2). The step counts and the rollout statistics are those of the LIBERO configuration reported in section 5.1.

and a repeatable evaluation on top of them. The implemented pipeline uses the **LIBERO** benchmark [36] (MuJoCo-backed) through LeRobot's environment factory, with the task suites `libero_spatial`, `libero_object`, `libero_goal` and `libero_10`. LIBERO was chosen over a rover-specific Gazebo or MuJoCo scene for two reasons. It ships with expert demonstrations, task suites and an evaluation protocol, so the training loop could be built against a working environment rather than against a simulation that had to be built and validated. And it is published, which makes a result comparable rather than self-referential. The cost is stated in section 5.1: the benchmark result is a LIBERO result, obtained on the LIBERO robot rather than on the rover manipulator. The pipeline is not confined to it, since most of the same stages are run again on rover demonstrations to produce the deployed policy (section A.2). The exception is rollout collection, which needs arm time, so only the LIBERO side runs the self-improvement loop.

A rover-specific MuJoCo scene exists alongside it, modelling the manipulator together with the ERC panel and carrying the ArUco markers in their real positions. Built while the arm was still being fabricated, it is where the training and deployment pipeline was first smoke-tested end to end and where the detection, pose estimation, and alignment loop of section 4.2 were first exercised. It is not a reinforcement-learning environment and no policy was trained in it for deployment.

4.7 Deployment on the Jetson Orin with TensorRT

The policies are optimised for the onboard Jetson Orin with TensorRT [54, 57], along the ONNX-to-engine route of section 3.6, which buys margin in the control cycle and re-planning rate (sections 4.6.3 and 5.3). The conversion is implemented for all three policies (ACT, SmolVLA, and SmolVLA+RECAP+SnapFlow), and the runtime wrappers expose the same chunk-prediction interface as the PyTorch policies, so an engine is a drop-in replacement.

4.7.1 Two stages across three machines

Only the engine build depends on the target machine (fig. 4.4): it times kernels on the device it runs on, so an engine built on a desktop card (SM_86) will not load on the Jetson (SM_87). The policy is therefore trained on the HPC cluster, exported to ONNX on a desktop GPU, and optimised into an engine on the rover. What is copied to the Jetson is a tarball of the platform-independent ONNX files, together with a rebuild script and

the configuration snippet that switches the policy over to the engines. The rebuild runs once, when a policy is installed, so the build cost is paid ahead of deployment rather than at start-up.

4.7.2 Engine contents

An engine holds a single forward pass with fixed input shapes. That one constraint decides how each policy is split: everything that loops, samples, or branches on the data has to stay in Python. ACT has no sampler and no data-dependent control flow, so the whole policy becomes one engine. SmolVLA is exported as the prefix and suffix graphs of section 3.6, with its Euler loop left in Python calling the suffix engine once per denoising step. The split is also a necessity, since the VLM's key–value cache is a dynamic Python object that cannot be serialised to ONNX. Exporting the VLM at all required graph patches to remove in-place operations the TensorRT parser rejects.

SmolVLA+RECAP+SnapFlow reuses those same two engines, calling the suffix once, since the distilled sampler jumps to the endpoint in a single step (section 4.6.3). Classifier-free guidance is also kept outside the engine: the prefix and the suffix are each run twice, once with the advantage token masked out and once with it active, and the two velocity fields are blended on the host. The alternative, an engine built for a doubled batch, would pay the prefix twice even when guidance is switched off, and the prefix is where nearly all of the chunk is spent. The progress tracker of section 4.5 is exported in the same style, with the backbone in the engine and the categorical read-out in Python. Fixed shapes also mean the language tokens and the state vector are padded to a constant width, identically in the exporter and the runtime wrapper, so the engine always receives the shape it was traced with.

4.7.3 The acceptance check

Correctness is checked by running the PyTorch and TensorRT policies on the same fixed-seed inputs across a set of samples and reporting the mean and maximum absolute error against the reference. An action dimension the adapter rounds to a fixed set of levels (section 4.4.1) is held out of those statistics and checked by whether it lands on the same level. A maximum absolute error is meaningless on such a channel: the engine only has to sit on the other side of a rounding boundary once to register an error the size of the step between levels.

ACT and the progress regressor are accepted on the maximum absolute error over the continuous dimensions, below 1×10^{-2} in FP16 and 5×10^{-3}

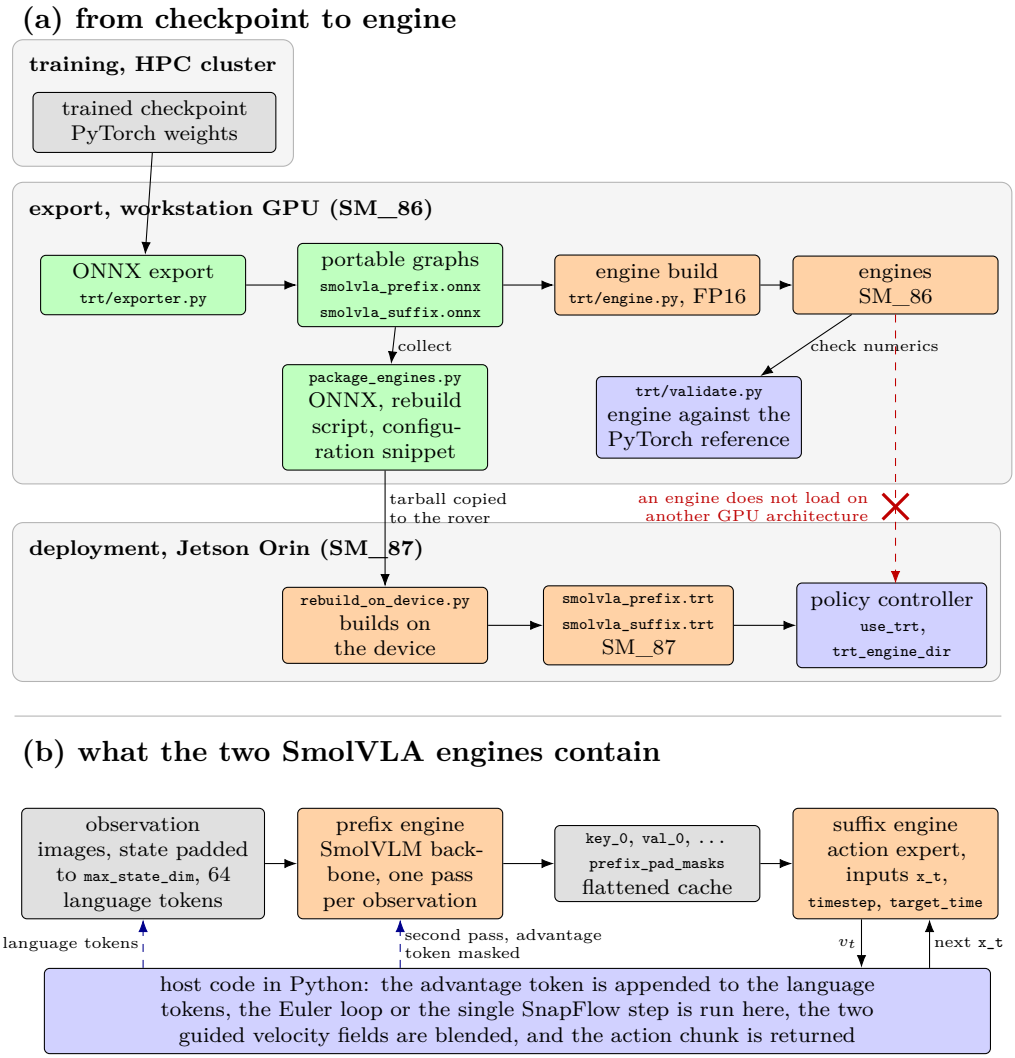


Figure 4.4: The TensorRT conversion pipeline and the SmolVLA engine split. Green marks what is platform independent, orange what is tied to one GPU architecture, blue the code that stays in Python, and grey the checkpoints and intermediate tensors. (a) The three shaded bands are the three machines the deployment spans. The ONNX export is a property of the model alone. The engine build times kernels on the device it runs on, so its result belongs to one architecture. The red cross is the transfer that does not work, and the engines are therefore built a second time on the rover. (b) An engine holds one forward pass with fixed input shapes. The prefix engine runs the VLM once per observation and returns its key–value cache as plain tensors. The suffix engine evaluates the action expert on them, once per Euler step, or once for the distilled policy of section 4.6.3.

in FP32. SmolVLA and RECAP are accepted on the mean absolute error instead, below 2×10^{-2} in FP16 and 1×10^{-2} in FP32. The maximum was dropped for them because the sampler integrates over ten Euler steps, where a small difference in one velocity estimate can carry a single sample onto a different trajectory while the remaining chunks agree closely. That is a genuine weakening of the criterion, and its origin is discussed in section 4.7.5.

A validated engine is only correct for the checkpoint it was built from, and it is easy to lose track of that on a robot where engines are copied between machines. The normalisation buffers are traced into the ONNX graph, so an engine built from a different checkpoint stays loadable and looks healthy while computing with the wrong statistics. The export therefore writes a fingerprint alongside the engine, and the runtime refuses to load one that does not match the checkpoint it is paired with.

4.7.4 FP16 on trained weights

Converted whole to FP16, the trained RECAP checkpoint produces a mean absolute error of 0.83 against its eager reference, more than forty times the threshold, and the predicted actions retain only 17 % of the reference spread, so the action distribution has collapsed towards a constant rather than scattered around the reference. The same conversion at FP32 is exact to five decimal places and is accepted. The two conversions differ only in precision: identical graph, checkpoint, fifty samples and seeded noise.

The ACT policy converts to FP16 with a mean error of 6×10^{-4} and the full spread of its actions intact, so the export machinery and the engine builder are not implicated. Plain SmolVLA, without any of RECAP's additions, collapses exactly as RECAP does, at 0.55. And building the prefix engine at FP32 while leaving the action expert at FP16 restores the policy to 1.3×10^{-3}, inside the threshold by a factor of fifteen. The fault is therefore in the VLM prefix. ACT's own FP16 engine fails the gate under the rounded-dimension hold-out as well, at 1.7×10^{-2} against 1×10^{-2}, with the error concentrated in the last joint, the axis that rotates the switches, while no other dimension exceeds 5.6×10^{-3}. The measured values are collected in table 5.9.

Trained transformer weights develop outlier activation channels that half precision's smaller exponent range cannot hold [19]. It accounts for all three observations at once. ACT, the policy without a vision-language backbone is unaffected, the same architecture with random weights converts cleanly because such channels are a product of training, and the fault sits in the graph that carries those activations.

The prefix does not have to fall all the way back to FP32: TF32 keeps the range those channels need (table 3.1) and prevents the collapse seen with

FP16. The rover runs a TF32 prefix in front of an FP16 suffix for the ten-step sampler and in front of an FP32 suffix for the distilled one. Since the distilled policy only runs the suffix once, the difference in latency is much smaller, and an FP16 suffix loses more accuracy.

4.7.5 Building the export pipeline with an agent

The export pipeline above was produced by an AI coding agent, given a goal, a means of checking its own work, and the hardware to run on, and left to iterate until the check passed (section 2.4.2). The task has a good *oracle*: the exported engine must reproduce the original network's output, so correctness is a number, with the reference implementation already in PyTorch, the input fixed by seed, and the criterion a threshold on the reproduction error. The work is also dominated by a long tail of opaque, mechanical failures, since which operations the parser rejects, and which rewrite satisfies it, is discovered one error at a time and needs no insight once the pattern is understood. Both properties suit an agent. The loop ran once, on the Antigravity CLI with a Gemini model, from a brief kept in the repository: about 19 h over 2230 steps, unattended after the initial prompt, committing 1252 lines across seven files that are still what runs. The author defined the goal and the acceptance criterion, reviewed the code, and ran the benchmarks of section 5.3.

Two results passed a check while being wrong. The brief asked for a maximum absolute error below 1×10^{-3} in FP32; ACT came in at 2.5×10^{-3}, the threshold was widened to accommodate it, and for SmolVLA and RECAP the metric was changed from the maximum to the mean. The reasoning is recorded in the code and reasonable, but it is still an agent that edited its own acceptance criterion. The generated rebuild script tested for the legacy FP16 builder flag, printed a note when it was missing, and carried on; on TensorRT 11 that flag no longer exists, so a rebuild requested at half precision silently produced an FP32 engine, which passes the accuracy check more easily than a correct FP16 one. The oracle could not see that defect at all, and it surfaced only in the latency numbers.

The check given to the agent to verify its own work was flawed, since it only used random weights, which did not detect the issues mentioned in section 4.7.4. They were only caught when validating against a trained checkpoint and required an additional iteration of the agent to establish the real performance–accuracy trade-off. So only the right test can give the right result.

4.8 Depth Sensing

The manipulator gripper carries a ToF distance sensor that returns an 8×8 array of depth values, giving the policy the near-field measurement of section 3.9. It is mounted in the palm, between the jaws and facing along the tool axis (fig. 4.5), and it publishes its 64 per-cell distances on a ROS topic (section 2.3.3).

The sensor topic is declared as an `Int16MultiArray` with 64 named elements and the tag `observation` in the adapter configuration (section 2.9). Its converter produces a flat vector of 64 values, concatenated with the manipulator's joint state into `observation.state` (section 4.4), with the `exp_decay` closeness mapping applied inside the same conversion (section 4.4.1). Adding the sensor therefore needs just one configuration entry: no node was written for it and the policy is unchanged. The policy receives the images and the state vector separately, and whatever relation it uses between them is learned (section 3.5.1).

The grid occupies 64 elements of a state vector that is padded to 128 whether they are used or not (section 3.5.1). It arrives as a 5 Hz topic held between frames like any other observation, with no relevant impact on the control cycle of table 5.8: the conversion stage there is dominated by the JPEG decode of the camera topics, which 64 integers do not pass through. At contact range the gripper occludes the cameras, so the grid supplies the one measurement the rest of the sensors cannot.

Figure 4.6 takes one frame of that dataset apart. At this instant it reads 47 mm to 128 mm across the sixty-four zones, the gradient across the middle rows is the panel surface lying at an angle to the gripper, and the outer rows read past the jaws into the background.

Flattening the grid discards the arrangement of the cells, which is the main limitation of this route. Two zones that lie next to each other on the sensor end up at two arbitrary dimensions of the state vector, and their adjacency is not represented in the input at all (fig. 4.6c). The vector is projected by a single linear layer into one state token (section 3.5.1), and a fully connected layer does not recover that structure efficiently. The policy can still learn it, since the mapping from zone to dimension is fixed across the dataset, but it then spends capacity and training samples on something the sensor layout would already have. An encoder over the 8×8 layout would avoid this.

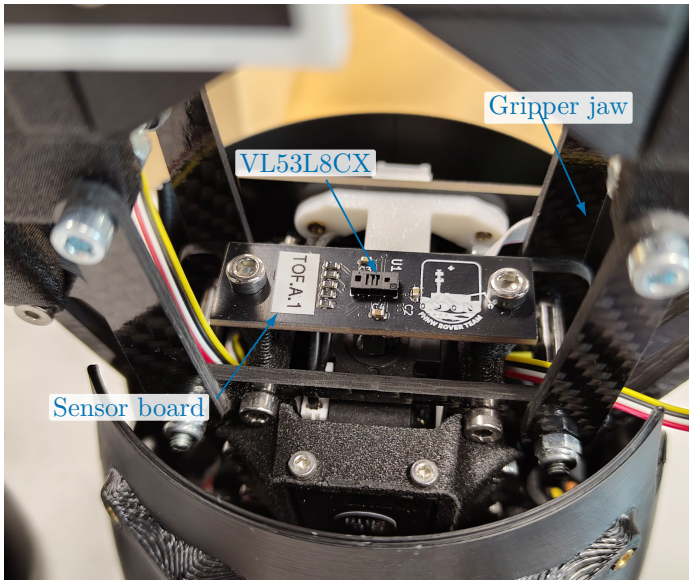


Figure 4.5: The time-of-flight sensor as mounted, looking into the gripper palm from in front of the jaws. The VL53L8CX sits at the centre of a carrier board bolted across the palm, so its 8×8 field of view looks out along the tool axis between the jaws and measures whatever the gripper is about to close on. This is the near field the cameras lose: at contact range the jaws occupy most of the gripper cameras' view of the target.

4.9 Workflow Automation and Agentic Control with n8n

The open-source workflow-automation tool n8n [1] is integrated with ROS2 to give operators a code-free way to compose and run task sequences. The integration is packaged as an n8n community node package (`@fhnw-rover/n8n-nodes-ros2`) that exposes ROS2 primitives as drag-and-drop nodes.[8] It talks to ROS2 over a **rosbridge** WebSocket endpoint using the `roslib` JavaScript library, so no `rclnodejs` or custom binary bridge is needed. As motivated in section 3.10, deterministic workflows drive the hardware while an LLM agent handles open-ended inspection; both are built from the same nodes. What was built on them, and what it does on the rover, is reported in section 5.5.

[8] Much of this package was written by an AI coding agent under the author's direction; see the Index of Assistive Technologies.

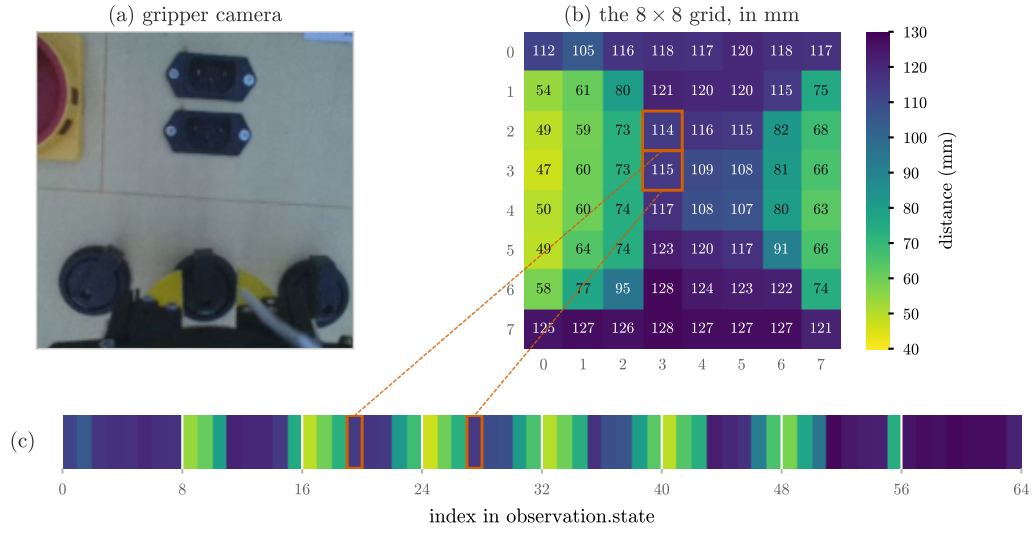


Figure 4.6: One frame of the panel-switch dataset, at the instant the gripper sits at the switch. (a) The left gripper camera. (b) The ToF grid at the same instant, converted back from the stored closeness values to millimetres. (c) The same sixty-four numbers as they reach the policy, one row of the grid after the other in `observation.state`, with the white lines marking the row boundaries. The two outlined cells lie one above the other on the sensor and eight apart in the vector; nothing in the input says so.

4.9.1 Node set

The package implements both directions of every ROS2 primitive, n8n as client and as server:

- **Topics:** *Topic Trigger* (subscribe, with a message filter), *Topic Next Message* (subscribe once), and *Topic Publish*.
- **Services:** *Service Call* (client) and *Service Trigger* (n8n advertises a service, so a workflow is invoked by a ROS2 service call).
- **Actions:** *Action Start/Status/Result/Feedback/Cancel* (client) and *Action Trigger*/*Action Send Feedback* (n8n as action server).
- **Introspection:** a *ROS2 API* node wrapping `rosapi` to discover the running system (listing topics, services, nodes, and actions, fetching their types and definitions, and getting or setting parameters), so a workflow or an agent can find what is available at runtime rather than being hard-coded.
- **Image capture:** *Topic Capture Image* subscribes to a `CompressedImage` and decodes its base64 payload into an n8n binary image, so camera frames can be displayed in the editor, handed to a vision step or stored as image to Asana.
- **Docker:** because the rover runs everything in containers, a *Docker Container* node lists, starts, stops, restarts, and reads logs from containers over the Docker socket, letting a workflow monitor and recover services on the rover.

All nodes are marked `usableAsTool` except the triggers, which n8n does not allow an agent to invoke. Figure 4.7 shows how the two directions reach the rover.

4.9.2 Bridging n8n and rosbridge

n8n's execution model is mainly built for REST APIs: a node runs, produces output, and finishes, with no long-lived state. rosbridge is the opposite: a persistent WebSocket carrying asynchronous topic traffic. This is bridged by holding connection state outside the node executions: every node resolves the same credential to an endpoint, and one pooled socket per endpoint is shared and reused rather than opened per execution. Concurrent connects are deduplicated, since parallel executions would otherwise each open a socket

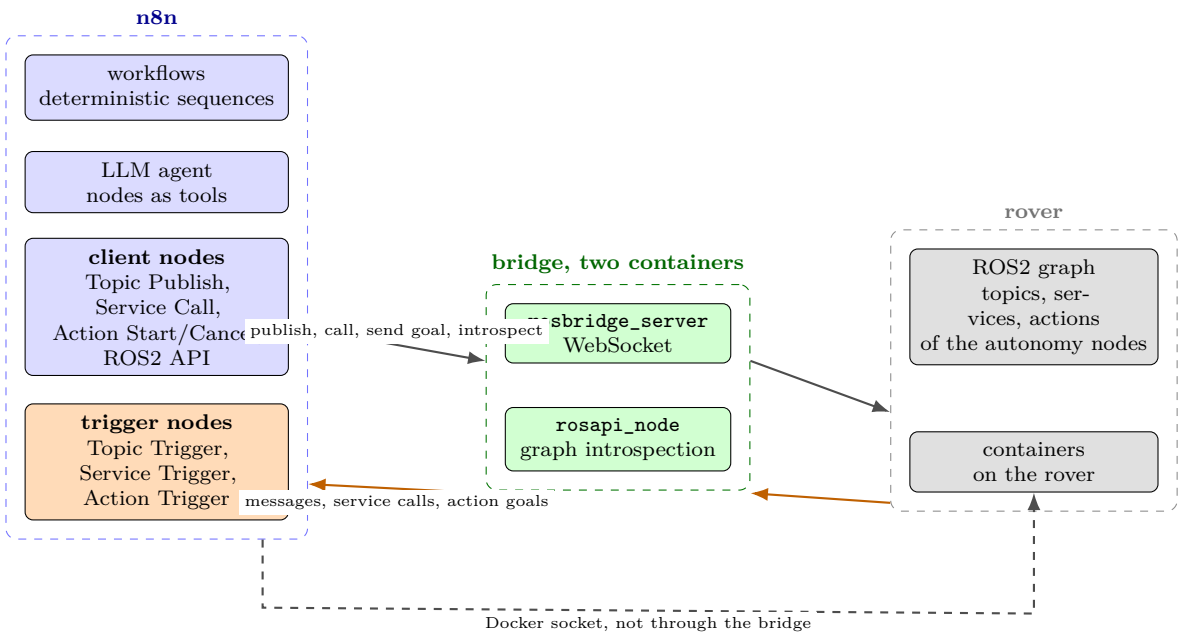


Figure 4.7: How a workflow reaches the rover. The upper lane is n8n as a client: a client node publishes, calls a service, sends a goal or asks `rosapi` what exists. The lower lane is n8n as a server: a trigger node hands the workflow a message, an incoming service call or an action goal, so ROS2 can start a workflow rather than only being driven by one. Both cross the same pooled WebSocket.

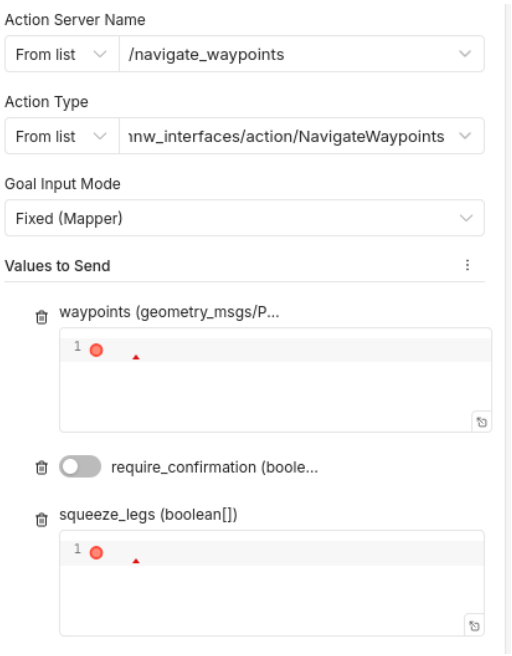


Figure 4.8: The n8n resource mappers makes it easier to input the required structure for calling actions, service or publishing messages on topics. The form structure is generated dynamically from the type information of the selected service, action or topic. Here the selected action is an action for autonomous navigation that receives a list of points to reach, on what points to wait for confirmation, and a flag indicating that the rover needs to squeeze (drive slower and with less safety margin).

and leak all but the last one, and a trigger node re-installs its subscriptions after a dropped socket instead of going silent.

Data Distribution Service (DDS) discovery latency is another problem: a freshly advertised publisher may not yet be known to subscribers, so the first messages are silently lost. A publisher therefore waits before it sends, and a message can be published as a short burst, so the loss of a first packet can be tolerated. Neither makes delivery certain. Where a command sets state rather than feeding a stream, the topic should be replaced by a service, whose request/response contract confirms receipt; the drill setpoints of the sampling workflow were ported from topics to services for exactly this reason (section 5.5.2).

4.9.3 Entering a message payload

A ROS2 payload is entered either as raw JSON or through n8n's resource mapper, which renders the message as a form with one input per field. The field list is generated from the live type definition, so the form always matches the message type currently selected, including nested and custom types, and an operator does not have to know the message layout by heart. An example of the mapper is shown in fig. 4.8. The mapper stores everything as strings, so the entered values are coerced back to their declared types. A payload that cannot be transformed is refused with an error before it is sent. Services and actions use the same mechanism for their request, response and goal payloads.

4.9.4 Error handling for agents

When the nodes are used as agent tools, a failure must not crash the workflow; the agent should instead see the error and react to it. Nodes running in an agent workflow soft-fail, and return a structured `{error: ...}` output, so the agent receives the failure as a tool result. Detecting that a node runs as an agent tool keys off the AI-tool output connection, because newer n8n executes tools through the regular workflow engine where the old execution flag no longer reports it.

4.9.5 Returning images from tool calls

Tool calling is a text protocol: the model emits a JSON call and the runtime returns a string. That is sufficient for reading a topic or invoking a service, but not for looking at a camera. In stock n8n the *Topic Capture Image* node produces an n8n binary attachment, and the agent framework discarded it

when packaging the tool result, so a vision-capable model was reduced to text: an image could only be attached once, at the start of a conversation, and never fetched on demand mid-task.

The fix was inside the core of n8n. The agent tool result path was extended so that a node's binary output is carried through to the model as a multimodal content block alongside the textual result. This makes "capture a frame and tell me what you see" a single tool call the model can issue at any point in its reasoning loop and allows the VLM to complete the vision tasks reported in section 5.5.7. The capture node also offers an optional downscale.

The change was submitted upstream as n8n pull request #34161,[9] "Support binary tool outputs and structured chat model tracing", which had passed triage and was still open on 2026-08-23. Until it is merged and released, the rover runs on a pinned fork.

4.9.6 Shared state: data tables and external stores

The agent's own memory is per-session and ends with the conversation (section 3.10). State that has to outlive a single execution is kept outside the workflow, in two stores.

n8n data tables hold the state the rover itself consumes. A data table is a persistent store that any workflow can read and write through dedicated nodes, and that the agent can be given as a tool. One case is joint-state configurations: a pose is stored as a row and replayed by name, so a position taught once is reused by every task that needs it, and a re-teach after a mechanical change is one row rather than a search through the whole canvas. Task state is kept the same way, which substeps are done and how the last attempt ended, so a workflow that is retried or handed between operator and agent reads that state instead of starting over. Data tables are visible in the n8n editor, so an operator can inspect and correct the shared state directly.

Replaying a stored position is also a workflow, *Position anfahren*, shown in fig. 4.9. It takes a position name, reads the matching row from the *Joint Positions* table, and calls `/move_to_joint_positions` with the joint values of that row. The service returns the estimated duration of the motion, and a wait node holds for that duration plus half a second, so the calling workflow continues only once the arm has arrived. Three triggers enter the same chain: a form for an operator, a webhook for an external caller, and the sub-workflow trigger, which the task workflows use, so moving the arm to a taught position only takes a single node. Teaching happens in another workflow: the arm is

[9] `https://github.com/n8n-io/n8n/pull/34161`

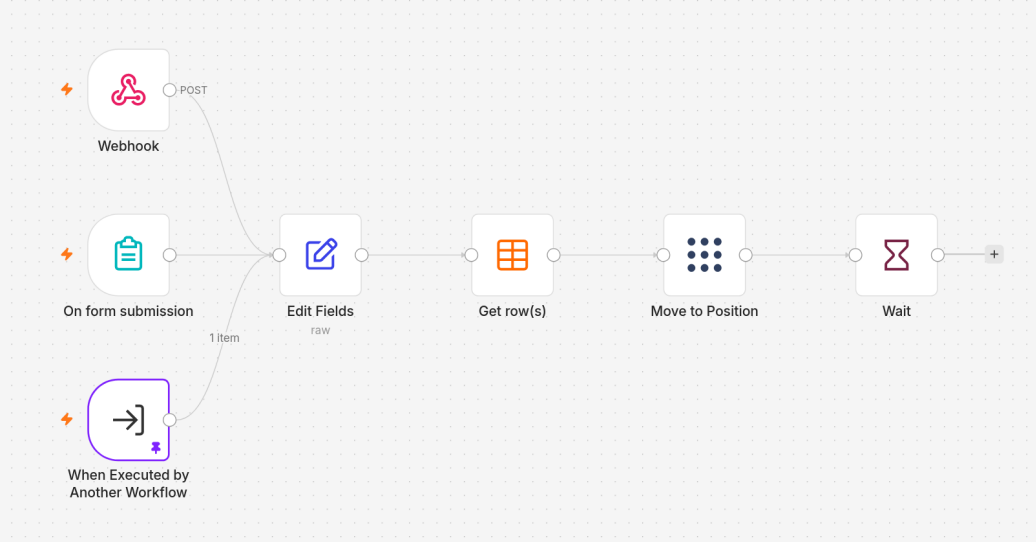


Figure 4.9: The *Position anfahren* sub-workflow, which replays one taught joint configuration. The three triggers on the left are a form, a webhook, and the call from another workflow. The row is read from the *Joint Positions* data table, the joint values go to `/move_to_joint_positions`, and the wait node holds for the duration the service reports.

driven to the pose by hand or by teleoperation, the workflow captures the joint state and writes it to the table with a name specified through a form.

Results that need to be worked on by team members are written to Asana. The workflows create tasks and can attach images, sensor measurements or system state to them. Team members can then act on these results in a tractable way. Asana was chosen because the team already tracks its work there, so an alert reaches an operator who is not watching the rover's battery live on the dashboard, and team members get access to all the required assets without a remote shell to the rover.

4.9.7 Agentic control

Agentic workflows combine n8n nodes with a LLM. The model reasons over a request and calls the ROS2 nodes as tools, reading topics, calling services, capturing an image, or discovering what is available through the *ROS2 API* node. In line with section 3.10, the agent is used mainly for inspection and diagnostics, and hardware actuation is left to deterministic workflows. The beacon is the exception only in appearance: setting an indicator light reports the rover's mode and moves nothing, so it stays inside the read-only intent even though it is a service call.

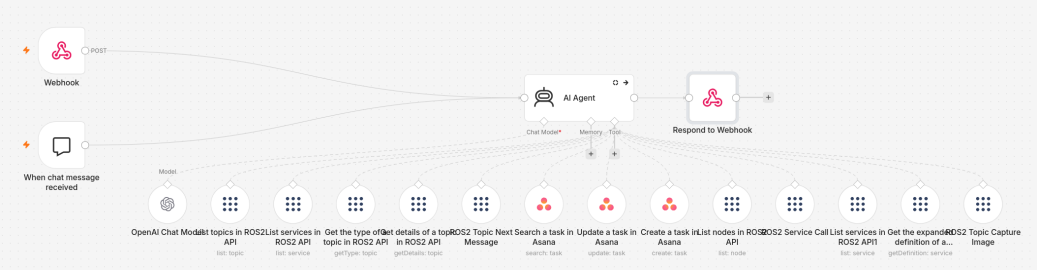


Figure 4.10: The deployed *ROS2 Agent* workflow on the n8n canvas. The two triggers on the left are the webhook and the chat window, the agent node in the middle carries the three ports that configure it, and an answer leaves through *Respond to Webhook* on the right. Everything strung along the bottom hangs off the tool port: the *ROS2 API* lookups, the topic, image-capture and service nodes, and the three Asana operations. The chat-model node sits at the left end of that row, and swapping the cloud model for the local endpoint of section 4.9.10 is that one connection moved. The memory port carries nothing, which is why a thread keeps its context only for the length of one conversation.

The deployed agent workflow is called *ROS2 Agent* and is shown in fig. 4.10. It is entered either from a chat window or from a webhook, and its reasoning loop is capped at 20 iterations. Its model is Gemini 3.5 Flash [25]; chat-model nodes for Claude Sonnet 4.6 and o3-mini sit unconnected on the same canvas and are swapped in by moving one connection, which is also how the local endpoint of section 4.9.10 replaces the cloud model. Fourteen tools are attached: seven *ROS2 API* lookups over the topic, service and node lists and the interface definitions, then *Topic Next Message*, *Topic Publish*, *Topic Capture Image* and *Service Call*, and three that create, update and search Asana tasks (section 4.9.6). The discovery tools are what let the agent work without hard-coded interface names: it lists what is running, fetches the definition of the interface it picked, and only then reads or calls it.

No action node and no *Docker Container* node is attached to this workflow, so the agent does not start long-running goals and does not restart containers. The workflow also carries no memory node, so an agent thread keeps its context only for the duration of one conversation.

What the agent can reach is decided mostly by which tools are attached to it. Below that, the *Topic Publish* node takes a list of allowed topic namespaces. Above it, a *read only* switch on the ROS2 and Docker credentials permits reading the graph, subscribing and reading container logs, while refusing every operation that changes state: publishing, advertising, service calls, action

goals and feedback, parameter writes, and container start or stop. Both checks run before the node opens its connection, so a refused operation never reaches the graph. The error names what was refused and why, which an agent reads back as the tool's observation. It sits on the credential rather than on the node, which is what makes it hold: a workflow author cannot lift it from inside the workflow, and neither can the agent, because an agent driving a node as a tool fills in parameters but never chooses the credential.

The main building block for workflows with human supervision is the n8n *Form* node, which inserts a human step between actions: the workflow pauses, shows the operator a form (for example a camera image and a choice), and issues the next service call or action only once the operator submits it. This is how human-in-the-loop confirmation is enforced before an unsafe operation, and the sampling workflow gates each of its five phases this way (section 5.5.2).

4.9.8 The autonomous task lifecycle

The maintenance task workflow consists mainly of one recurring pattern. A substep (like rotating a switch) runs as: select the target (the panel detector of section 4.2.3 returns candidate elements and one is chosen), approach it (the `align_to_target` action servos the gripper to the target pose, section 4.2.3), manipulate it (the `run_policy` action executes the learned policy until the progress tracker of section 4.5 reports completion), then verify and record the outcome to the shared state before advancing.

Every phase is a ROS2 action with a terminal result, so the workflow always learns how a phase ended (completed, timed out, cancelled, or aborted by the e-stop) and can branch on it rather than assuming success: a failed substep can be retried, and a repeatedly failing one escalated to an operator instead of continuing blindly. The sequencing lives in the workflow, so the order the ERC rules demand is enforced by the graph's topology. It is visible to anyone reading the canvas and the active node is visible during execution, so the flow can be viewed live.

4.9.8.1 The maintenance workflow

The ERC maintenance task is the largest instance of the pattern: 87 nodes covering the substeps of section 2.1.1, the main switch, the lever switches, the rotary switches, the plug, the rotary power switches, and the plate on the electromagnet. Each substep is the loop above wrapped in a repeat: the panel detector returns the candidate elements of one type, the operator picks one from a dropdown, the arm aligns and the policy runs, the operator confirms

the outcome, and the workflow returns to detection until that substep is reported done.

Before every substep the workflow offers a switch to manual operation. It puts the beacon into manual and takes the SpaceMouse node out of auto mode, which stops that node publishing the operator heartbeat and starts it publishing twist commands again. After the substep the autonomous mode is activated again: the beacon goes back to blue and the workflow waits the 5 s the ERC rules require before any autonomous motion is commanded (section 2.1.3). The delay is therefore enforced once per return to autonomy rather than once per run. The plug is the other: it requires a different tool, so that substep calls the tool-change workflow of section 4.9.9 twice, swapping the gripper before and after this substep.

Branch order is positional. Where a node has two outgoing connections, execution is depth first and the upper branch runs to completion first, so a branch's vertical position on the canvas is part of the program rather than cosmetic.[10] The maintenance workflow depends on this: the nodes that return the beacon and the SpaceMouse to autonomous mode are placed above the branch that starts the next substep, because below it the next substep would begin while the arm was still under manual control. Moving a node for tidiness can therefore change behaviour.

4.9.9 Tool changing

Thanks to the toolchanging mechanism the manipulator can take a tool from the rack bolted to the chassis (fig. 5.6) and park it again. The rack is on the rover, so the whole exchange is a chain of taught positions (section 3.10.4) and needs neither a camera nor inverse kinematics. It is a workflow of 25 nodes, entered either from a form, where the operator picks pickup or park and the slot, or from another workflow with those two inputs, which is how the maintenance workflow above calls it. Before anything moves, one message is read from `/joint_states` and kept, so the arm can be returned to where it started; a switch node then splits the two branches for parking or pickup.

Pickup approaches the rack, moves to the pickup position of the chosen slot, unlocks the coupling, presses into the slot along the coupling axis, and shakes the wrist for a second while the arm presses. The shaking is what makes a taught position sufficient: under a straight push the residual misalignment would jam the coupling, while oscillating the wrist lets the lead-in pull the two halves together, so the mechanism absorbs the error. The coupling is then

[10] `https://docs.n8n.io/build/flow-logic/understand-execution-order`

locked, a form has the operator confirm that the tool is held, the arm lifts it clear, and the tool is initialised. Park is the same in reverse and shorter, and both branches end by replaying the recorded joint state.

Every move in both branches is a call of the *Position anfahren* sub-workflow of fig. 4.9, so the canvas holds no joint value at all. The slot chosen in the form is substituted into the position name, so re-teaching a slot after a mechanical change is one row in the data table (section 4.9.6). The measured runs are reported in section 5.5.3.

4.9.10 A local agent model: harness and fine-tuning

The agent that runs on the rover should run *onboard*, meaning offline, at low latency and at no per-call cost, which means a small model (section 3.10.5). A small model must first be taught the rover's specific tools, and the training data comes from the agent's own runs: the same n8n agent, wired to the ROS2 nodes as tools, is exercised on operator-style prompts while a cloud model drives it, and every execution is captured to become a training example. Running the deployed agent thus doubles as data collection for its on-device replacement.

Collection. A collection script pulls completed executions from the n8n REST API and reconstructs each agent turn: the system and user messages, the exact tool schemas, and the tool calls with their results. Camera frames returned by the *Topic Capture Image* node are kept as multimodal tool results, so the vision path is trained and not only text tool use. The tool schemas are captured from the running system rather than written out by hand, which is what guarantees the model is trained against the interface it will be deployed on.

Quality control. Not every run is worth imitating. Each trajectory is graded pass/fail by hand, against per-prompt pass criteria and with the captured camera frames available to the grader, and only clean runs are kept; verbose final answers are compressed to one or two terse sentences and over-long tool observations are truncated, so the model learns concise, correct behaviour. Of 923 harvested trajectories, 796 passed grading. Those are then split at the prompt-*id* level, stratified by task category so that no prompt leaks across the split, 720 of them to training and the rest to validation, over eight task categories (power, resources, diagnostics, controllers, manipulator, beacon, imaging, and vision).

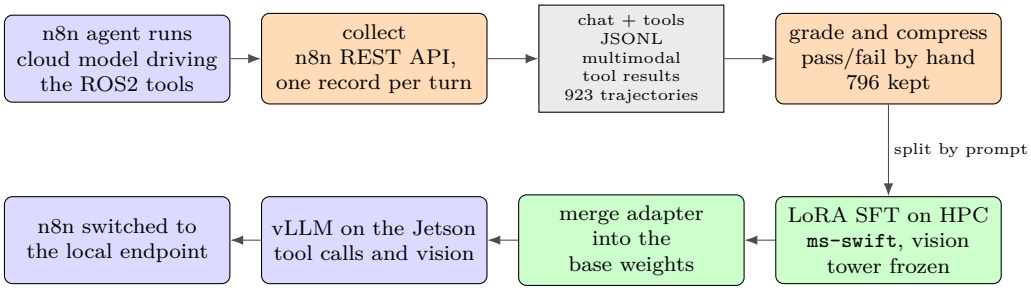


Figure 4.11: From live agent runs to a model on the rover. Nothing in the chain is synthetic: the training data is what the deployed agent did against the real ROS2 stack, driven by a cloud model, and the grading step is the only place a human intervenes.

Training and deployment. The base model, Qwen3.5-0.8B, is fine-tuned with LoRA supervised fine-tuning using `ms-swift` [62] on the FHNW SLURM cluster; the vision tower is frozen, which leaves 1.25 % of the parameters trainable (section A.3). The winning adapter is merged back into the base weights and served on the onboard Jetson through vLLM [34] with tool-calling and vision enabled. The n8n workflow is rewired to the local endpoint instead of the cloud. A system-prompt ablation (full prompt vs. a short anchor vs. none) showed no measurable difference, so the deployed model needs no elaborate prompt: the tool-use behaviour is in the weights. The full training configuration is given in section A.3. Figure 4.11 shows the pipeline end to end.

4.9.11 Deployment and system testing

The package is deployed on the Jetson via Docker Compose, alongside a separate container that executes Code nodes. The n8n container mounts the host's Docker socket, which is what the *Docker Container* node talks to. The image is built from the fork at the pinned commit that carries the change of section 4.9.5.

On the ROS2 side the bridge runs inside another container, the WebSocket endpoint all the nodes connect to and the graph-query service behind the *ROS2 API* node. Two bugs in `rosbridge_suite` crash the rosbridge server: its action-type lookup, which is how an action node resolves the type of the server it points at, broken since it was added in rosbridge 2.4.0, and its parameter cache, which throws once any parameter has been read. Both

fixes went upstream as pull requests[11],[12] and a third, for bounded string and sequence field types[13]. As of August 25, 2026 the first two are merged but not released and the third is open. To have a working setup, a fork with those fixes applied is used instead of the upstream version. The bridge uses an executor setting that keeps the WebSocket server from spinning a CPU core the Jetson needs for inference.

Resource leaks and lost messages under load, cannot be detected by mocked unit tests, so the package is validated by a system-test harness that runs the real nodes against a live `rosbridge` stack inside a container. It stress-tests the three cases the connection handling of section 4.9 is designed for. Repeated image capture confirms that decoded frames are released and the socket reused rather than reopened per execution. Publish bursts confirm messages survive the discovery delay. Subscribe/unsubscribe cycles confirm that trigger nodes clean up their subscriptions instead of accumulating stale ones over a long deployment. The results are reported in section 5.5.5.

4.10 Rover Manipulator Control and Inverse Kinematics

The 6-DoF manipulator on the rover "Barbara" (fig. 2.3) is driven through `ros2_control`. Inverse kinematics comes from the FZI `cartesian_controllers` package, whose default solver implements Forward Dynamics Compliance Control (FDCC): the arm is modeled as a virtual dynamic system and the target pose reached by integrating the joint accelerations $\ddot{q} = H^{-1}J^{\top}f$, with the joint-space inertia matrix H and the Jacobian J from the Orocos Kinematics and Dynamics Library (KDL) [52]. It runs at 120 Hz with 20 solver iterations per cycle and the Cartesian error scaled by six before it enters the virtual dynamics.

The solver was adapted by weighting the forearm roll. The fourth joint rotates the forearm about its own long axis and the sixth rolls the wrist about the tool axis, and the two axes become parallel whenever the wrist pitch between them is near zero, which covers most of the working range for a panel in front of the rover. The arm is then locally redundant and the solver distributes a commanded rotation arbitrarily between the two, which shows up as a large forearm swing for a small tool rotation. A weight on the forearm joint adds to its diagonal entry of H, so it accelerates less under the same

[11] `https://github.com/RobotWebTools/rosbridge_suite/pull/1307`
[12] `https://github.com/RobotWebTools/rosbridge_suite/pull/1308`
[13] `https://github.com/RobotWebTools/rosbridge_suite/pull/1312`

Cartesian force and the wrist takes up the rotation; a spring and damper on the same joint keep it from winding up over a long teleoperated session. All other joints are unweighted, and the values were tuned by hand on the real arm.

Three command paths reach the arm, and the system switches between. An operator's twist from a SpaceMouse or PS4 controller is integrated into a Cartesian target at 50 Hz, leashed to 8 cm and 1 rad from the measured pose so it cannot run away from a blocked arm that would then chase it at full speed once free. The alignment controller skips the integration and publishes a Cartesian pose directly. The policy controller writes joint positions to the position controller instead, because the policy predicts joint targets.

The gripper is commanded separately from the arm, and by jaw velocity rather than position: open, close, or stop. The command has to be repeated, because the hardware stops the jaws by itself after about a second with no message, so the teleoperation nodes republish it while a button is held. A grasp therefore has no target position to command. The autonomous operations close the jaws until the position feedback falls below a threshold or stops changing, which is how a jaw that has captured an object behaves. The policy drives the same interface, which is why its gripper action is clamped and rounded to $\{-1, 0, 1\}$ before it is published (section 4.4.1).

4.10.1 Hardware interface and homing

The joints are Dynamixel servos driven over two RS485 buses through a custom `ros2_control` hardware interface, each coupled through a simple transmission except the wrist, whose pitch and roll come from two motors through a differential one. Some joint limits depend on where the other joints are, so they are given in the Unified Robot Description Format (URDF) as expressions over the live joint positions and evaluated in the control loop: the shoulder range, for example, flips to the opposite side once the base has turned past 90°, which keeps the arm out of the rover body and still allows the full range of motion when the arm looks towards the rover.

4.10.2 Forward-kinematic accuracy

The kinematic model initially did not reach the accuracy the classical vision pipeline assumes, and its error varied depending on where the manipulator was operating at. Joint zeros, transmission ratios, link origins and joint-axis tilts were fitted from recordings of the arm moving in front of a fixed calibration board, on the principle of the gripper-marker survey of section 4.2.1: the board does not move, so any variation in its reconstructed position is model

error. That variation fell from 9.4 cm to about 2 cm, and two later recordings reproduced a median deviation below 1 cm through the deployed chain. The resulting forwards kinematic was also measured with a ruler to be accurate within 1 cm.

The remaining centimetre is consistent with belt backlash and structural flex, which are load- and direction-dependent and beyond a static model. No further improvement is needed: a centimetre sits well inside the 25 mm the approach phase has to reach, so the fitted chain can be trusted with the motion between sightings. The panel's position is expressed in a frame the kinematics knows nothing about and must still come from vision (section 4.2).

Chapter 5

Evaluation

The sections below are organised by contribution (section 1.4) and evaluate what each achieved and where it failed.

The policy results are measured in simulation, on the LIBERO benchmark. LIBERO is hardware-independent. Its robot, scenes, and tasks are fixed and public, an experiment can be repeated exactly, and the result can be compared against published work. The rover offers none of this. Its manipulator is a custom build whose state changes between sessions, and a task can fail for reasons that have nothing to do with the policy, such as backlash after a collision or a defective camera mount. Separating policy quality from hardware variance would need more repetition. The simulation therefore carries the quantitative claims about the policy, while evaluation on the rover shows that the whole system works on the actual hardware. The P8 baseline (the per-task ACT policies, see section 1.2) is carried through this chapter as a latency reference.

5.1 SmolVLA and Reinforcement Learning

Every configuration is evaluated on three LIBERO suites: SPATIAL and GOAL are short-horizon, and LONG (LIBERO-10) holds ten long-horizon tasks whose instructions need multi-step reasoning over object, spatial and goal knowledge; some of them chain two subgoals. A number is the success rate over 50 episodes per task, so 500 episodes per suite, taken at the configuration's best guidance weight. All advantage-conditioned policies are swept over the same guidance weights $w \in \{0, 0.5, 1, 1.5, 2\}$; the baseline policy is not conditioned on an advantage token, so guidance is meaningless, it is evaluated at $w = 1$ alone. The suite-level standard error at this sample size is about 2 points, which is the resolution every comparison below has to clear.

Table 5.1: Success rate in percent on the three LIBERO suites, 500 episodes per suite, each configuration at its best guidance weight. The average is the unweighted mean of the three suites. A + marks an addition to the pipeline of the row above it, with f^+ the positive fraction of the advantage labelling (section 5.1.1). The last column is the time to generate one 20-action chunk in bfloat16, measured for the deployed configurations only, on a Jetson Orin Nano 8 GB development unit rather than on the rover's Orin NX (section 5.3).

configuration	spatial	goal	long	avg	ms/chunk
SmolVLA, no knowledge insulation	65.0	70.6	30.2	55.3	–
SmolVLA with insulation (base)	71.2	75.8	41.6	62.9	–
+ rollout data ($f^+ = 0.8$)	77.8	80.0	35.8	64.5	–
+ co-training ($f^+ = 0.4$)	75.4	84.2	**43.8**	67.8	–
+ co-training ($f^+ = 0.8$)	**80.0**	**86.2**	38.2	**68.1**	922
+ SnapFlow distillation	74.2	84.2	40.0	66.1	**255**

The rollout data behind the post-training is 107 410 frames over 600 episodes on 30 tasks, collected by the base policy at a 64.3 % success rate. All of it was collected autonomously and labelled by the critic, the same procedure fig. 3.3 illustrates on one rollout episode. Table 5.1 reports the results.

These evaluations re-plan on every simulator step rather than replaying a chunk, which is what the SmolVLA paper does [51], so the numbers stay comparable to published ones. The rover cannot be run that way: re-planning every tick puts a full forward pass inside each 50 ms control cycle, which no configuration in table 5.7 comes close to meeting. The arm therefore replays a 50-action chunk and re-plans a few times per second (section 5.3).

The training recipe itself is the largest single effect: knowledge insulation gains between 5 and 11 points per suite over the same architecture trained the conventional way. Training on the policy's own critic-labelled rollouts then improves both short-horizon suites, by 6.6 points on SPATIAL and 4.2 on GOAL, but it degrades the long-horizon suite, from 41.6 % to 35.8 %. Mixing expert demonstrations back into the fine-tuning repairs most of that and lifts every suite above rollout-only training, giving the two best configurations at 68.1 % and 67.8 % average against 62.9 % for the base. The recipe therefore transfers to a compact VLA, with the qualification that rollout data alone is not sufficient for the long-horizon tasks, which section 5.1.2 traces to error compounding rather than to any post-training method.

No single configuration wins everywhere. The best average labels 80 % of the successful frames as positive and is the strongest on both short-horizon

Table 5.2: Composition of the rollout labels under each scheme, in percent of all 107 410 frames, failures included. "pos:fail" is the share of all frames that sit in a failed episode and are labelled positive, "neg:succ" the share that sit in a successful episode and are labelled negative.

scheme	positive	negative	pos:fail	neg:succ
$f^+ = 0.3$	15.0	85.0	1.7	30.9
$f^+ = 0.4$	20.7	79.3	3.0	26.5
$f^+ = 0.8$	51.5	48.5	16.2	8.9

suites, but it leaves LONG below the base it started from. The stricter labelling at $f^+ = 0.4$ is the only configuration that clears that regression, at 43.8 %, and pays for it with four points on SPATIAL. Both differences are on the order of the 2-point standard error, so the conclusion that holds is that co-training removes the long-horizon damage and that the threshold decides which suite gets the remainder.

5.1.1 The labelling threshold and what guidance adds

The threshold is the one free parameter of the labelling, and table 5.2 shows what it does to the data before any policy is trained on it. At $f^+ = 0.3$ and 0.4, between 26 % and 31 % of all frames are frames inside *successful* episodes that still receive the negative label. Those are largely the approach and transport motions the task requires. Labelling them negative trains the policy to treat necessary actions as behaviour to avoid.

Figure 5.1a shows the consequence on SPATIAL. At $f^+ = 0.3$ the whole sweep sits below the base. At $f^+ = 0.4$ the unguided policy still edges past it, but guidance erodes the result steadily, down to 65.4 % at $w = 2$. That is exactly what the guided velocity field predicts for a mislabelled dataset: if the necessary motions were labelled negative, the positive conditional no longer contains them, the difference $v_{\text{pos}} - v_{\text{unc}}$ of section 3.5.5 points away from them, and a larger weight pushes the policy further along that direction. At $f^+ = 0.8$ the negative label is reserved for genuinely low-advantage behaviour, which mostly comes from failed episodes, and both the unguided policy and moderate guidance improve on the base. This is the setting used throughout.

Even at the best threshold, guidance itself contributes little as long as the fine-tuning sees rollout data only. Figure 5.1b puts that in one picture: the rollout-only policy gains 1.0 point from steering, while nearly all of its lead over the base is already present at $w = 0$. The gain comes from fine-tuning on labelled rollouts, not from steering at inference. Co-training changes this.

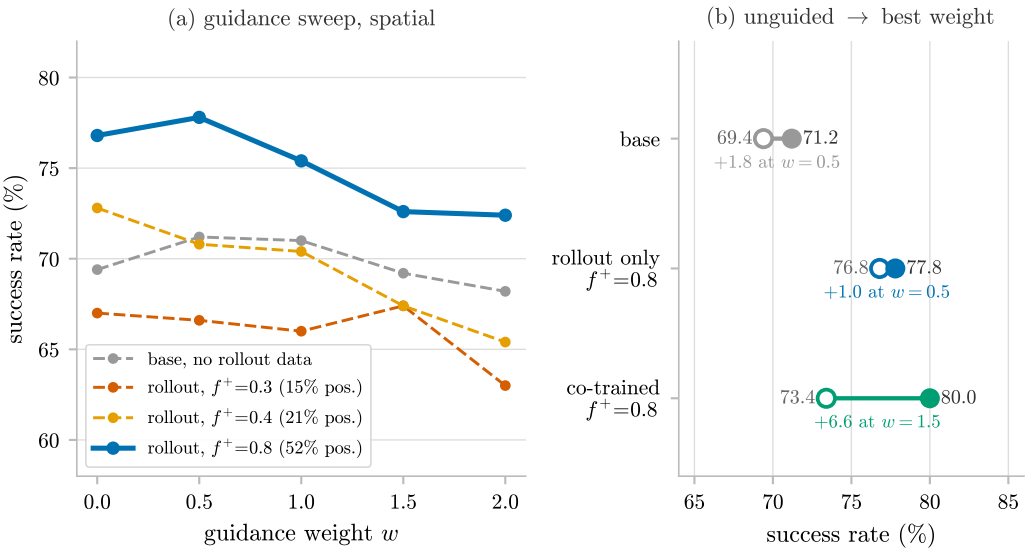


Figure 5.1: Effect of the guidance weight on the SPATIAL suite, 500 episodes per point. (a) The sweep for the base policy and for three rollout-only fine-tunes that differ only in the labelling threshold. Aggressive labelling ($f^+ = 0.3$, 0.4) makes guidance actively harmful, since the motions it steers away from are ones the task needs. The deployed threshold ($f^+ = 0.8$, solid) is the only one that improves on the base. (b) What guidance is worth, as the step from the unguided policy to the best weight. It buys almost nothing until expert demonstrations are co-trained back in.

The co-trained policy at $f^+ = 0.8$ starts lower unguided, at 73.4 %, and gains 6.6 points from guidance on SPATIAL and 2.4 on GOAL, ending as the best configuration in table 5.1.

Forcing the *negative* token at inference points to the same conclusion. For the $f^+ = 0.3$ policy the probe finds no usable separation at all: the negative token scores 66.4 % against 66.0 % for the positive one on SPATIAL, and on GOAL it scores higher, 76.4 % against 73.6 %. With seven of every ten successful frames labelled negative, the token never carried a good-versus-bad distinction in the first place. On LONG the separation exists but stays small, at 1.0 point for $f^+ = 0.8$ and 3.8 for $f^+ = 0.4$. A conditioning token whose two settings produce nearly the same behaviour leaves guidance nothing to amplify, which is the mechanism behind fig. 5.1b.

(a) One subgoal: “pick up the book and place it in the back compartment of the caddy”. The policy completes it.

(b) Two subgoals: “put both the alphabet soup and the cream cheese box in the basket”. The first object is placed, the second is never picked up, and the episode runs to timeout.

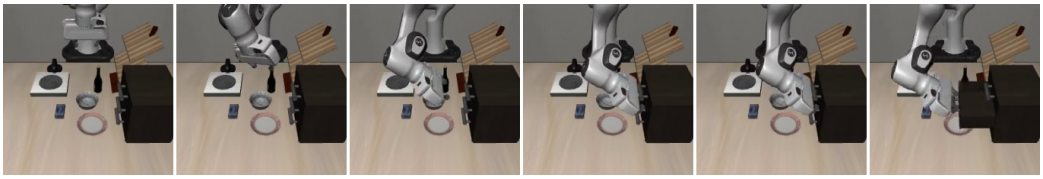
Figure 5.2: Six evenly spaced frames of two rollouts of the co-trained policy on the LONG suite. The suite average is low because instructions that chain two subgoals compound the error of each; single-subgoal tasks in the same suite reach 70 % to 80 %.

5.1.2 Where the policies still fail

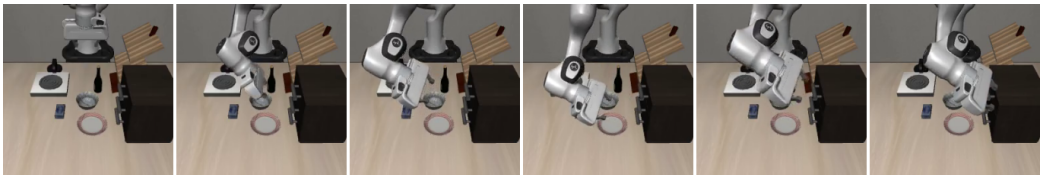
Per-task rates at 50 trials carry a binomial confidence interval of roughly ±14 %, so only patterns that repeat across configurations are read here.

The dominant remaining failure is the long-horizon tasks that chain two pick-and-place subgoals in one instruction. Every configuration stays below about 40 % on that subset of LONG, and below 16 % on the hardest of them, “put both the alphabet soup and the cream cheese box in the basket”. The single-subgoal tasks in the same suite reach 70 % to 80 %, which is in line with the short-horizon suites. The low LONG averages are therefore error compounding across subgoals rather than a property of any post-training method. Figure 5.2 shows both cases from the same suite and the same policy. In the failing rollout the arm places the first object and then spends the remaining time near the basket without ever picking up the second.

The second pattern is what co-training is for. If the base policy never succeeds at a skill, its rollouts contain no positive example of that skill, and rollout post-training has nothing to reinforce. Opening a drawer is the plainest case: the base policy’s 600 rollout episodes contain no successful drawer opening, and only the co-trained policy acquires it (fig. 5.3). Rollout data sharpens behaviour the policy already shows; behaviour it lacks has to



(a) Co-trained policy: the gripper reaches the handle and pulls the drawer open.



(b) Base policy: the gripper hovers in front of the cabinet, never engages the handle, and the episode times out.

Figure 5.3: Six evenly spaced frames of two rollouts of the GOAL task "open the middle drawer of the cabinet". The skill is present in the expert demonstrations and absent from the base policy's own rollouts, so only the configuration that mixes demonstrations back into fine-tuning learns it.

come from the demonstrations. This is the role human corrections play in the original recipe, filled here by data that was already on disk.

5.1.3 One-step distillation

SnapFlow costs 2 points of average success at its best guidance weight, and 3.6 as deployed, against a factor of 3.6 in latency on the eager path measured in table 5.1. On the deployed TensorRT engine the gain is smaller, because the sampler is no longer the dominant cost there (section 5.3). The student matches its teacher on GOAL, beats it on LONG, and loses on SPATIAL, which is the suite where guidance helps the teacher most and where a single integration step cannot reproduce the guided field.

Baking the guidance into the student was tried as well, so that a single unguided pass would reproduce the behaviour the guided teacher shows, and table 5.3 reports what came of it. With the training model supplying its own distillation targets it collapses, to 57.3 %: the flow-matching term is anchored on the training data and pulls the student back towards the unguided field, while the blend's unconditional branch is trained by no loss term at all and drifts. A frozen copy of the teacher supplying both terms fixes the mechanism where guidance helps the teacher most, reaching 76.8 % on SPATIAL in a single pass, within 3.2 points of the guided teacher, but it overshoots badly on LONG. Guidance baking is therefore worth it only on the

Table 5.3: Success rate in percent of the distilled students against their teacher, the co-trained $f^+ = 0.8$ policy. All students are trained for 15k steps on the teacher's own fine-tuning mixture. Guidance baking distills the blended field at $w = 1.5$, so that one unguided pass reproduces guided behaviour.

policy	spatial	goal	long	avg
distilled student	74.2	**84.2**	**40.0**	**66.1**
+ guidance baked, self-distilled	67.0	72.4	32.6	57.3
+ guidance baked, frozen teacher	**76.8**	75.2	25.2	59.1
teacher, ten-step sampler	80.0	86.2	38.2	68.1

suites where the teacher itself profits from $w > 1$. The plain student is the deployed one, at 64.5 % average at its single-pass operating point; any other weight runs the vision-language prefix a second time as well as the action expert (section 4.7), and since the prefix is where nearly all of the chunk is spent, that gives back more latency than the distillation bought (section 5.3).

5.1.4 The object suite and image augmentation

LIBERO has a fourth suite, `libero_object`, and it is missing from every table above, because every policy trained here scores 0 % on the OBJECT suite, base and fine-tunes alike, running each episode to timeout without completing the task. A suite on which nothing scores cannot separate methods, so it was excluded rather than reported as a row of zeros.

Re-running the base pre-training unchanged except for photometric image augmentation lifts `libero_object` from 0 % to 53.6 %, while the three reported suites stay within evaluation noise (63.5 average against 62.9). The suite was never broken and the demonstrations were never corrupted. The policy had overfitted to the exact appearance of the training images and did not survive the change in object appearance the suite exists to test, and colour jitter in training closes most of that gap. The same sensitivity, and the same partial remedy, turn up on the rover, where the light in the laboratory changes between sessions (section 5.7). Because all configurations compared in table 5.1 are trained without augmentation, they are affected identically and the exclusion does not favour any of them.

5.1.5 What these numbers do not show

First, every model here is trained on the LIBERO demonstrations alone, without the large community corpus the released SmolVLA is pre-trained on, so the absolute numbers are not comparable to published ones. The released LIBERO fine-tune scores close to the no-insulation baseline under this protocol, which makes that baseline a fair stand-in for the original recipe. Second, each configuration was trained with a single seed, and reporting the maximum of a five-point guidance sweep is worth an optimistic point or two on top. Every advantage-conditioned configuration receives the identical sweep, so the comparisons among them hold; the absolute values are flattered, and the no-insulation baseline, which has one operating point, does not get that benefit. Third, only three positive fractions were tested, and the differences between them on LONG sit at the edge of what 500 episodes can resolve.

The largest limitation is the platform: these numbers are from the LIBERO robot and not the rover arm. A policy for the manipulator was trained with the same recipe, minus the rollout stage (section A.2), so the method is not confined to the benchmark; what does not transfer is the measurement. This section says nothing about how the rover policy performs on the hardware of section 5.7, and the gain reported here was obtained on a different robot, a different dataset and a different observation layout. A comparison against the per-task ACT policies of P8 is not possible either, and for the same reason the rover cannot be compared with LIBERO: those policies were trained and evaluated on the rover, in a setting these numbers do not share.

5.2 Visual Servoing

The approach phase has one requirement to meet: place the gripper close enough to a panel element that the policy can take over. It is evaluated here three ways. First as a closed loop under its own control, where what is scored is the residual the gripper physically ends up with. Then as the tracking chain that reports where the tool is (section 5.2.1), and finally as the forward kinematics that chain rests on (section 5.2.2). The three measure different quantities and the distinction matters, because only the first involves the controller at all.

With all three panel markers visible the board pose is recovered correctly, and the `align_to_target` action drives the gripper to any target position on the board. The residual at the target, held against a ruler, is 1 cm to 2 cm, which is inside the 25 mm the substep needs and close enough for the policy to take over at the correct switch.

A base camera watching a moving arm loses sight of the gripper's markers at exactly the poses where precision matters. The deployed architecture does not depend on the gripper being visible at all. Only the board is tracked by vision, and the board is static, so one full view of its markers fixes the panel pose for the remainder of the run. The gripper is carried by forward kinematics through the registered camera-to-base transform (section 4.2.2), which is why the measurement of section 5.2.2 matters as much as it does: the chain that carries the tool is the accurate one. This holds as long as the board is seen in full before manipulation begins.

If the board or the rover shifts mid-task while the board's markers are not fully visible, the stored panel pose describes a panel that is no longer there. Nothing in the loop notices, because the gripper estimate it is being compared against is still perfectly self-consistent. The recovery is mechanical rather than algorithmic: the gripper is moved clear of the camera's field of view so the markers are unobstructed, and a fresh fix is taken on the board. The maintenance workflow therefore re-runs the object detection on the board on every pass instead of once per run (section 4.9), so the switch positions are re-read rather than carried through the whole sequence. That does not clear a stale *panel* pose: the board filter's innovation gate rejects a measurement far from its state, so a panel that has moved has to be cleared by restarting the filter (section 4.2.2). An alignment takes a few seconds on the rover, and at most a few tens of seconds from a distant starting pose. The time is set by the distance travelled and the velocity clamp of section 4.2.3 rather than by the controller, so it is not a quantity this work optimises or reports against.

The alignment controller was reimplemented in C++ along with the rest of the approach phase (section 4.2). Table 5.4 times both versions of that node stage by stage, over the same recorded input. The regular build runs a cycle in 58.8 µs against 964.2 µs, a factor of 16.4, and a release build reaches 17.6 µs, a factor of 54.9. For a loop with a deadline the tail is important: the Python node's 99th percentile is 1.3 ms and its worst cycle 3.7 ms, against 99.3 µs and 158.5 µs for the deployed C++ one.

The measurement comes from one node in isolation, measured on a development workstation rather than on the rover's Jetson. The absolute figures are not the ones the rover runs at. The rover's Jetsons are both already running at an elevated load, because they are responsible for sensor readings, camera streaming, communication between nodes and son on, so efficient implementations are important, even if looked at alone the Python version would be fast enough.

Table 5.4: Per-cycle cost of the alignment controller, stage by stage, for the Python node and the deployed C++ one. Means over the same recorded input, on a development workstation and not on the rover. A release build of the same C++ node reaches 17.6 µs over the whole cycle.

Stage	Python (µs)	C++ (µs)	Factor
guards	5.3	0.2	32.1
transform lookup	35.7	1.2	30.9
pose error	132.5	6.3	21.2
debug publish	223.6	19.5	11.5
feedback publish	16.9	3.5	4.8
velocity	100.2	11.9	8.4
target publish	449.8	16.3	27.6
whole cycle	964.2	58.8	16.4

5.2.1 Board targeting with the base camera

The base-camera mode is measured end to end, against the requirement it exists to meet: place the gripper within 25 mm of a switch in the panel plane. Depth enters only in that the tool must not reach the surface. The architecture under test is the one of section 4.2.2, in which a wiggle pins the camera-to-base registration once and every move afterwards is forward kinematics, with no marker visible at the working pose.

Each recording follows the protocol in order. First the panel alone, with the gripper out of frame, which is where the object detector reads the switch positions and where the board pose used for the whole recording is measured; the rover must not move afterwards. Then about a minute of wiggle with the gripper closed, holding roughly 2 s per pose and tilting the wrist between poses. Then each target in turn, pausing still and closed on arrival and opening and closing the gripper there and nowhere else, so the opening events index the targets automatically. The arm is driven to the targets by hand. What is scored is the position the tracking chain reports at each target against the position the object detector gives for the switch that was touched.

The instant that is scored is the last still frame before the gripper opens, since the marker offsets hold only with the jaws shut (section 4.2.1).

Table 5.5 reports three recordings of black switches, with no fitted parameters: the tip offset is the physical 126.9 mm of section 4.2.1 and the registration is the stored constant. Driving on a recording's own registration puts every one of the twelve targets within tolerance, at a median of 14.9 mm. Using a registration solved from a different recording costs about 3 mm of

Table 5.5: Position error in the panel plane at each visited switch, over three recordings and twelve targets, with no fitted parameters. "Own registration" uses the camera-to-base transform solved from the same recording, "transferred" one solved from a different one, and the last row pools all nine combinations of the three registrations with the three recordings. Red switches are excluded; see the text.

registration	median (mm)	p90 (mm)	within ±25 mm	
			%	targets
own recording	14.9	–	100	(12/12)
transferred	18.1	–	83	
all nine pooled	17.0	24.9	89	(32/36)

median error and drops four of thirty-six targets outside the tolerance. Pooling all nine combinations gives 17.0 mm at the median, 24.9 mm at the 90th percentile and 38.2 mm at the worst. Height is not closed by vision in this architecture and inherits the registration alone, which is worth about 22 mm.

The red switches are left out of the table because they are larger targets and the operator approached them higher and less carefully; including them raises the pooled median to about 24 mm, which still meets the requirement but measures the operator as much as the system. The per-target error is scored with a leave-one-out fit, so each target is predicted from the others. The reference is the weaker point: the errors are measured against the object detector's switch positions, and three of those positions move by 5.0 mm to 8.5 mm between recordings (section 4.2.3). Several millimetres of the budget above are therefore the reference and not the gripper chain, and settling that needs an externally surveyed panel.

Underneath the end-to-end figure the error separates into parts that behave differently, measured over 1411 still frames at thirty holds across eleven stations spanning 605 mm. Frame-to-frame scatter with the arm stationary is 1.49 mm at the median and 3.52 mm at the 90th percentile, and averaging twenty frames takes it to about 0.33 mm. What remains after one rigid camera-to-base transform and one rigid tool offset are fitted to every hold is 10.87 mm at the median and 17.19 mm at the 90th percentile. The systematic part is therefore roughly seven times the random part. This is the number that decides where further effort is worth spending: temporal fusion, a longer filter or a pose graph all attack the random part and would change the result very little.

That systematic term is characterised but not explained. It grows with how far the arm reaches, and vision reads roughly 15 % to 20 % less reach displacement than forward kinematics. The trend survives spatial hold-out, beating the null model in eight of eight disjoint workspace regions, and in the one genuine extrapolation available it predicted a station at 782 mm to 793 mm of reach to 9.7 mm against a 51.4 mm null. Lens distortion was ruled out, since zeroing it entirely moves the pose by 1.05 mm. So were principal-point error, image position, wrist attitude, joint-zero errors, and a monocular scale error, whose required radial shape fails the same hold-out. Camera-side and arm-side causes cannot be separated in the available recording: reach and camera range correlate at 0.97 there, and the two candidate directions are 7.9° apart. The decisive experiment is a second rover-fixed camera seeing the same markers, which the rover already carries, though its view of the markers has not been checked.

5.2.2 Forward-kinematic accuracy against a ruler

The symptom that started the investigation above was a commanded height wrong by tens of centimetres, varying with where on the panel the arm was working. Either source of the composed chain could produce it: the arm is belt-driven and compliant, and monocular PnP on a 25 mm marker at 400 mm range is not obviously better.

Forward kinematics was measured on its own: the arm was driven to touch the three panel markers in turn, and the distances between the three reported positions were compared against a ruler. It reproduced the triangle as 259.0 mm, 384.6 mm and 462.0 mm against a measured 265 mm, 380 mm and 463.3 mm. The largest disagreement is 6 mm, but that is finer than the check itself resolves: a ruler read against a gripper that wobbles under its own compliance is good to about a centimetre, not to the millimetre. What the test establishes is therefore that the kinematic chain is true at the centimetre level, which the argument below needs, and not a millimetre figure for it.

The error was therefore on the vision side, and it had two causes: the two-fold PnP ambiguity of section 4.2.1 and the tool-axis offset of section 4.2.1 that had been left at zero. Both are pose-dependent, which explains why the error varied with position.

The ruler check measures relative displacement within one region of the workspace, which the panel task needs. It says nothing about the absolute tool pose across the whole reachable range.

5.3 Real-Time Performance on the Jetson Orin

5.3.1 Benchmark method

The measurements come from a standalone, ROS-free benchmark that runs on the Jetson Orin (SM_87) and imports the same TensorRT wrapper that ships in the policy controller, so the numbers cannot drift from the deployed code. The inputs match the rover configuration: three cameras at 360×640, an 83-dimensional observation state, a seven-dimensional action, and a chunk of 50 actions, with advantage conditioning on and a guidance weight of 1. That holds for every figure on the deployed policy. The plain-SmolVLA figures of table 5.6 and fig. 5.4 are the LIBERO checkpoint instead, on its own two-camera configuration, because no plain SmolVLA runs on the rover: the deployed pair is ACT and SmolVLA+RECAP. They are reported to place the two architectures against each other, not to be read against the deployed rows of table 5.7. Each figure is the average over 20 runs of a full `predict_action_chunk`, which covers the vision-language prefix and the denoising loop. The benchmarks run on the trained checkpoints, on one toolchain throughout: torch 2.11 with TensorRT 11.2 and CUDA 13.

Each run also records the median behind that average and the 95th and 99th percentiles. On the TensorRT engines the 95th percentile lies between 0.1 % and 1.6 % above the median in every run reported here; the eager and `torch.compile` paths are the noisier of the two, at between 0.3 % and 4.5 %. That is small against the differences this section argues from. The two SnapFlow suffixes of table 5.7 lie 6 ms apart, at medians of 324.4 ms and 330.4 ms with 95th percentiles of 325.2 ms and 331.5 ms. The faster engine's 95th percentile still sits below the slower one's median and the 6 ms is a difference rather than noise.

5.3.2 Latency across backends

Table 5.6, plotted in fig. 5.4, puts the two policy families on the same hardware. ACT is the P8 baseline of section 1.2 and the much smaller model. On a FP16 engine its chunk of 100 actions costs 15.9 ms, against 196 ms for a 50-action SmolVLA chunk. Per action *consumed* — the controller takes 50 from either policy — that is 0.32 ms against 3.9 ms. Both fit the open-loop budget of chapter 2. What the single large policy costs is therefore the re-planning rate rather than the command rate, and what it buys is one model for all tasks instead of one per task.

Table 5.6: Latency of one action chunk on the Jetson Orin, averaged over 20 runs. ACT predicts 100 actions per chunk, SmolVLA 50 with the ten-step sampler. The SmolVLA column is the plain LIBERO checkpoint, since no plain SmolVLA runs on the rover; the deployed policy's own latencies are in table 5.7. Bold marks the fastest backend measured, which for both policies is the whole-graph FP16 engine; * marks that the acceptance check on trained weights rejects that engine in both columns, for different reasons: SmolVLA's actions collapse, while ACT's are intact on average and fail on one joint alone, the axis that travels furthest, which its deployed TF32 engine resolves at half the cost of FP32 (section 4.7.4).

backend	ACT (ms)	SmolVLA (ms)
eager FP32	84.3	1393
eager FP16 (autocast)	61.9	968
`torch.compile` (cudagraphs)	85.0	1396
TensorRT FP32	77.5	856
TensorRT TF32	38.4	–
TensorRT FP16*	**15.9**	**196**

Compilation with `torch.compile` in the cudagraphs mode gives nothing on either policy, so it is not used. The export itself buys little at strict FP32: a factor of 1.09 on ACT and 1.6 on SmolVLA. The deployed policy says the same on the configuration it actually runs: with the prefix held at strict FP32 the export is worth a factor of 1.74 on the ten-step chunk and 1.23 on the SnapFlow one, measured against the fastest FP32-prefix engine the acceptance check admits in each case (table 5.7). For ACT an FP32 engine is barely worth building; on the deployed policy the same export is worth the 1.74 and 1.23 above, before any precision is given up. Dropping ACT to TF32 doubles its factor, to 2.2 over eager, and a whole-graph FP16 engine reaches 5.3; SmolVLA reaches 7.1 the same way. Neither FP16 engine is accepted on trained weights, and for different reasons (section 4.7.4), so those last two figures are what half precision would have bought rather than what is deployed. TF32 is accepted on both policies, and it is what delivers the usable speed-up.

5.3.3 The deployed policy and what each precision costs

The deployed policy is SmolVLA with RECAP post-training, exported as a prefix and a suffix graph (section 4.7). Table 5.7 reports it with both samplers and at every prefix precision the check was run against, since which of them

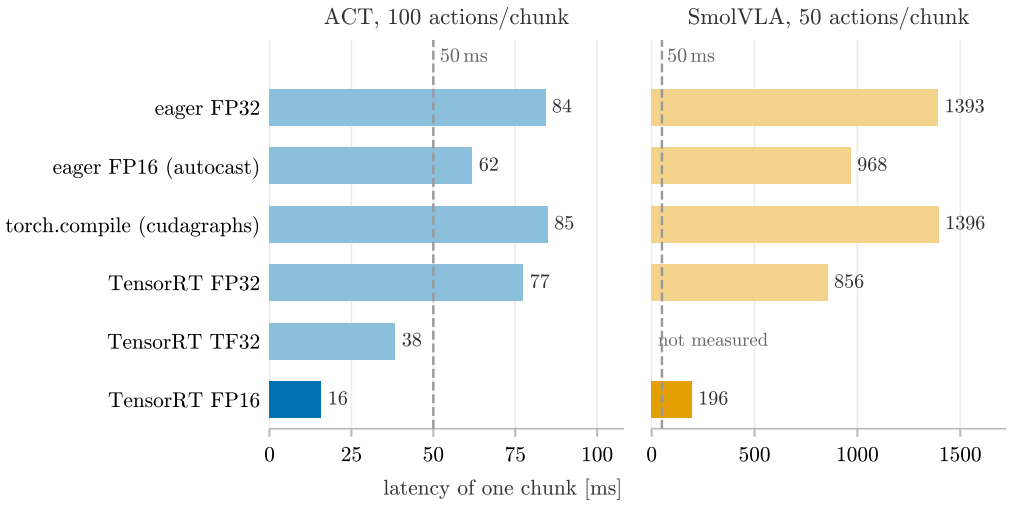


Figure 5.4: Latency of one action chunk on the Jetson Orin, averaged over 20 runs, for the P8 ACT baseline (left) and SmolVLA (right). The solid bar is the fastest backend measured. In both panels that is a whole-graph FP16 engine, and in both it is an engine the acceptance check on trained weights rejects (section 4.7.4), for different reasons: SmolVLA's actions collapse, while ACT's are intact on average and fail on one joint alone, which its deployed TF32 engine resolves at half the cost of FP32. The two panels carry separate scales, since the policies differ by more than a factor of ten and a shared axis would flatten ACT into the baseline. The dashed line marks the 50 ms of one control cycle at 20 Hz: a chunk is replayed over the 50 of those cycles the controller consumes it in rather than in one, so the line is the re-planning reference of table 5.8 and not a bound the chunk has to fit under.

it admits decides the deployed configuration. On a strict FP32 prefix the ten-step chunk falls to 778 ms, and SnapFlow removes the nine remaining denoising steps to reach 674 ms. That is not the deployed configuration, because the prefix does not in fact need full FP32. A TF32 prefix (table 3.1) passes the acceptance check on trained weights with the full action spread intact (table 5.9), and takes the deployed SnapFlow chunk to 330 ms and the ten-step chunk to 434 ms, roughly halving the FP32-prefix figure.

Whole-graph FP16 would have reached 128 ms, and the distance between that and any accepted engine is almost entirely the prefix. Reading the engine rows against each other separates the two graphs: one denoising step costs about 7 ms in an FP16 suffix fed by an FP16 prefix and about 12 ms otherwise, which puts the shared vision-language prefix at roughly 662 ms at

Table 5.7: Latency of one 50-action chunk of the deployed SmolVLA+RECAP policy on the Jetson Orin, averaged over 20 runs. The left column is the standard ten-step flow-matching sampler, the right one the single-step SnapFlow variant (section 4.6.3). The whole-graph FP16 row does not survive the acceptance check on trained weights (section 4.7.4); it is kept because it shows what the optimisation would have been worth had it been usable. The deployed configuration is the fastest one that clears the acceptance check of table 5.9: a TF32 prefix with an FP16 suffix for the ten-step sampler, and a TF32 prefix with an FP32 suffix for SnapFlow. Both are set in bold. The ten-step sampler was not measured with an FP32 suffix, since it is not the deployed sampler. All rows are measured on the rover policy, on the panel-switch dataset of fig. 2.6, so they carry its three camera streams and its 50-action chunk.

backend	ten-step (ms)	SnapFlow (ms)
eager FP32	1357	837
eager FP16 (autocast)	976	324
TensorRT FP32	858	683
TensorRT FP32 prefix, FP16 suffix	778	674
TensorRT TF32 prefix, FP16 suffix	**434**	324
TensorRT TF32 prefix, FP32 suffix	–	**330**
rejected by the acceptance check, listed for comparison		
TensorRT FP16 (whole graph)	194	128

FP32, 312 ms at TF32, and 121 ms at FP16. The prefix is around 95 % of the deployed chunk whichever precision it runs at, so it is the only part of this graph where optimisation effort pays. The suffix precision barely matters by comparison: with a TF32 prefix in front of it, the FP16 and FP32 SnapFlow suffixes differ by 6 ms, which lets the deployed student take the FP32 suffix and the cleaner acceptance result that comes with it (table 5.9).

SnapFlow removes 104 ms from the deployed chunk, because nine denoising steps are a small charge against a prefix that costs 312 ms even at TF32. That is why SnapFlow is worth less here than the factor of 3.6 in table 5.1, which is the eager bfloat16 path on an Orin Nano, where a step is expensive relative to the prefix. The rover policy reads three camera streams rather than two, which the prefix pays for, so SnapFlow is worth less again here than that figure suggests. The headline is a factor of 4.1 from TensorRT and distillation together, against the 10.6 a whole-graph FP16 engine would have shown, so most, though not all, of what the FP16 collapse appeared to

cost was recoverable. Prefix precision is a range rather than a switch: the reduced-mantissa mode between the two is accurate enough and worth 344 ms a chunk.

Every figure in table 5.7 is taken at a guidance weight of 1, where a single advantage-conditioned pass produces the chunk. Guidance runs both graphs a second time with the advantage token masked out (section 4.7). Measured on the Jetson at $w = 1.5$, over three runs rather than the twenty behind the other rows, it takes the eager ten-step chunk from 1357 ms to 2072 ms and the FP16 autocast chunk from 976 ms to 1106 ms, factors of 1.53 and 1.13. Guidance pays for the prefix twice, which is what makes a guidance weight expensive on this hardware. Guidance baking the desired weight into the SnapFlow distillation achieves classifier free guidance, but the weight is no longer a parameter that can be changed at runtime.

5.3.4 The control-cycle budget

A second benchmark times the whole cycle the policy controller runs, with the topic set taken from the deployed configuration, so the image decode costs what it costs in production. Table 5.8, drawn against the budget in fig. 5.5, gives the breakdown. The controller replays a chunk while it computes the next one, so inference enters the per-cycle budget amortised over its 50 actions, at 7 ms on the deployed path. The conversion of ROS messages to tensors is by then the largest item by far at 15 ms, most of which is the JPEG decode of the camera topics, followed by batching and the transfer to the GPU at 4 ms. The cycle closes at 29 ms against the 50 ms budget, so 42 % of it is free. Running ACT instead leaves 29 ms free, and the difference between the two is almost entirely inference: the ROS-side stages are the same work whichever policy is loaded, which makes them the floor of this budget rather than the policy.

The path without the distillation clears the budget as comfortably. The ten-step chunk is longer, so it amortises to 9 ms instead of 7 ms, but the cycle still closes at 29 ms and leaves 21 ms free, which is the deployed student's headroom to within a fraction of a millisecond. The distillation therefore raises the re-planning rate of table 5.7; it does not decide whether the cycle closes.

The eager column is the reference for both. Amortised over 50 actions, even a 1353 ms chunk costs 27 ms per cycle, and the cycle still fits, at 49 ms with 1 ms to spare. So the acceleration of section 4.7 is not what makes the command rate achievable; chunking already does that. It adds margin: 1 ms of headroom is one slow JPEG decode away from an overrun, against 21 ms on the deployed path. It also raises the re-planning rate below.

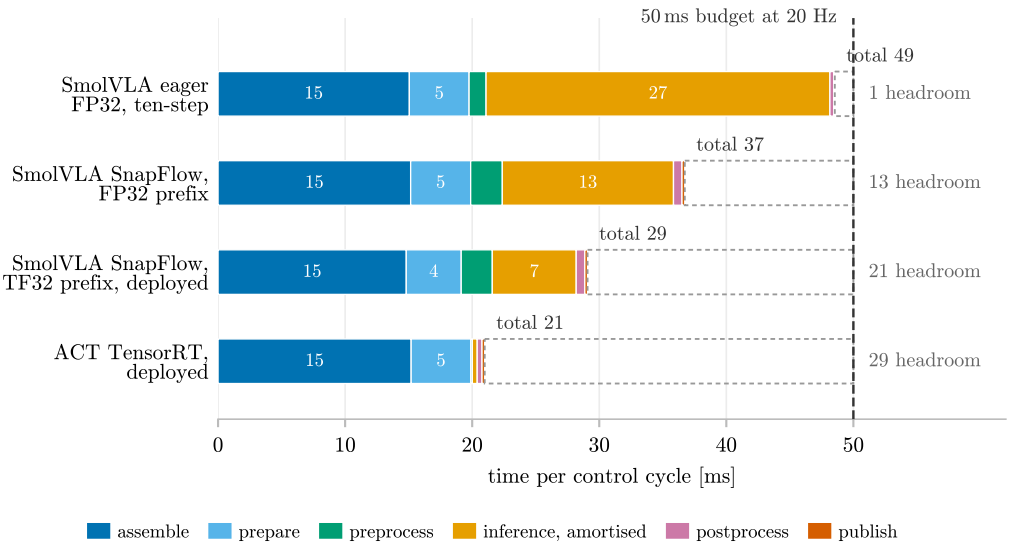


Figure 5.5: One control cycle of the policy controller against the 20 Hz deadline, for the eager ten-step path, SnapFlow on the FP32-prefix and the deployed TF32-prefix engines, and the deployed ACT baseline. The stages are additive, so each segment is what that stage adds to the cycle and the dashed remainder is the headroom. Inference enters amortised over the chunk it produces, which is why a chunk that differs by a factor of four in table 5.7 is worth 27 ms against 7 ms here, and why ACT's own inference is under a millisecond once its chunk is spread over the 50 cycles that replay it. Only the segments wide enough to hold a number carry one, rounded to the nearest millisecond as in table 5.8; the rest are in that table. The plot shows four of the table's five cycles, leaving out the ten-step sampler on its TF32 engine, whose cycle totals the same 29 ms as SnapFlow's and would draw as the same bar.

The amortisation assumes that the prediction and the publishing overlap, which is how the controller runs them, and the benchmark does not measure whether the two contend for the GPU. In-cycle inference matches the standalone chunk in every configuration measured (672.8 ms against 673.6 ms for the FP32 prefix, 333.8 ms against 330.2 ms for the deployed TF32 one, 430.0 ms against 434.3 ms for the ten-step engine), so driving an engine from the deployed observation layout costs nothing over the synthetic one.

The budget holds no progress tracker: the deployed policy will load one once it is configured (section 4.5), and the tracker consumes a frame on every tick, so it will take part of what is free here. And the benchmark has the Jetson to itself, where on the rover the rest of the autonomy stack shares the

machine, so the figures are the optimistic end of the margin rather than what a loaded rover leaves. Neither is enough to put the engine paths at risk, with about two fifths of the budget spare; the eager path, with 1 ms, has nothing to give away.

The same chunking that amortises the inference also makes most of the observation work unnecessary. Inference only runs once the action queue has drained to `action_queue_size`, so only a fraction of the ticks feed a forward pass; the rest decode three camera frames, assemble them, and throw the result away. A frame gate lets the consumer declare which ticks it can use, and the policy controller gates on exactly the condition the prediction loop tests. Measured against live topics over 30 s per arm at a one-in-25 trigger, the gate converts 24 frames instead of 600, which drops ingest and conversion from 1.23 to 0.39 cores, 68 %, or 0.84 of a core returned to the rest of the stack.[1] The gate is disabled while a progress tracker is loaded (section 4.5); recovering the saving for a strided tracker is the obvious next step, since a stride-10 window needs 2 Hz of frames and not 20 Hz.

The policy therefore meets the real-time requirement on the Jetson, and the fallback to the smaller per-task ACT models, which would have been taken had SmolVLA proved too slow, was not necessary on that count. One chunk holds 50 actions, which is 2.5 s of motion at 20 Hz, so what matters is how often the policy can re-plan. The engines sustain 3.0 Hz with SnapFlow, 2.3 Hz with the ten-step sampler, and 0.7 Hz in eager FP32, where the deployment asks for less: the 40-of-50 re-inference trigger re-plans every ten actions, 2 Hz, so the SnapFlow engine runs with a margin rather than at its limit (section 2.9). That rate bounds how quickly the policy can react to a change in the scene, and it is what the prefix's unusable precision ultimately costs: a usable whole-graph FP16 engine would have sustained about 7.8 Hz. Whether the deployed 2 Hz is enough is a question about the task rather than about the hardware: the panel is bolted down and the arm is the only thing moving in the scene, so a plan goes stale slowly and the next one arrives long before the chunk runs out. It would not be enough for a target that moves on its own.

The benchmark runs as a manual CI job on a separate build Jetson. It publishes each latency together with the GPU, torch, CUDA and TensorRT versions that produced it, so the record stays current instead of being copied by hand. The earlier measurements on torch 2.10 with TensorRT 11.1 were

[1] The per-conversion cost is unchanged, at the roughly 19 ms the assemble and prepare stages of table 5.8 cost on the deployed path, so the saving comes entirely from converting less often. The deployed 40-of-50 trigger re-plans every ten ticks and so converts 60 of the 600, avoiding 90 % of the conversions rather than the 96 % measured here, for a proportionally smaller saving.

slower throughout and reached 137 ms for SnapFlow on a FP16 engine. The newer toolchain gains more on the engines than on the eager path, so the speed-up ratios of that run do not carry over, though the conclusion is unchanged.

5.3.5 Numerical accuracy on trained weights

The conversion script checks accuracy separately, before an engine is deployed. It takes 50 samples spread evenly over the dataset, passes each through the same preprocessor, and compares the action chunk the engine produces against the eager policy's. Both paths receive the same noise tensor, seeded per sample, so the randomness of the flow-matching sampler does not enter the difference. SmolVLA is gated on the mean absolute error and ACT on the maximum, at the thresholds and for the reason given in section 4.7.

Table 5.9 reports that check on the trained checkpoints, and it is why the previous table has the shape it does. Converted whole to FP16 the policy is not usable: the mean error is more than forty times its threshold on both samplers, and the predicted actions lose most of their spread. Backing the prefix off to TF32 restores it with an order of magnitude to spare, and backing it off further to FP32 gains nothing beyond that.

With an FP32 prefix and an FP16 suffix, four of 2500 commands land on the wrong level; holding the suffix at FP16 and dropping the prefix to TF32 leaves three, which is the same result with the prefix changed underneath it; and holding the prefix at TF32 while raising the suffix to FP32 removes them altogether. It is the FP16 action expert, not the vision-language prefix, that rounds a command across a level boundary, which is consistent with the continuous error as well, since the two FP16-suffix engines agree at 2.7×10^{-3} whichever prefix feeds them and the FP32 suffix is an order of magnitude better at 1.7×10^{-4}.

The deployed student therefore takes the FP32 suffix. It costs 6 ms a chunk against the engine it replaces, under 2 %, and it passes every criterion outright. The engines behind the bold rows are what the rover runs. A rejected engine does exist on disk, since the check runs after the build, but the conversion exits with an error and packaging for the rover is a separate command issued only on a conversion that succeeded.

ACT reproduces its eager reference closely on average at either precision, and its action statistics are intact: the engine keeps the full spread of the policy's actions, where the rejected SmolVLA engines keep between a sixth and two thirds of it. At FP32 it is accepted outright, at 1.9×10^{-3} across the six joints against a 5×10^{-3} gate, with the gripper landing on the reference level every time. At FP16 it is not, and the failure is narrow rather than

diffuse: the error sits almost entirely in the sixth joint, and two of 5000 gripper commands land on the wrong level.

That joint is the one that rotates the switches, so it travels furthest and a fixed relative precision resolves it worst in absolute terms. The FP32 engine confirms the reading: the same joint is still the largest error there, at 1.9×10^{-3} against 9×10^{-4} for the next, so the ranking belongs to the channel and half precision only scales it up by a factor of nine, far enough to cross the gate. FP16 costs ACT 5.9×10^{-4} on average, concentrated in the joint that rotates the switches.

TF32 and FP16 carry the same 10-bit mantissa, so if mantissa width were the problem they would fail together. They do not: on the same checkpoint and the same fifty samples, TF32 reaches a maximum error of 2.1×10^{-3} and FP16 1.7×10^{-2}, an order of magnitude apart. What separates them is range, and the fact that TF32 rounds to those ten bits only at the tensor-core multiply while keeping an FP32 datapath and accumulator around it, where FP16 replaces the datapath as well (table 3.1). Mantissa width alone does not account for the gap. The outlier activation channels of section 4.7.4 need that range. It is the mechanism behind the SmolVLA result as well, and why a prefix that collapses at FP16 is comfortable one step up.

For ACT the choice is then not even a trade. The TF32 engine is gated at 1×10^{-2}, the tolerance the check reserves for an engine containing half precision, but its maximum error of 2.1×10^{-3} is inside the 5×10^{-3} a strict FP32 engine has to meet, and it agrees with the reference on all 5000 gripper commands. The fastest ACT backend the check accepts, 38.4 ms against 77.5 ms for the FP32 engine and 84.3 ms eager, is therefore also accurate to the FP32-class criterion, and it is the one the deployed fallback baseline runs.

5.4 Depth-Augmented Policy

The sensor is mounted in the gripper and publishes on the rover (section 2.3.3), the adapter routes it into the policy's state through one configuration entry (section 4.8), the recorded panel-switch dataset carries all 64 cells on every frame (fig. 2.6), and the policies reported above are trained with it in the observation.

Whether it improves the success rate was not measured, and nothing here is offered as an indirect substitute. The comparison that would settle it, two policies trained on the same dataset differing only in whether the grid is in the state and evaluated on a marker-free positioning task, was planned and not run. The obstacle is not the training, which is a configuration flag and a second run, but the evaluation: telling the two apart means resolving a

success-rate difference against the arm's own run-to-run variance, which needs more real-world executions than the available arm time allowed (section 5.1.5). A comparison with too few attempts behind it would have produced a number without the power to support it.

The grid is therefore kept on the cost argument of section 4.8: an engineering decision, and not a result. Section 6.3.4 describes the more interesting version of the experiment that would change that.

5.5 Workflow Automation and Agent

The agent itself is evaluated in section 5.5.7, where the on-device model is compared against the cloud model it replaces over a set of held-out prompts, with complete transcripts in chapter B. The two largest canvases discussed below are too wide for a page in the text and are reproduced full-page in chapter C.

5.5.1 The maintenance workflow

The maintenance workflow is the largest built on the node package (section 4.9.8). It is entered from a form, and each substep of the panel runs in a loop of its own that repeats until the operator reports that substep done. Every loop offers a switch to manual operation before it starts, which hands the arm back by taking the SpaceMouse node out of auto mode, so the operator heartbeat stops (section 2.5.1). The substeps that face several candidate elements of the same type (the lever switches, the rotary switches, the power switches) run the select–align–manipulate–confirm structure of section 4.9.8: the detector returns the candidates, the operator picks one, the arm aligns, the policy runs, and the outcome is confirmed before the next pass.

Two substeps do not have that shape, because each has a single fixed target rather than a choice: the plug that goes into the socket, and the rod placement that ends the sequence. Nothing has to be detected or selected there, so those loops drop the detection and alignment steps and go straight to the manipulation. The plug also does not fit the default gripper, so its substep calls the tool-change workflow of section 5.5.3 to fetch a purpose-built tool and again to park it afterwards.

The workflow has been executed on the rover. The substeps whose policies exist run through their loops as described; the others are wired but done manually while the policy is not capable of doing it.

5.5.2 A full task workflow: deep sampling

The ERC science task requires a soil core to be drilled, unloaded into a container, weighed, and documented with photographs taken before and after the manipulation (chapter 1). The workflow that runs it has 70 nodes in five phases: set parameters, drill, unload, tare, and weigh the sample, which makes it the second largest built on the node package, behind the maintenance workflow above. It was built by another team member, not by the author, which is the strongest evidence available for the code-free claim of section 2.2.4.

The workflow spans two levels that are normally separate systems. Downwards it writes the drill controller's setpoints and polls its state machine, its depth and the load cell; upwards it captures the two required photographs, creates an Asana task for the sample and attaches them to it. An operator form gates each phase before it starts, and one further form handles the case where the drill fails to retract.

Every setpoint is written with a service call. An earlier version of the workflow published to the corresponding topic instead, and messages were occasionally lost, so the drill kept its previous value while the workflow continued as if the command had been received. The controller exposes each value both as a topic and as a `SetValue` service, and the service returns a response, so the workflow only advances once the value was accepted. This is the concrete case behind the rule stated in section 4.9: a topic carries a stream, a service carries a command that sets state.

Measured runs. Three complete runs were taken from the n8n execution log. All three finished without a node error, and a run took between 111 s and 148 s end to end. Each run executed 111 to 131 node steps over 41 to 43 distinct nodes, so most nodes ran repeatedly. The repetition is polling: waiting for a hardware phase to end is expressed as a topic read followed by a condition, repeated until the drill's state machine reports the expected value. That accounts for 43 to 52 topic reads and about as many condition evaluations per run. Fixed waits for mechanical settling add a further 33 s to 40 s.

The service calls give the round-trip cost through the bridge. The 43 calls across the three runs have a median of 563 ms and a 90th percentile of 1.1 s. Their durations cluster near multiples of half a second, and a state topic read takes a median of 495 ms with a spread of a few milliseconds, which is consistent with the drill controller updating at 2 Hz. The latency of these calls is therefore set by the subsystem's own cycle and not by n8n, whose own overhead is an order of magnitude smaller: a capture that waits on nothing

costs about 120 ms end to end (section 5.5.5). Either way a workflow step is bounded well above a control period, so the loop a workflow can close runs at roughly one step per second, which is enough to sequence substeps and far from a control rate (section 5.5.6). The full canvas is in fig. C.1.

5.5.3 Tool changing

Taking a tool from the rack of fig. 5.6 and parking it again is a workflow of taught positions, with no camera and no inverse kinematics in the loop, because the rack is bolted to the chassis and the motion is the same every time (section 3.10.4). The sequence is described in section 4.9.9 and the full canvas is in fig. C.2. The rack carries two slots.

Nine of ten pickups worked. The one failure did not latch on the first attempt. Running the same procedure again picked the tool up, which is the property a deterministic sequence of taught positions has: a retry starts from the same state as the first attempt. Parking worked every time. It is the easier direction, since gravity drops the tool into its bay and the coupling only has to release.

The workflow records its own timings: a pickup takes about 30 s and a park about 20 s, the difference being the extra viewing position and the shake that seats the coupling. Both are dominated by deliberately slow motion rather than by anything computational, and both are well inside what the competition allows for a tool change. The one failed pickup therefore cost a second 30 s run and nothing else.

5.5.4 Reaching beyond ROS2: the battery warning

The battery-warning workflow of fig. 5.7 is the smallest case of n8n reaching a service outside the robot (section 2.2.4). It runs every five minutes. It first reads the power source, and stops if the rover is on external power. On battery it reads the state of charge (SoC) and compares it against a threshold. Below the threshold it checks the stored alert state, creates an Asana task if none has been raised yet, and records that it has alerted. Above the threshold it clears that state again, so the next discharge raises a new task.

The workflow is ten nodes and contains no code. Two of them read ROS2 topics over the rosbridge, three hold the alert state in a data table (section 4.9.6), and one creates the Asana task. The equivalent in a bespoke sequencer would require an Asana client, its credentials, and a place to persist the alert flag. The debouncing through the data table is what stops the workflow raising a task every five minutes for as long as the battery stays low.

5.5.5 System tests of the node package

The harness of section 4.9.11 was run once on a workstation, not on the Jetson.

Repeated image capture runs five executions in 3.1 s, of which 2.5 s are the fixed delays between them, so one capture costs roughly 120 ms including the round trip and the decode. The n8n process does not grow over the executions and ends slightly below its baseline of 785 MiB, which is the property the shared connection pool exists to provide. A burst of 100 published messages completes in 1.8 s. The harness measures the execution time but does not count how many of those messages arrive, so it does not quantify the loss described above; the evidence for that remains operational. The third test publishes to 200 distinct topics and then waits: the topic count rises from 7 to 207 and returns to 7 once the publisher's time to live (TTL) and the rosbridge unregister timeout have expired, so no publisher is left advertised. All three tests pass.

The credential guardrails of section 4.9.7 were exercised outside this harness, by workflows on the rover and in a demo environment. A read-only credential refuses publishing, service calls and container control while the observing operations continue to work, and the *Topic Publish* allowlist refuses a topic outside its permitted namespaces, in both cases with the error that names what was refused.

5.5.6 n8n as a robot orchestration layer

This section collects the practical experience of using n8n as a robot orchestration layer. It matters for deciding what to build with it and how, and when a dedicated ROS2 node is the better choice.

5.5.6.1 Rate

A workflow step costs about 120 ms of n8n overhead alone, and about a second once it waits on a subsystem that publishes at a few hertz (sections 5.5.2 and 5.5.5). Either figure is far above a 50 ms control period, so a workflow cannot run a closed control loop. Everything that runs at a higher rate needs to stay in ROS2: the 20 Hz policy loop, the alignment controller, and the heartbeat of section 4.3.

5.5.6.2 One node at a time

Within a single execution, n8n runs one node at a time and one branch to completion before the next. On a robot this can be a safety feature:

two branches cannot command the arm concurrently by accident. But the executions run depth-first (section 4.9). The branch that runs first is the upper one on the canvas, so safety-relevant sequencing is encoded in node positions rather than in edges. It also costs throughput where a step's work is independent of the arm's, such as filing a photograph while the arm moves.

5.5.6.3 Concurrency

Nothing in n8n prevents two workflows from driving the hardware at the same time. Analytical workflows can overlap safely, but drilling, driving and operating the maintenance panel each assume they have the rover to themselves. The competition setup does not reach this limit, because the rover performs one task at a time by procedure. An interlock would be needed before actuating workflows are triggered automatically. The controller switching of section 4.3 arbitrates controllers, not workflows. The remedy n8n offers, capping the whole instance at one execution, serialises the monitoring workflows as well, so mutual exclusion is better held in a data table by the few workflows that actuate.

5.5.6.4 Debugging

A test execution starts at the trigger, so changing a node near the end runs every node before it again. In web automation that costs a few API calls; on a workflow that drives hardware it costs a drill cycle per iteration, and the sampling workflow of section 5.5.2 needs over two minutes before a node in its last phase is reached at all. n8n's answer is pinned data, a frozen node output read instead of re-running the node.

5.5.6.5 Operator experience

Apart from the debugging behaviour, the editor was not a limitation, and nobody who built workflows reported a problem with it. The canvas is a visual representation of the task, so the order of operations can be read off it, and it runs in a browser behind authentication, so an operator reaches the rover from any machine on the network. ROS2 offers no equivalent, having no access control and usually needing a terminal on a machine with the workspace set up.

5.5.7 Local fine-tuned model versus cloud

The agent model on the device (section 4.9.10) was qualitatively assessed against two references: the same Qwen3.5-0.8B *before* fine-tuning, and the

cloud configuration it is intended to replace, which was exercised with both Gemini 3.5 Flash and Claude Sonnet 4.6 (section 3.10.5). All of them were run on the same 43 held-out level-1 prompts covering all eight task categories. The prompts were *reworded* from the training prompts, same intent and fresh phrasing, so the comparison measures learned behaviour instead of memorised strings. Table 5.10 summarises the results. Five complete transcripts are reproduced in chapter B, covering the cloud model and the local model before and after fine-tuning, with both a success and a failure of the fine-tuned model on the state-of-charge request.

Each prompt was run once, so the two local columns of table 5.10 rest on 43 single observations each and the cloud column pools its two models over 86. The figures characterise behaviour; they are not a statistical benchmark. A difference of a few points between two columns is a prompt or two and should not be read as significant: the 93 % against the cloud configuration's 92 % says that fine-tuning closed the tool-use gap, not that the local model overtook the cloud one. The results indicate that fine-tuning is a viable tool to make small local VLMs usable for specific tasks.

Fine-tuning fixes the core failure. The base model mostly *refuses to act*: it answers in prose, often hallucinating (asked for the state of charge, it described an unrelated "crossfire range display"), and calls a tool on only 37 % of prompts. After fine-tuning, it calls tools on 93 %, matching the cloud model's 92 %, at a similar call volume. The remaining differences are about *how well*, not *whether*, the agent acts.

Vision works on-device. Asked to capture and describe a frame, both the fine-tuned and the cloud model discovered a camera topic, captured a *real* image, and described it faithfully; each description matched its own (different, live) frame (fig. 5.8). The local vision path (camera frame returned as a tool result, read by the small VLM) works end to end.

The gap is on the hard part: discovery, efficiency, and fidelity. The cloud model remains better where the task is harder. It probes more efficiently (three CPU-load topics where the fine-tuned model probed nine) and it discovers tools more reliably under paraphrase. Asked to "read out the current SoC value", the fine-tuned model guessed four names (`sohc_state`, `energy`, `battery`, `power`), subscribed to each in turn, and gave up when none of them carried a message. The cloud model on the same request tried `battery` and `power`, got nothing, then searched the available topics for `soc`, found `/wecant/AFB/SoC_1/Value` and read it. Only a literal match on the

acronym reaches that topic. Two of the six power prompts were answered correctly by the fine-tuned model, both times because it grepped the available topics for `soc` first. Chapter B reproduces one of those successes alongside the failure. What separates the successes from the failures is therefore whether the model happens to try the right substring, not whether the topic is within its reach. The fine-tuned model also occasionally runs long without converging (sixteen calls on one prompt, inventing topic names, with corrupted tokens spliced into a path) and fumbles argument schemas: asked to turn the beacon red, it guessed a `Value` field instead of checking the schema first, and although a later retry succeeded, its final answer reported failure and cited a service that never appeared in any tool result (the full transcript is in chapter B). The cloud model checks the schema before calling and accurately reports what it did.

5.6 Manipulator Inverse Kinematics

The inverse kinematics is FZI's Forward-Dynamics Compliance Control solver [47], used as it ships, and the kinematic model it solves against is the mechanical team's URDF. Neither is this project's work. What was added is the per-joint weighting that stops the forearm roll from absorbing wrist rotations, the dynamic joint limits evaluated in the control loop, and the leash on the integrated target (section 4.10), which are adjustments to a working solver rather than a solver of its own.

The kinematic chain the solver is given has to be true enough for the vision pipeline built on top of it, which is what section 5.2.2 reports: the chain is true at the centimetre level, which is as fine as a ruler check resolves. A centimetre is the right order for the present use cases.

The IK available on the rover's manipulator in the previous project "only works in a limited region" [17]; the new IK removes that limitation. The arm reaches the panel across the working volume the maintenance task requires without drifting. The reach is demonstrated by the rest of the work in this chapter. A fuller characterisation, covering reachable workspace, tracking error under load and behaviour near singularities, belongs to a mechanical evaluation of the arm rather than to this report.

5.7 Performance on the Rover System

All numbers in this section come from laboratory testing on the maintenance-panel replica (section 2.1.1). Due to hardware issues and communication

problems with the onboard cameras, testing the policy systematically was not possible. This report is submitted before the European Rover Challenge, so no competition results exist yet.

The tasks themselves are easier than the LIBERO ones. The panel elements sit at known positions, the motions are short, and the scene does not change between runs. What makes the rover harder is the custom hardware. A failure can come from the arm, a camera or a sensor or from the policy. Real sensors are prone to failure, camera mounts have been bent under tensile load, and wrong offsets in the encoder lead to misaligned motor joints. So a policy failure could be attributed to any of these reasons, or a defect in the policy itself.

The policy works on the tasks it was trained for and in the region it was trained in. Driven from the taught starting pose, it is able to turn the panel's switches. The deployed policy is trained on the combined dataset with demonstrations for different tasks (table 2.1), so it is a multi-task policy. This demonstrates the pipeline in full, from teleoperated recording through annotation, critic training, advantage-conditioned fine-tuning, distillation and TensorRT export to a policy driving the manipulator on real hardware.

Success dropped when the policy was trained on indoor scenes only and tested outdoors, and it also weakened when the panel was placed in a new region. Photometric augmentation during training recovers part of that: as on the OBJECT suite, the policy fits the exact appearance of its training images (section 5.1.4). The augmentation is only a mitigation; the proper fix is to record a diverse dataset under different conditions.

Rotary switches work well enough to be useful. Roughly four attempts in five turn the switch, which is below what a teleoperating human manages. A failed attempt costs only a retry from the same taught pose, however, so the substep can be repeated until it succeeds. This is the substep with the most demonstrations behind it, since it is also the easiest for a human operator to achieve.

Lever switches are harder for the policy, and hard for a human operator too. The motion is not a steady push but a quick one with tensile load: the switch takes tension and then snaps once enough of it has built, so the useful part of the demonstration is short and the moment of release is what has to be imitated. The collected dataset also has less demonstrations of this task, which can also explain part of why the policy struggles.

Inserting the plug has not been trained or validated. It was excluded because the task could not be teleoperated well enough to demonstrate, so no training data existed (section 6.2.4). A purpose-built tool now exists that holds the plug rigidly (fig. 5.9) and rotates it when grasping. The substep appears learnable by a policy once demonstrations can be recorded through

the tool. That has not been tried due to limited time, but it could be tried until the ERC.

The approach phase is the reliable half, by construction and by measurement (sections 5.2 and 5.2.1): it places the gripper inside the 25 mm the substep needs, and it is retried from a known pose when it does not. The failures that remain are in the manipulation phase executed by the policy.

The stopping layers. All three software layers of section 2.1.4 were exercised on the running rover. With a policy executing and the SpaceMouse node in auto mode, toggling that node out of auto mode stops the operator heartbeat, and the arm stops commanding within a second. The software e-stop was triggered on the running system and aborted the active goal as specified, in addition to the unit tests that cover its latch semantics (section 4.3.2). Goal cancellation, the topmost layer, was exercised throughout development: phase actions were cancelled routinely from the dashboard and from the workflow layer, and the phase ends cleanly with control returned to the caller. This is a functional check and not an instrumented latency measurement: what section 2.1.3 states is a bound rather than a figure, and the observed behaviour is inside it.

Table 5.8: Per-stage cost of one control cycle of the policy controller in milliseconds, averaged over 20 runs, driven from the deployed topic configuration in every case. Every value is rounded to the nearest millisecond, so the stage rows do not visibly sum to the total, and the stages that stay under one are given as < 1 rather than as zero. The first two columns are the ten-step sampler, in eager FP32 and on its qualified TF32-prefix engine; the next two are SnapFlow on the FP32-prefix and on the deployed TF32-prefix engine of table 5.7; the last is the deployed ACT baseline on its TensorRT engine. The stage names are those of the controller's pipeline, and preprocessing covers normalisation and tokenisation. Inference is amortised over the actions the controller consumes, 50 for both policies, since the ACT configuration takes the first 50 of the 100 it predicts (section 2.9), and the chunk itself costs 1353 ms eager, 430 ms for the ten-step sampler on TF32, 674 ms on the FP32 prefix, 330 ms on the deployed TF32 prefix and 38.4 ms for the ACT engine. All four SmolVLA cycles were recorded on the engines they name. The ACT cycle was recorded on a whole-graph FP16 engine, so its inference row is the deployed TF32 engine's chunk divided over its actions while the surrounding stages are as measured; those stages do not depend on the engine's precision.

stage	ten-step eager	ten-step TF32	SnapFlow FP32	SnapFlow TF32	ACT TF32
assemble (JPEG decode)	15	15	15	15	15
prepare (batch, to GPU)	5	3	5	4	5
preprocess (normalise)	1	1	2	2	< 1
inference, amortised	27	9	13	7	< 1
postprocess (to host)	< 1	< 1	< 1	< 1	< 1
publish (to ROS)	< 1	< 1	< 1	< 1	< 1
total per cycle	49	29	37	29	21
budget at 20 Hz	50	50	50	50	50
headroom	1	21	13	**21**	29

Table 5.9: Action-chunk error of each engine against its eager reference, over 50 samples spread across the dataset with per-sample seeded noise, on the trained checkpoints. The RECAP rows are measured on the rover policy and the panel-switch dataset of fig. 2.6, as is ACT; only the plain-SmolVLA row is a LIBERO checkpoint (section 5.1), which is why it is not comparable with the RECAP rows above it. The gripper column counts commands whose rounded level differs from the reference, over all 50 chunks. *Two SnapFlow engines clear the mean criterion by an order of magnitude and miss the zero-mismatch rule on the gripper by three or four commands in 2500; neither is deployed, because a TF32 prefix with an FP32 suffix passes outright at nearly the same cost. Deployed engines are set in bold. A single precision in the engine column means the whole graph; two mean the prefix and the suffix engine respectively, which the SmolVLA family splits and ACT does not have (section 4.7). The gate is on the mean for the SmolVLA family and on the maximum for ACT, and the last column says whether the engine met it; both error columns are given for every row so the gate can be checked against the criterion it is not applied to. Both are taken over the continuous action dimensions. ACT's seventh dimension is the gripper, which is rounded to $\{-1, 0, 1\}$ before publishing and is therefore checked by level agreement instead: the deployed TF32 engine matches the reference on all 5000 commands, and the FP16 engine misses two. The action-spread column is the standard deviation of the engine's actions as a percentage of the eager policy's, over the same samples: an engine that has stopped tracking the policy flattens rather than simply erring.

policy	engine	mean abs	max abs	gate	action spread	gripper	accepted
RECAP ten-step	FP16	0.826	3.774	mean 2×10^{-2}	17	252/2500	no
RECAP ten-step	TF32/FP16	0.0014	0.028	mean 2×10^{-2}	100	0/2500	**yes**
RECAP ten-step	FP32/FP16	0.0013	0.028	mean 2×10^{-2}	100	0/2500	yes
RECAP ten-step	FP32	0.00000	0.000	mean 1×10^{-2}	100	0/2500	yes
RECAP SnapFlow	FP16	0.897	4.907	mean 2×10^{-2}	62	344/2500	no
RECAP SnapFlow	TF32/FP16	0.0027	0.023	mean 2×10^{-2}	100	3/2500	no*
RECAP SnapFlow	TF32/FP32	0.00017	0.001	mean 1×10^{-2}	100	0/2500	**yes**
RECAP SnapFlow	FP32/FP16	0.0027	0.022	mean 2×10^{-2}	100	4/2500	no*
RECAP SnapFlow	FP32	0.00000	0.000	mean 1×10^{-2}	100	0/2500	yes
SmolVLA	FP16	0.549	3.797	mean 2×10^{-2}	67	–	no
ACT	FP16	0.00059	0.017	max 1×10^{-2}	100	2/5000	no
ACT	TF32	0.00014	0.002	max 1×10^{-2}	100	0/5000	**yes**
ACT	FP32	0.00006	0.002	max 5×10^{-3}	100	0/5000	yes

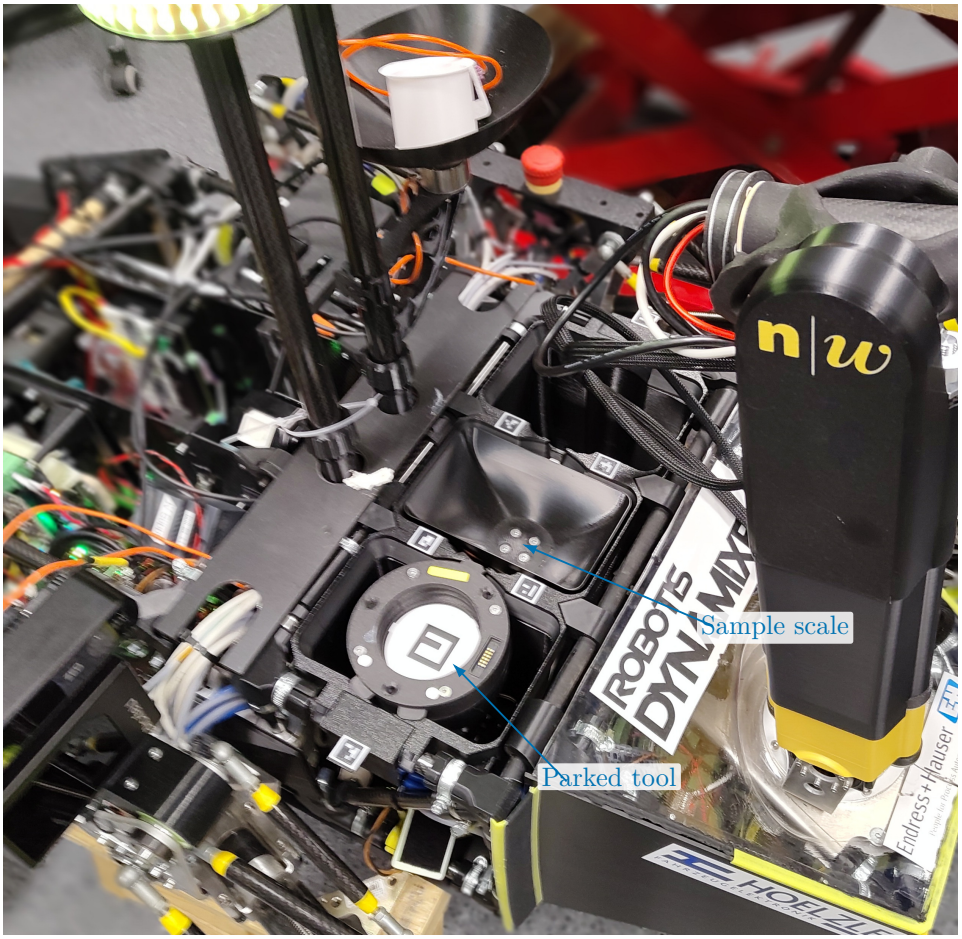


Figure 5.6: The tool rack, with one tool parked in its bay. The rack is bolted to the chassis, which keeps the workflow's taught positions valid from one run to the next. The scale beside it belongs to the science task and weighs a soil sample (section 5.5.2).

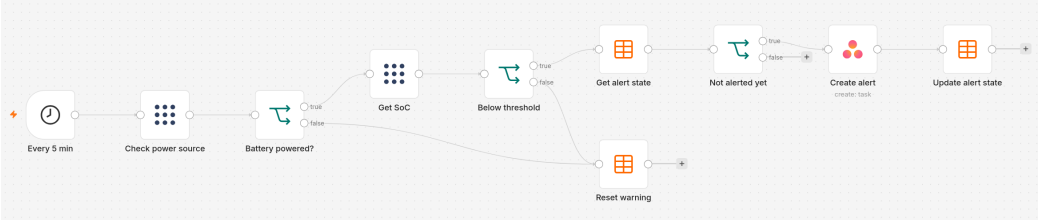


Figure 5.7: The battery-warning workflow on the n8n canvas. The schedule trigger and the two ROS2 topic reads are on the left, the alert state is held in data tables, and the Asana task is created only if no alert is open.

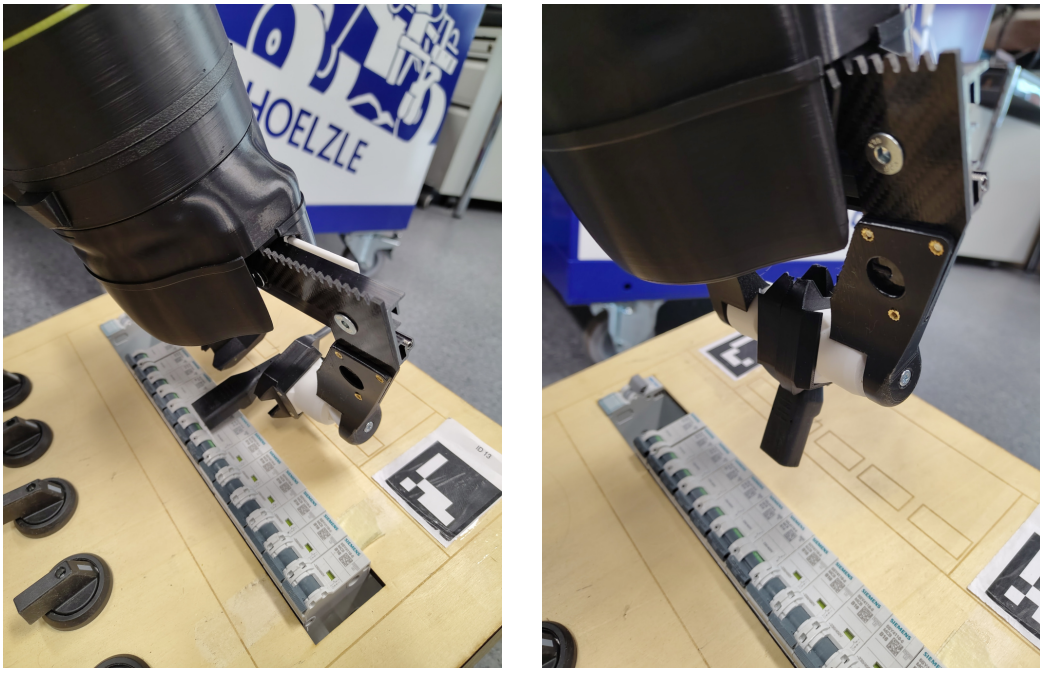
Table 5.10: On-device agent model on 43 reworded held-out prompts, compared with the untrained base model and with the cloud configuration, whose column pools Gemini 3.5 Flash and Claude Sonnet 4.6 over 86 runs. Fine-tuning closes the tool-*use* gap almost entirely.

Metric	Untrained	Fine-tuned	Cloud
Prompts with $\geq$1 tool call (%)	37	93	92
Mean tool calls per run	1.0	5.2	5.5

"The foreground is a blue, curved frame containing a person in an orange shirt kneeling on the ground and another person in a grey shirt and shorts standing; the workbench, tools, equipment, and lab environment are visible."

"Looking upward at what appears to be the underside of a large structure or panel — possibly a ceiling, an overhead platform, or the belly of another vehicle/machine."

Figure 5.8: Vision through tool calls, on-device (left) versus cloud (Claude Sonnet 4.6, right). Both agents were asked to capture a frame and describe it, discovered a camera topic themselves, and captured a live frame at different moments. Each description (quoted beneath its frame) matches its own image: the fine-tuned local model reads the lab scene, the cloud model notes its dark, upward-facing shot (which is an artifact of the camera being mounted upside down).



(a) Open, with the plug lying flat between the jaws.

(b) Closed, with the plug turned upright and its pins presented downwards.

Figure 5.9: The tool built for the plug substep, mounted on the manipulator above the panel replica. The jaw rotates through 90° as it closes, so a plug picked up lying flat comes out of the grasp already upright and aligned with the socket. That removes the regrasp of section 2.1.1, which is the part an operator cannot reliably teleoperate (section 6.3.2). The tool exists but has not been used to record demonstrations.

Chapter 6

Conclusions

A language-conditioned policy runs on the rover's own compute inside the control budget. An operator drives the whole stack through workflows a non-programmer can compose, and the classical approach phase places the gripper accurately enough for the policy to take over. Whether reinforcement learning makes the policy better *on this arm* could not be established, because time on the arm was not available. It has been shown to work in a benchmark simulation.

6.1 Goal Achievement

Each contribution is assessed here against what it was meant to deliver, and the standard applied is deployment on the rover rather than a working prototype. The three research questions come first, because each of them is answered by more than one contribution.

6.1.1 The research questions

The questions are those set out in the introduction, answered in the order they were posed.

How can the maintenance task and further ERC tasks be automated, and how can reinforcement learning be used to train more capable and robust manipulation policies? The task is split into an approach phase and a manipulation phase, each automated with a different tool. The approach phase is classical: visual servoing against markers, which is what makes a repeated, ambiguous target reachable (section 6.1.6). The manipulation is a language-conditioned policy, so one model covers several

substeps instead of one per substep. Reinforcement learning enters that phase as post-training on the policy's own rollouts rather than as a control method (section 6.1.2). Above both sits n8n, which generalises the answer beyond manipulation. A task becomes a workflow of ROS2 calls, with no learned component where none is warranted, as the deep-sampling drill sequence shows (sections 5.5.2 and 6.1.5).

How can the gripper be positioned autonomously and precisely using depth information when no fiducial markers are available? Not by a result. The route taken is to hand the policy the depth and let it learn what to do with it: the in-gripper time-of-flight grid enters the state as 64 values on every frame (section 6.1.4), with no term that rewards using them and no ablation to show whether the policy does (section 5.4). What is deployed and measured is the marker-based approach phase, and it needs three markers on the panel to work (section 4.2), which is the case the question excludes. The target switch itself carries no marker and is found by a colour-and-contour detector, so the question is answered for the *element* and not for the environment. Positioning the gripper with no fiducial in view remains untested.

How can errors, unknown states, and changes of plan be handled safely and purposefully during autonomous operation? Safety sits below the autonomy. An operator heartbeat means the absence of a signal stops motion rather than the presence of a command, a latching software e-stop aborts on command and refuses to restart until it is reset, and the hardware stop is underneath both (section 2.1.4). Nothing autonomous moves without them, whichever controller is driving. Here *autonomous* is meant as section 2.1.3 defines it: a controller deciding its own commands. A move between joint configurations taught beforehand is not that, since every pose in it is fixed in advance and the sequence is released by a human through a form or by a workflow, and it is covered by the hardware stop alone. Errors and unexpected states can be handled with n8n, by branching based on the outcome of the actions that were called. Deployed are the hold, an operator confirmation form before anything unsafe, and the escalation for a drill that fails to retract (section 5.5.2).

6.1.2 SmolVLA and reinforcement learning

The contribution was to replace the per-task policies of P8 with one language-conditioned model and to improve that model with reinforcement learning rather than with more demonstrations.

The quantitative claim comes from the LIBERO benchmark. Critic-guided rollout post-training, published on a model an order of magnitude larger, transfers to a 450M-parameter policy: the best configuration reaches 68.1 % average success across three suites against 62.9 % for the same policy without it, and 55.3 % for the architecture trained the conventional way (section 5.1). Both findings below go beyond the original recipe. Training on the policy's own rollouts alone degrades the long-horizon suite, and mixing the expert demonstrations back into fine-tuning undoes that: in a setting without human corrections, the demonstrations take over the role those corrections play. The labelling threshold then decides whether classifier-free guidance helps at all: label the necessary motions negative and guidance steers away from them, reliably and in proportion to its weight. One-step distillation costs 2 points of average success at the student's best guidance weight. It reduces inference time and thus raises the re-planning rate (section 5.3).

On the rover arm the claim is narrower. The policy runs on the manipulator and performs the panel-switch task it was trained for; its main weakness is its susceptibility to changing environments. Photometric augmentation in training recovers part of that, but a more robust model requires a broader training dataset. What could not be established is whether the reinforcement learning helps on the rover: too few rollouts fit into the available time on the arm to separate the method from the hardware's own variance. What the arm inherits from this contribution is the architecture and advantage conditioning, not the RECAP loop that produces the gain.

Doing that requires collecting the policy's own rollouts on the arm, labelling them with the critic, and fine-tuning on them. Collecting them both autonomously and with human interventions is worth trying, since the interventions are what the original recipe uses to cover the states the policy reaches and the demonstrations do not. Competition preparation may leave no time for it beforehand, but the competition itself is a source of exactly that data: recording the ERC runs as rosbags makes them convertible into training input afterwards (section 4.4.3).

6.1.3 Deployment on the Jetson Orin

This contribution is finished. The policies meet the real-time budget on the Jetson Orin, with the margins reported in section 5.3 (the deployed cycle

closes at 29 ms of the 50 ms budget, leaving 21 ms free), but only with the right precision configuration. Converting the whole graph to FP16 produces an engine whose actions collapse on trained weights, which makes it unusable. The deployed engines therefore reduce the prefix's precision without going all the way to half precision: TensorRT's TF32 mode keeps FP32's exponent range while truncating the mantissa, which preserves its output close enough to the eager policy.

The TensorRT engines are loaded by the live policy controller behind the `use_trt` switch, with a second switch for the progress tracker, and the rover runs its trained SmolVLA policy on the arm through that path (section 5.7). The smaller per-task ACT models would have been the fallback had SmolVLA proved too slow on the Jetson; the real-time budget did not need them, though they stay deployed as a quick alternative.

6.1.4 Depth sensing

This contribution is integrated but has not been ablated. The in-gripper ToF grid is mounted and publishing on the rover, present on every frame of the recorded datasets, and also part of the observation the deployed policy was trained on (section 4.8). Extending a policy's observation with a new sensor is a configuration change, not an engineering project. A comparison between a policy with and without depth data was not run. There is little reason to expect that adding sensor values harms the policy.

6.1.5 Workflow automation and agent

The n8n integration exposes ROS2 topics, services, and actions as drag-and-drop nodes usable by both operators and an LLM agent. The agent was then moved off the cloud onto the rover: a Qwen3.5-0.8B model, fine-tuned on the agent's own executions, is served locally and drives the same tools. Fine-tuning closed the tool-*use* gap, from tool calls on 37 % of prompts to 93 % against the cloud configuration's 92 %, and single-step diagnostic tasks such as capturing a camera frame and describing it are handled on-device (section 5.5.7). What remain are the harder parts: multi-step planning, efficient probing, correct argument schemas, and tool discovery. These scale with data and model size, so the natural next step is more collected trajectories in the weaker categories and a larger model.

6.1.6 Visual servoing

The approach phase meets the requirement it was built for. Driving the gripper to a switch from the base camera puts it within the required tolerance in the panel plane, at a median of 18.1 mm on a stored registration, which is the condition the rover runs in. This allows the policy to manipulate directly, without having to choose between five switches.

The manipulator's forward kinematics, based on the motor encoders' reported positions, turned out to be a reliable source for the gripper's movement. In contrast, the vision-based detection degrades when the gripper moves further away from the camera. The exact reason for this has not been diagnosed; a bad camera calibration was ruled out.

The arm is accurate to a centimetre, and what visual servoing compensates for is a monocular pose estimate that degrades with reach. That the estimate degrades is measured; which side of the chain it comes from is not, since reach and camera range are almost perfectly correlated in the available recording (section 5.2.2). The registration works because the markers are used where they are most reliable (solving the registration from a viewpoint chosen for the purpose), and the arm is then trusted to carry that registration to a pose where markers are less reliable.

6.1.7 Manipulator inverse kinematics

The solver is FZI's and the kinematic model is the mechanical team's; what was added is the per-joint weighting, the live joint limits, and the leashed target of section 4.10. The result is that P8's "the IK only works in a limited region" [17] is no longer the binding constraint. That the arm reaches the panel across the volume the maintenance task needs is a prerequisite of the rest of the project: every run reported here was driven to the panel by this solver. What was measured is that the kinematic chain agrees with a ruler measurement to about a centimetre (section 5.2.2). The remaining error is belt backlash and structural flex, a hardware property that no controller change removes.

6.2 Lessons Learned

The following are the most important things learned during the project.

6.2.1 Edge AI for robotics is viable

Two models that would have been cloud workloads only a few years ago now run on two Jetson Orin NX devices on the rover. One is a 450M-parameter flow-matching VLA driving the arm inside the open-loop control-cycle budget (section 6.1.3); the other is a sub-billion-parameter agent model answering operator questions by calling ROS2 tools (section 6.1.5). The policy is deployed and run on the arm's Jetson under the cycle budget measured in sections 5.3 and 5.7. The agent model is served on the second Jetson, where nothing binds it to a control period: it answers in text and calls diagnostic tools, so its latency was never measured against a budget the way the policy's was.

The policy as trained only just runs: its eager cycle closes at 49 ms of the 50 ms budget and it re-plans at 0.7 Hz (section 5.3). Reaching the deployed margin took a TensorRT export path built and validated against an eager reference, a distillation stage to cut the sampler to a single step, and a numerical check on every engine before it was trusted.

6.2.2 Benefit and limitations of fiducial markers

Fiducials are cheap and metric, a wrong reading can be inspected in the image that produced it. That last property makes errors tractable: every problem that cost real time (the two-fold pose ambiguity of a single square marker, the tool-axis offset left at zero, the jaw markers moving when the gripper opens) was traceable, and each turned into a measurement rather than a hyperparameter (section 6.1.6).

A single fiducial facing the camera head on is easy to detect and among the worst to measure: its four coplanar corners admit two poses that reproject almost identically. On the rover's gripper markers, the lever arm to the tool frame turned that into centimetres of position error flipping at random between frames. Two markers mounted at 90° to each other, each seen at roughly 45°, detect less reliably but solve far better. Pooling their corners into one fit removes the second solution outright (section 4.2.1). So mount fiducials in pairs at an angle and size the detector for the harder of the two views.

6.2.3 Mechanical tolerance beats control accuracy

The tool change is the clearest case. A taught joint position leaves a residual misalignment at the coupling, and pressing straight down on it can jam rather than engage. Nothing in the software removes that error: the coupling has a lead-in, and oscillating the wrist while pressing lets the geometry pull the two

halves together. The whole exchange therefore runs from positions taught once and stored in a table, with no camera and no inverse kinematics in the loop, and nine of ten pickups succeeded, with the one failure engaging on a retry from an unchanged state (section 5.5.3).

6.2.4 You can only imitate what you can demonstrate

Inserting the plug into the socket cannot be teleoperated reliably with the current gripper, so no demonstrations of it exist and nothing could be trained on it. The learning method is not the limitation, the available training data is. This bound is invisible in a benchmark, where the demonstrations arrive with the suite. On a real robot the data-collection interface is part of the learning system, and improving it, a better tool or a better teleoperation rig, can matter more than improving the architecture of the policy.

One route could sidestep the demonstration problem altogether. Reinforcement learning from scratch in simulation needs no human demonstrations, so the teleoperability bound does not apply. But a contact-rich insertion, with a plug, a socket and a cable free to swing and deform is hard to simulate accurately. That is a harder modelling problem than the one the existing MuJoCo scene covers (section 4.6). Because the policy also uses vision, the scene needs have to render well enough for the visual features to transfer to the real world. This would require photorealism rather than just a model with the right dimensions. Training a policy purely from simulation with no demonstration is less efficient, since the exploration space of the policy is much bigger and the learning signal weaker. So a simulation would need more steps and those steps are more expensive, since every step needs to be simulated and rendered, which means longer and more expensive training.

6.2.5 The reach and the rate limit of n8n

The value of the integration turned out to be breadth. One bridge is enough for ROS2 to reach a whole range of external tooling: the battery-warning workflow raises a task in an issue tracker from a ROS2 topic without a line of code (section 5.5.4) and without setting up a cron job or systemd service and timer. The deep-sampling workflow reaches the other way, from a drill's setpoints to photographs filed against a sample (section 5.5.2). The latter was built by someone who did not write the node package.

Single-threaded execution is a limitation of n8n. It is a genuine safety property, since two branches cannot drive the arm at once, but it is also why sequencing ends up encoded in where nodes sit on the canvas. n8n belongs above the control loop, used for sequencing substeps, setting parameters,

reading back measurements, and connecting the robot to the outside world. Anything inside the control loop stays in ROS2.

Several of the rules found in this project are invisible in the editor and were learned by running into them. A command that sets state goes through a service. The output of a node is pinned before a workflow is debugged, so that a test run does not drive the hardware again. An actuating workflow takes a lock in a data table first. A short manual tailored to this setting, n8n against ROS2 on a rover, would pass them to the next team, and writing it before the next competition cycle would probably help more than another node in the package.

6.3 Future Work

The following lists some improvements and experiments that can extend this work.

6.3.1 Real-time chunking

The policy controller stitches consecutive action chunks together by blending the overlap with a weight that decays across it (section 4.4.2), which stops the arm jumping when a fresh prediction disagrees with the actions still queued. The blend is applied to the finished chunk, so a disagreement is averaged away after the fact rather than avoided.

Real-time chunking (RTC) [9] is the version of this built for flow-matching policies. Instead of mixing two chunks afterwards, it treats the start of the new chunk as an inpainting problem and adds a guidance term *inside* the denoising process, so the overlapping timesteps are pulled towards the portion of the previous chunk the robot has already executed, under a soft mask that decays over the overlap. The new chunk is generated to continue the old one rather than corrected into agreeing with it, and the policy stays free to react outside the overlap.

Both of this project's inference optimisations conflict with this approach: RTC needs the denoising trajectory, and both optimisations remove it. SnapFlow distils the ten-step sampler into a single step (section 4.6.3), and a single step has no intermediate states for a guidance term to act on. The TensorRT export freezes the suffix into an engine with fixed inputs (section 4.7), so the additional conditioning RTC needs (the executed tail, the mask, the guidance weight) has no way in without re-exporting the graph around it. Either the multi-step sampler runs in eager mode, which would void the latency optimizations the distillation was introduced to avoid. Or a suffix

engine takes the guidance inputs with the loop kept in Python around it, which the current export structure already does for the ten-step sampler. The second would fit the current implementation better and should be tried first.

6.3.2 Demonstrating and learning the cable-insertion task

Inserting the plug into the socket does not work with the regular gripper. The plug needs to be picked up in a different pose than it needs to be inserted in. With the regular gripper this does not work, as the gripper itself would then collide with the panel. No policy has yet been trained for this substep, due to missing human demonstrations.

A tool built for this substep can do the reorientation mechanically: its jaw rotates through 90° as it closes, so a plug picked up lying flat points downwards and is aligned with the socket when the grasp completes (fig. 5.9). The tool is built and can be used to insert the plug manually, so it is now possible to record and train this substep as well. Changing to the right tool during the maintenance task is already solved by the maintenance workflow built in n8n.

6.3.3 An ablation for the depth grid

The grid is in on every frame of every recorded dataset and in the observation the deployed policy was trained on, but no experiment separates a policy that sees it from one that does not (section 5.4). The experiment can be performed with the collected dataset: A policy needs to be trained with the same parameters on the same recordings, once with the 64 cells in the state and once without. Those two policies can then be evaluated on real world performance to determine whether adding the depth sensors actually helps the policy. Until this is done, the depth grid is kept, since it requires only a configuration and is cheap compared to the whole processing step.

6.3.4 A spatial encoder for the depth grid

The ToF grid enters the policy as a flat vector of 64 values, so the arrangement of the cells is lost before the model sees it (section 4.8). An encoder that reads the 8×8 layout would keep it. The grid is a low-resolution depth image, so a small convolutional encoder is the natural choice, and it would be trained together with the policy. The change is confined to the observation pipeline and leaves the action expert untouched.

A flat state vector becomes impractical once the sensor has a few hundred zones, while a convolutional encoder produces a fixed-size embedding at any resolution. Whether it pays off at 8×8 remains open: the grid is small enough that the linear projection may already be sufficient.

6.3.5 A shared LeRobot–ROS2 interface

The adapter used here (section 4.4) is one of several independent bridges between LeRobot and ROS2, and none of them is official: LeRobot leaves ROS2 integration to the community, and the projects that filled the gap picked incompatible shapes: LeRobot inside ROS2, as in this project, or ROS2 behind a LeRobot plugin. A request for comments opened in the LeRobot repository in August 2026 surveys these projects and proposes a common direction, from documentation and a rosbag-to-dataset converter up to running a policy as a node.[1] The transport question (rclpy-native, Zenoh, or a separate bridge process) is still open. The adapter of this project is one of the surveyed projects and takes part in that discussion. A measurement contributed from the rover showed that for image topics the encode and decode cost dominates while the transport cost is negligible, which removes performance as an argument for one transport over another. Whether this adapter converges on the shape eventually agreed on remains open. It would be worthwhile: a shared interface means the recording, training and deployment tooling works on any ROS2 robot without a project-specific bridge in between.

6.3.6 Improving ROS2 for agentic AI

Tool discovery is the most obvious limitation. The agent finds tools by listing ROS2 topics and services and matching them by name, so discovery is only as good as the names. The state-of-charge case shows this: asked about the battery, the model searches for names like `battery` and `energy` and never reaches `/wecant/AFB/SoC_1/Value`, the topic that carries the charge. On other phrasings it searches `soc` and succeeds, so what decides the outcome is whether the right substring is tried (section 5.5.7). This is partially a limitation of the model itself, but also of the naming conventions used for topics on the rover and a limitation of ROS2 topic inventory management.

Several things can be done to improve it: (1) *semantic* topic discovery, matching a query against topic descriptions and not just on a substring of the name, so that "battery" can reach `SoC_1`; (2) a machine-readable *system description* that states what each subsystem and topic means, instead of

[1] `https://github.com/huggingface/lerobot/issues/4368`

leaving the agent to infer intent from identifiers; and (3) predictable, descriptive topic names. The charge and the bus voltage sit under `/wecant/AFB/` as `SoC_1` and `Input_Voltage_1`, where the prefix names the board rather than what it does, alongside topics called `MrCrap_A_2`. A convention such as `power_delivery_system/state_of_charge` costs nothing when the topic is first published.

Bibliography

[1] *AI Workflow Automation Platform — n8n*. URL: https://n8n.io/ (visited on 08/19/2026).

[2] Ali Amin et al. $\pi^*_{0.6}$*: a VLA That Learns From Experience*. 2025. arXiv: 2511.14759 [cs.LG]. URL: https://arxiv.org/abs/2511.14759 (visited on 08/19/2026).

[3] Jason Ansel et al. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Vol. 2. ACM, Apr. 2024. DOI: 10.1145/3620665.3640366. URL: https://docs.pytorch.org/assets/pytorch2-2.pdf (visited on 08/19/2026).

[4] Anthropic. *Introducing the Model Context Protocol*. 2024. URL: https://www.anthropic.com/news/model-context-protocol (visited on 08/19/2026).

[5] Anthropic. *System Card: Claude Sonnet 4.6*. Feb. 2026. URL: https://www.anthropic.com/claude-sonnet-4-6-system-card (visited on 08/19/2026).

[6] Alex Becker. *Kalman Filter from the Ground Up*. Second. Sept. 2023.

[7] Marc G. Bellemare, Will Dabney, and Rémi Munos. "A Distributional Perspective on Reinforcement Learning". In: *Proceedings of the 34th International Conference on Machine Learning (ICML)*. 2017. arXiv: 1707.06887 [cs.LG].

[8] Loubna Ben Allal et al. *SmolLM2: When Smol Goes Big — Data-Centric Training of a Small Language Model*. 2025. arXiv: 2502.02737 [cs.CL]. URL: https://arxiv.org/abs/2502.02737 (visited on 08/19/2026).

[9] Kevin Black, Manuel Y. Galliker, and Sergey Levine. *Real-Time Execution of Action Chunking Flow Policies*. 2025. arXiv: 2506.07339 [cs.RO].

[10] Gary Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* 25.11 (2000), pp. 120–125.

[11] Remi Cadene et al. *LeRobot: State-of-the-Art Machine Learning for Real-World Robotics in PyTorch*. `https://github.com/huggingface/lerobot`. 2024. URL: `https://github.com/huggingface/lerobot` (visited on 08/19/2026).

[12] François Chaumette and Seth Hutchinson. “Visual Servo Control. I. Basic Approaches”. In: *IEEE Robotics & Automation Magazine* 13.4 (2006), pp. 82–90. DOI: `10.1109/MRA.2006.250573`.

[13] Jessie Y. C. Chen and Jennifer E. Thropp. “Review of Low Frame Rate Effects on Human Performance”. In: *IEEE Transactions on Systems, Man, and Cybernetics — Part A: Systems and Humans* 37.6 (Nov. 2007), pp. 1063–1076. DOI: `10.1109/TSMCA.2007.904779`.

[14] Lawrence Yunliang Chen, Simeon Adebola, and Ken Goldberg. *Berkeley UR5 Demonstration Dataset*. Released as `berkeley_autolab_ur5` in the Open X-Embodiment collection. 2023. URL: `https://sites.google.com/view/berkeley-ur5/home` (visited on 08/19/2026).

[15] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI: An Introduction*. CRC Press, 2018. DOI: `10.1201/9780429489105`.

[16] Toby Collins and Adrien Bartoli. “Infinitesimal Plane-Based Pose Estimation”. In: *International Journal of Computer Vision* 109.3 (2014), pp. 252–286. DOI: `10.1007/s11263-014-0725-5`.

[17] Sandro Covo. *P8 Mars Rover Manipulator*. Project report. FHNW University of Applied Sciences and Arts Northwestern Switzerland, School of Engineering, 2026.

[18] Christopher Crick et al. “Rosbridge: ROS for Non-ROS Users”. In: *Robotics Research: The 15th International Symposium (ISRR)*. Vol. 100. Springer Tracts in Advanced Robotics. Springer, 2016, pp. 493–504. DOI: `10.1007/978-3-319-29363-9_28`.

[19] Tim Dettmers et al. “LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 35. 2022, pp. 30318–30332.

[20] Danny Driess et al. *Knowledge Insulating Vision-Language-Action Models: Train Fast, Run Fast, Generalize Better*. 2025. arXiv: `2505.23705 [cs.LG]`.

[21] European Space Foundation. *RULES On-Site Formula ERC2026*. Tech. rep. Krakow: European Space Foundation, Aug. 2026, pp. 38–43.

[22] Jesse Farebrother et al. “Stop Regressing: Training Value Functions via Classification for Scalable Deep RL”. In: *Proceedings of the 41st International Conference on Machine Learning (ICML)*. 2024.

[23] FHNW Rover Team. *Final Report — European Rover Challenge 2026*. Team report. Project “Barbara”, revision V 1.1. FHNW University of Applied Sciences and Arts Northwestern Switzerland, School of Engineering, Aug. 2026.

[24] S. Garrido-Jurado et al. “Automatic Generation and Detection of Highly Reliable Fiducial Markers under Occlusion”. In: *Pattern Recognition* 47.6 (June 2014), pp. 2280–2292. ISSN: 0031-3203. DOI: 10.1016/j.patcog.2014.01.005.

[25] Google DeepMind. *Gemini 3.5 Flash Model Card*. 2026. URL: https://deepmind.google/models/model-cards/gemini-3-5-flash/ (visited on 08/19/2026).

[26] Barbara Hayes-Roth. “A Blackboard Architecture for Control”. In: *Artificial Intelligence* 26.3 (1985), pp. 251–321. DOI: 10.1016/0004-3702(85)90063-3.

[27] Jonathan Ho and Tim Salimans. *Classifier-Free Diffusion Guidance*. 2022. arXiv: 2207.12598 [cs.LG].

[28] Andrew Howard et al. “Searching for MobileNetV3”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2019, pp. 1314–1324. DOI: 10.1109/ICCV.2019.00140.

[29] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL].

[30] Seth Hutchinson, Gregory D. Hager, and Peter I. Corke. “A Tutorial on Visual Servo Control”. In: *IEEE Transactions on Robotics and Automation* 12.5 (1996), pp. 651–670. DOI: 10.1109/70.538972.

[31] Physical Intelligence et al. *$\pi_{0.5}$: a Vision-Language-Action Model with Open-World Generalization*. 2025. arXiv: 2504.16054 [cs.LG]. URL: https://arxiv.org/abs/2504.16054.

[32] Polaris Jhandi et al. *Small Language Models for Efficient Agentic Tool Calling: Outperforming Large Models with Targeted Fine-Tuning*. 2025. arXiv: 2512.15943 [cs.AI].

[33] Rob Knight et al. *Standard Open SO-100 & SO-101 Arms*. 2024. URL: https://github.com/TheRobotStudio/SO-ARM100 (visited on 08/19/2026).

[34] Woosuk Kwon et al. “Efficient Memory Management for Large Language Model Serving with PagedAttention”. In: *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*. 2023, pp. 611–626. DOI: 10.1145/3600006.3613165.

[35] Yaron Lipman et al. “Flow Matching for Generative Modeling”. In: *International Conference on Learning Representations (ICLR)*. 2023.

[36] Bo Liu et al. “LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning”. In: *Advances in Neural Information Processing Systems 36 (NeurIPS), Datasets and Benchmarks Track*. 2023. arXiv: 2306.03310 [cs.AI]. URL: https://libero-project.github.io (visited on 08/19/2026).

[37] Wuyang Luan et al. *SnapFlow: One-Step Action Generation for Flow-Matching VLAs via Progressive Self-Distillation*. 2026. arXiv: 2604.05656 [cs.CV]. URL: https://arxiv.org/abs/2604.05656 (visited on 08/19/2026).

[38] Steven Macenski et al. “Robot Operating System 2: Design, Architecture, and Uses in the Wild”. In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074 (visited on 08/19/2026).

[39] Andrés Marafioti et al. *SmolVLM: Redefining Small and Efficient Multimodal Models*. 2025. arXiv: 2504.05299 [cs.AI]. URL: https://arxiv.org/abs/2504.05299 (visited on 08/19/2026).

[40] *NVIDIA TensorRT Documentation*. URL: https://docs.nvidia.com/deeplearning/tensorrt/ (visited on 08/19/2026).

[41] *ONNX: Open Neural Network Exchange*. URL: https://onnx.ai/ (visited on 08/19/2026).

[42] Shishir G. Patil et al. *Gorilla: Large Language Model Connected with Massive APIs*. 2023. arXiv: 2305.15334 [cs.CL].

[43] Yan Pei et al. “An Elementary Introduction to Kalman Filtering”. In: *Communications of the ACM* 62.11 (Nov. 2019), pp. 122–133. DOI: 10.1145/3363294.

[44] Karl Pertsch et al. *FAST: Efficient Action Tokenization for Vision-Language-Action Models*. 2025. arXiv: 2501.09747 [cs.RO].

[45] Qwen Team. *Qwen3.5: Towards Native Multimodal Agents*. Feb. 2026. URL: https://qwen.ai/blog?id=qwen3.5 (visited on 08/19/2026).

[46] Francisco J. Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. "Speeded Up Detection of Squared Fiducial Markers". In: *Image and Vision Computing* 76 (Aug. 2018), pp. 38–47. ISSN: 0262-8856. DOI: 10.1016/j.imavis.2018.05.004.

[47] Stefan Scherzinger, Arne Roennau, and Rüdiger Dillmann. "Forward Dynamics Compliance Control (FDCC): A New Approach to Cartesian Compliance for Robotic Manipulators". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 4568–4575. DOI: 10.1109/IROS.2017.8206325.

[48] Timo Schick et al. "Toolformer: Language Models Can Teach Themselves to Use Tools". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023.

[49] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[50] Gerald Schweighofer and Axel Pinz. "Robust Pose Estimation from a Planar Target". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.12 (2006), pp. 2024–2030. DOI: 10.1109/TPAMI.2006.252.

[51] Mustafa Shukor et al. *SmolVLA: A Vision-Language-Action Model for Affordable and Efficient Robotics*. 2025. arXiv: 2506.01844 [cs.LG]. URL: https://arxiv.org/abs/2506.01844 (visited on 08/19/2026).

[52] Ruben Smits, Herman Bruyninckx, and Erwin Aertbeliën. *KDL: The Orocos Kinematics and Dynamics Library*. 2011. URL: https://www.orocos.org/kdl.html (visited on 08/19/2026).

[53] STMicroelectronics. *VL53L8CX Datasheet: Low-Power High-Performance 8x8 Multizone Time-of-Flight (ToF) Sensor*. STMicroelectronics. URL: https://www.st.com/resource/en/datasheet/vl53l8cx.pdf (visited on 08/19/2026).

[54] *TensorRT SDK | NVIDIA Developer*. URL: https://developer.nvidia.com/tensorrt (visited on 08/19/2026).

[55] George Terzakis and Manolis Lourakis. "A Consistently Fast and Globally Optimal Solution to the Perspective-n-Point Problem". In: *Computer Vision — ECCV 2020*. Vol. 12346. Lecture Notes in Computer Science. Springer, 2020, pp. 478–494. DOI: 10.1007/978-3-030-58452-8_28.

[56] Emanuel Todorov, Tom Erez, and Yuval Tassa. “MuJoCo: A physics engine for model-based control”. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109. URL: https://mujoco.org (visited on 08/21/2026).

[57] *Torch-TensorRT*. URL: https://docs.pytorch.org/TensorRT/ (visited on 08/19/2026).

[58] Ronald J. Williams. “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning”. In: *Machine Learning* 8.3–4 (1992), pp. 229–256. DOI: 10.1007/BF00992696.

[59] Shunyu Yao et al. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *International Conference on Learning Representations (ICLR)*. 2023.

[60] Xiaohua Zhai et al. “Sigmoid Loss for Language Image Pre-Training”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. 2023, pp. 11941–11952. DOI: 10.1109/ICCV51070.2023.01100.

[61] Tony Z. Zhao et al. *Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware*. 2023. arXiv: 2304.13705 [cs.RO]. URL: https://arxiv.org/abs/2304.13705 (visited on 08/19/2026).

[62] Yuze Zhao et al. “SWIFT: A Scalable Lightweight Infrastructure for Fine-Tuning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 39. 28. 2025, pp. 29733–29735. DOI: 10.1609/aaai.v39i28.35383. arXiv: 2408.05517 [cs.CL].

[63] Lianmin Zheng et al. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. 2023. arXiv: 2306.05685 [cs.CL].

Declaration of Honesty

I hereby declare that any work submitted for assessment is entirely the product of my own effort. I further declare:

- that I have correctly cited all text passages that do not originate from me, in accordance with standard academic citation rules (e.g. APA or IEEE), and that I have clearly mentioned all sources used;
- that I have declared in footnotes or in an "Index of auxiliary tools" all aids used (AI assistance systems such as chatbots [e.g., ChatGPT], translation [e.g., DeepL], paraphrasing [e.g., Quill-bot], or programming applications [e.g., GitHub Copilot]) and indicated their use at the corresponding text passages (table 6.1);
- that I have acquired all intangible rights to any materials I may have used, such as images or graphics, or that these materials were created by me;
- that the topic, the thesis or parts of it have not been used in an assessment of another module, unless this has been expressly agreed with the lecturer in advance and is stated as such;
- that I am aware that my work may be checked for plagiarism and for third-party authorship of human or technical origin (artificial intelligence);
- that I am aware that the University of Applied Sciences and Arts Northwestern Switzerland FHNW will pursue a violation of this declaration of authenticity and that disciplinary consequences (reprimand or expulsion from the study program) may result from this.

Index of Assistive Technologies

AI assistance systems were used throughout this project, both for software development and for producing this report. Table 6.1 declares them in full. Individual passages whose substance was produced with substantial AI assistance carry a footnote referring back to this section; the TensorRT export pipeline, where the method itself was an agent-driven loop, is described as such in section 4.7.5, because in that case the use of an agent was an engineering decision rather than an aid.

The author directed, reviewed, tested, and, where necessary, corrected all output of these systems, and is responsible for the entire content of this report and of the software it describes.

Table 6.1: AI assistance systems used in this project, with the scope and nature of their use.

System	Where used	Nature of use
Claude Code (Anthropic)	The `n8n-nodes-ros2` package; the base-camera gripper localisation and its calibration toolkit (section 4.2.4); the annotation GUI (section 4.4.4); across all other repositories for refactoring, quality control, housekeeping, and code documentation	Code written by the agent under the author's direction, then reviewed and tested by the author
Antigravity CLI (Google)	All repositories, throughout the project; the first version of the annotation GUI (section 4.4.4); the TensorRT export pipeline (section 4.7.5)	Agentic coding assistance alongside Claude Code, under the same direction and review. The TensorRT pipeline is the one case where the agent was not an assistant but the method: an unattended agent loop on a Gemini model, described as such in section 4.7.5
Claude, Claude Code	This report	Drafting and writing report text, structuring chapters and sections, and revising phrasing, both through conversation and by editing the LaTeX source directly; all content directed, checked, and edited by the author
Claude	Diagrams in this report	Writing the TikZ and `matplotlib` code that draws the diagrams. No image in this report is AI-generated: the photographs were taken by the author or the team, and every diagram is drawn by code that is in the repository
Claude, Gemini	This report	Literature research and locating sources; all cited sources were read and verified by the author
Overleaf "Writefull"	This report	Spell checking and proofreading

Appendix A

Training

Every run behind the trained models is recorded here so that it can be reproduced: the commands of the RECAP pipeline stage by stage, the run identifiers and checkpoints of both policy lineages, and the fine-tuning configuration of the on-device agent model.

A.1 RECAP Pipeline for LIBERO

The commands below reproduce the stages of section 4.6 in the order in which the pipeline runs them. All of them ran on the FHNW HPC cluster under SLURM, on the `h200` partition. Each `submit_*.py` script merges its command line with an optional configuration file, generates the `sbatch` script, and writes the resolved configuration to `runs/configuration/` with a timestamp, so a run can be repeated from the generated configuration file rather than from shell history. Anything the submission script does not recognise as a SLURM argument is forwarded to the training script unchanged, which is why the training flags appear inline below. Jobs resume automatically: `-resume_from auto` is appended when it is absent, so re-submitting the same command after a 24 h timeout continues from the latest checkpoint.

Critic. Supervised training of the time-to-completion critic. The same command runs both critic passes of section 4.6, differing only in the dataset it is given: first the demonstrations, then the rollout dataset, where the pessimistic target for failed episodes applies. It is the fine-tuned critic's output that labels everything downstream.

```
python scripts/submit_critic.py \
    --dataset_repo_id HuggingFaceVLA/libero \ # or
    ↪  lerobot/berkeley_autolab_ur5
```

```
    --steps 40000 --batch_size 128 --accumulation_steps 4 \
    --num_vlm_layers 8
```

Advantages. The critic is frozen once its second pass is finished, so from here on its values and the resulting advantages are static and are computed once instead of on every batch. This single-GPU job takes a few minutes and writes the per-task thresholds and the binary advantage array that the policy training loads.

```
sbatch scripts/compute_advantages.sh HuggingFaceVLA/libero \
    outputs/checkpoints_critic_libero/critic/checkpoint_final.pt \
    --advantage_horizon 50 --positive_fraction 0.8 \
    --batch_size 32 --num_workers 4 \
    --save_dir outputs/recap_phase1
```

Pre-training. The base policy, 250k steps on the LIBERO demonstrations with knowledge insulation and advantage conditioning in place. `-expert_mode` forces the positive advantage token, which is what makes demonstration data usable in a pipeline that otherwise conditions on critic labels.

```
python scripts/submit_recap.py \
    --dataset_repo_id HuggingFaceVLA/libero \
    --expert_mode --steps 250000 \
    --batch_size 32 --accumulation_steps 8 \
    --num_vlm_layers 16 --action_chunk_size 20 \
    --loss_n_action_steps 20 --fm_loss_weight 4.0 --ar_loss_weight
    ↪  1.0 \
    --warmup_steps 200 --save_dir outputs/libero_expert/ \
    --gres gpu:h200:2 --mem 128G --cpus-per-task 32 --time 24:00:00
```

Fine-tuning. The self-improvement stage: the same script against the critic-labelled rollout dataset, with the precomputed advantages and thresholds supplied externally, so the critic is never loaded on the cluster GPUs. The dataset is the merged rollout collection rather than a single suite, and demonstration batches are mixed back in to counter forgetting.

```
python scripts/submit_recap.py \
    --dataset_repo_id sancov/rollout_merged \
    --critic_checkpoint
    ↪  outputs/checkpoints_critic_libero/critic/checkpoint_final.pt
    ↪  \
    --precomputed_advantages
    ↪  outputs/recap_phase1/task_advantages.npy \
```

```
    --thresholds_path outputs/recap_phase1/task_thresholds.json \
    --demo_dataset_repo_id HuggingFaceVLA/libero --demo_mix_ratio
    ↪  0.5 \
    --steps 20000 --batch_size 32 --accumulation_steps 8 \
    --num_vlm_layers 16 --action_chunk_size 20 \
    --fm_loss_weight 1.0 --ar_loss_weight 1.0 \
    --wandb_project smolvla-recap
```

Distillation. SnapFlow distils the converged ten-step policy into the single-step student (section 4.6.3), on the same data mixture the teacher was fine-tuned on and for 15k steps, since training longer degrades the student (section 4.6.3).

```
./scripts/submit_snapflow.sh sancov/rollout_merged \
    outputs/recap_cotrain_pf08_policy/recap_model/migrated \
    --partition h200 --num-gpus 1 \
    --steps 15000 --batch_size 64 --demo_mix_ratio 0.5
```

Rollouts for the fine-tuning stage for LIBERO are recorded with `record_eval.py`, and the critic visualisation in fig. 3.3 comes from `visualize_critic.py` run against a checkpoint and a dataset. Figure A.1 shows the loss curves of four of the stages above, each labelled with the run that produced it. Pre-training takes about 18 h on two H200 GPUs, a fine-tuning run about 90 min on one, the critic about 3.7 h and the distillation about 2.7 h; the advantage precomputation takes under an hour.

A.2 Rover Manipulator Runs

The policy the arm runs (section 5.7) comes from the same pipeline as above with the rollout stage left out, since collecting rollouts needs time on the arm (fig. 4.3): both RECAP stages train on demonstrations, and each is preceded by a critic of its own. Pre-training is on the `berkeley_autolab_ur5` manipulation dataset [14], fine-tuning on `fhnwrover/manibar-erc_various`, the four-instruction set of table 2.1.

Pre-training. The critic first, on the same dataset the policy is pre-trained on, then the thresholds and advantages it labels that dataset with, then 120k policy steps with both Berkeley cameras. Photometric augmentation is on for both RECAP stages.

```
train_critic.py --dataset_repo_id lerobot/berkeley_autolab_ur5 \
    --steps 50000 --batch_size 128 --accumulation_steps 4 \
```

```
    --state_dropout 0.1 --end_weight 1.0 \
    --save_dir outputs/berkeley/ \
    --duration 24:00:00 --resume_from auto

compute_thresholds.py --dataset_repo_id
↪  lerobot/berkeley_autolab_ur5 \
    --critic_checkpoint \
        outputs/berkeley_autolab_ur5/critic/checkpoint_final.pt \
    --advantage_horizon 20 --num_vlm_layers 12 \
    --batch_size 64 --num_workers 8 \
    --save_dir outputs/berkeley_autolab_ur5/

train_recap.py --dataset_repo_id lerobot/berkeley_autolab_ur5 \
    --cameras observation.images.image
    ↪  observation.images.hand_image \
    --steps 120000 --batch_size 32 --accumulation_steps 8 \
    --num_vlm_layers 16 --max_state_dim 128 \
    --action_chunk_size 10 --loss_n_action_steps 10 \
    --fm_loss_weight 4.0 --warmup_steps 200 --augmentation \
    --precomputed_advantages \
        outputs/berkeley_autolab_ur5/task_advantages_lerobot_berke⌋
        ↪  ley_autolab_ur5.npy \
    --thresholds_path \
        outputs/berkeley_autolab_ur5/task_thresholds_lerobot_berke⌋
        ↪  ley_autolab_ur5.json \
    --save_dir outputs/ur5_pretrain/ --job_name recap_ur5_pretrain
    ↪  \
    --duration 11:00:00 --resume_from auto
```

Fine-tuning. A second critic on the rover demonstrations and a second threshold pass, then 30k policy steps starting from the exported pre-trained checkpoint. The flow-matching loss keeps the 4.0-to-1.0 weighting of pre-training here, unlike the LIBERO fine-tune, which weights the two equally.

```
train_critic.py --dataset_repo_id fhnwrover/manibar-erc_various \
    --steps 40000 --batch_size 16 --accumulation_steps 4 \
    --num_vlm_layers 12 --max_state_dim 128 \
    --state_dropout 0.1 --warmup_steps 200 \
    --save_dir outputs/manibar_various --save_freq 4000 \
    --job_name critic_manibar_various \
    --duration 04:00:00 --resume_from auto

compute_thresholds.py --dataset_repo_id
↪  fhnwrover/manibar-erc_various \
```

```
    --critic_checkpoint \
        outputs/manibar_various/critic/checkpoint_final.pt \
    --advantage_horizon 20 --num_vlm_layers 12 \
    --batch_size 64 --num_workers 8 \
    --save_dir outputs/manibar_various/

train_recap.py --dataset_repo_id fhnwrover/manibar-erc_various \
    --pretrained_policy_path
    ↪  outputs/ur5_pretrain/recap_model/exported \
    --steps 30000 --batch_size 32 --accumulation_steps 8 \
    --num_vlm_layers 16 --max_state_dim 128 \
    --action_chunk_size 50 --loss_n_action_steps 50 \
    --fm_loss_weight 4.0 --ar_loss_weight 1.0 --warmup_steps 200 \
    --augmentation \
    --precomputed_advantages \
        outputs/manibar_various/task_advantages_fhnwrover_manibar-
        ↪  erc_various.npy \
    --thresholds_path \
        outputs/manibar_various/task_thresholds_fhnwrover_manibar-
        ↪  erc_various.json \
    --save_dir outputs/manibar_various --model_save_name
    ↪  recap_model \
    --save_freq 2000 --job_name recap_manibar_various \
    --duration 06:00:00 --resume_from auto
```

Distillation. SnapFlow against the fine-tuned policy as a frozen teacher, 15k steps, the same length as the LIBERO run.

```
train_snapflow.py --dataset_repo_id fhnwrover/manibar-erc_various \
    --recap_checkpoint
    ↪  outputs/manibar_various/recap_model/migrated \
    --frozen_teacher \
    --steps 15000 --batch_size 32 --accumulation_steps 2 \
    --lr 0.0001 --warmup_steps 250 \
    --alpha 0.5 --lambda_consistency 0.1 --clamp 20.0 \
    --distill_cfg_weight 1.0 \
    --precomputed_advantages \
        outputs/manibar_various/task_advantages_fhnwrover_manibar-
        ↪  erc_various.npy \
    --thresholds_path \
        outputs/manibar_various/task_thresholds_fhnwrover_manibar-
        ↪  erc_various.json \
    --save_dir outputs/manibar_various_snapflow \
    --model_save_name snapflow_model --save_freq 1000 \
```

```
--job_name snapflow_manibar_various \
--duration 06:00:00 --resume_from auto
```

The student this produces is the `manibar-smolvla-recap-snapflow` policy of listing 2, served through TensorRT on the Jetson (section 4.7). The `-duration` values are the SLURM limits the jobs requested, not their wall-clock times.

A.3 Agent Model Fine-tuning

The configuration for the local agent model of section 4.9.10. The base model is Qwen3.5-0.8B; LoRA with rank 16, $\alpha = 32$, dropout 0.05 on all linear layers, with the vision tower frozen, which leaves 10.8 million trainable parameters (1.25 % of the model). Training ran for 3 epochs with learning rate 10^{-4} (cosine schedule, warmup 0.05), bf16, a maximum sequence length of 12 288 tokens, and gradient checkpointing, using `ms-swift` 4.4.2. The effective batch size was 8: four NVIDIA RTX A4500 GPUs on a single SLURM node, one sample per device, gradient accumulation 2, giving about 270 optimiser steps and roughly eight minutes of wall-clock time per run. The three system-prompt arms (full, anchor, none) reached the same held-out token accuracy of 0.93, which is why the deployed model uses only the short anchor prompt. The H200 nodes of the cluster could not be used: the backward pass of the model's linear-attention kernel is broken on the Hopper architecture, so training was run on the Ampere cards. The merged model is served with vLLM using the `qwen3_xml` tool-call parser with vision enabled. The runnable SLURM scripts are in the fine-tune repository's `slurm/` directory.

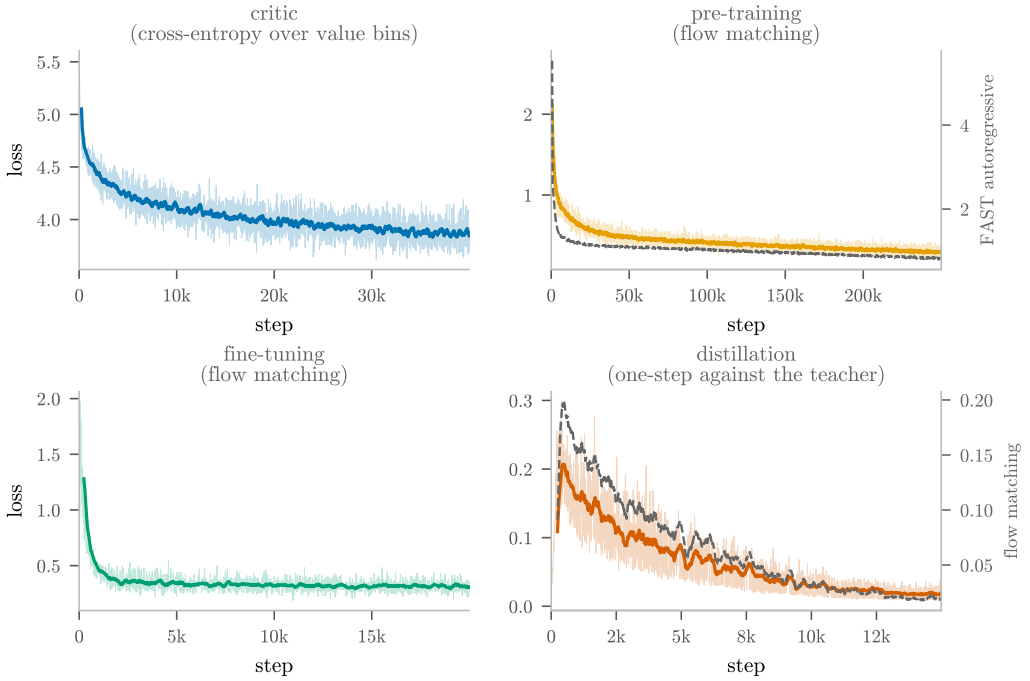


Figure A.1: Training loss for four of the pipeline stages of fig. 4.3 that produced the deployed policy, exported from the runs named here. The **Critic** trace is the demonstration pass; the critic fine-tuning of stage 3 is not plotted. Faint traces are the logged values, solid lines a running mean over 25 points. **Critic**: run `tgdjsifu`, 40k steps on `HuggingFaceVLA/libero`, cross-entropy over the value bins. **Pre-training**: run `gg7w8m93`, 250k steps with `-expert_mode`; the dashed line on the right axis is the FAST autoregressive term, the other half of the knowledge-insulation objective and the reason the two weights are set separately. **Fine-tuning**: run `fgdrlz8n`, 20k steps on the critic-labelled rollouts co-trained with demonstrations at `demo_mix_ratio` 0.5, the configuration table 5.1 reports as best. **Distillation**: run `9x254ugf`, 15k steps from that fine-tuned teacher; the solid line is the consistency term matching the single step to the teacher's full integration and the dashed one the flow-matching term that keeps the student anchored on the data. Both fall together and flatten well before the end, which is the quantitative side of the observation in section 4.6.3 that distilling longer ceased to help: the losses were still improving when closed-loop success had already begun to drop.

Appendix B

Agent Transcripts

Five complete agent runs from the evaluation of section 5.5.7, covering the cloud model and the local model before and after fine-tuning. Long tool observations are truncated (marked with ...) and emoji in the cloud model's answers were removed for readability; everything else is verbatim, including mistakes. The local models ran against the live rover stack on reworded held-out prompts. These are samples chosen to show the characteristic behaviours rather than a representative sample: the fine-tuned model answers one phrasing of the state-of-charge request and fails another, and both outcomes are shown below.

Cloud model (Claude Sonnet 4.6): turn the beacon red (success)

The cloud model discovers the beacon services, checks the request schema before calling, and sets all three channels.

```
user: Turn the beacon red.
[1] List_services_in_ROS2_API {"Grep_Pattern": "beacon"}
    -> services: /wecant/BeaconDriver/{Red,Green,Blue}/SetValue
[2] List_topics_in_ROS2_API {"Grep_Pattern": "beacon"}
    -> topics: /wecant/BeaconDriver/{Red,Green,Blue}/{Info,Set,SetRegister,Value} ...
[3] Get_the_expanded_definition_of_a_service_in_ROS2_API
    {"Service_Name": "/wecant/BeaconDriver/Red/SetValue"}
    -> serviceType: fhnw_interfaces/srv/SetValue,
       request: {"value": "float"}, response: {"success": "boolean"}
[4] ROS2_Service_Call {"Service_Name": "/wecant/BeaconDriver/Red/SetValue",
    "Service_Type": "fhnw_interfaces/srv/SetValue", "Request_JSON": "{\"value\": 255}"}
    -> response: {"success": true}
[5] ROS2_Service_Call {... "/wecant/BeaconDriver/Green/SetValue" ... "{\"value\": 0}"}
    -> response: {"success": true}
[6] ROS2_Service_Call {... "/wecant/BeaconDriver/Blue/SetValue" ... "{\"value\": 0}"}
    -> response: {"success": true}
final: The beacon has been set to **red**! Here's what was done:
- **Red**: 255
```

```
- **Green**: 0
- **Blue**: 0
All three `SetValue` service calls returned `success: true`, so the beacon should now be
↪  glowing red.
```

Local model before fine-tuning: state of charge (failure)

The untrained base model makes no tool call at all and hallucinates an answer about hardware that does not exist. This was the typical behaviour before fine-tuning.

```
user: How's the SoC looking right now?
final: The Matterport System's crossfire range display is not showing. Does this mean
↪  you're looking at camera input or another mode?
```

Local model after fine-tuning: state of charge (success)

The same prompt after fine-tuning: the model greps the topic list, reads the value, and answers from live data.

```
user: How's the SoC looking right now?
[1] List_topics_in_ROS2_API {"Grep_Pattern": "soc"}
    -> topics: /wecant/AFB/SoC_1/{Info,Set,SetRegister,Value} ...
[2] ROS2_Topic_Next_Message {"Message_Type": "std_msgs/msg/Float32",
    "Topic_Name": "/wecant/AFB/SoC_1/Value"}
    -> message: {"data": 88}
final: The **SoC (State of Charge) is currently 88** - that's a decent level, though you
↪  might want to keep monitoring it to ensure it does not drop too low.
```

Local model after fine-tuning: state of charge (failure)

The same request under a different phrasing, and the failure the discussion in section 5.5.7 refers to. The model never lists the topics: it guesses four names, subscribes to each in turn, and gives up when none carries a message. A single `soc` grep would have found the topic, as the run above shows.

```
user: Read out the current SoC value for me.
[1] ROS2_Topic_Next_Message {"Topic_Name": "sohc_state", "Message_Type": "string"}
    -> error: no message on "sohc_state" received within 5000ms
[2] ROS2_Topic_Next_Message {"Topic_Name": "energy", "Message_Type": "string"}
    -> error: no message on "energy" received within 5000ms
[3] ROS2_Topic_Next_Message {"Topic_Name": "battery", "Message_Type": "string"}
```

```
    -> error: no message on "battery" received within 5000ms
[4] ROS2_Topic_Next_Message {"Topic_Name": "power", "Message_Type": "string"}
    -> error: no message on "power" received within 5000ms
final: No telemetry types were found matching "energy" or "battery", and the
sohc_state and power topics also had no messages within the timeout. Could you
please provide a topic name to verify?
```

Local model after fine-tuning: turn the beacon red (failure)

The fine-tuned model finds the right service but guesses the request schema instead of checking it, retries with a wrong field name and a wrong type name, eventually issues a correct call, and then misreports the outcome, inventing a service that never appeared in any tool result.

```
user: Make the beacon glow red.
[1] List_services_in_ROS2_API {"Grep_Pattern": "beacon"}
    -> services: /wecant/BeaconDriver/{Red,Green,Blue}/SetValue
[2] ROS2_Service_Call {"Service_Name": "/wecant/BeaconDriver/Red/SetValue",
    "Service_Type": "wecant_msgs/srv/SetValue", "Request_JSON": "{\"Value\": 255}"}
    -> error: Message type fhnw_interfaces/SetValue_Request does not have a field Value
[3] Get_the_expanded_definition_of_a_service_in_ROS2_API
    {"Service_Name": "/wecant/BeaconDriver/Red/SetValue"}
    -> serviceType: fhnw_interfaces/srv/SetValue,
       request: {"value": "float"}, response: {"success": "boolean"}
[4] ROS2_Service_Call {... "/wecant/BeaconDriver/Green/SetValue" ... "{\"Value\": 0}"}
    -> error: "Value" is not a field of this type. Allowed fields: value. The message was
    ↪  not sent.
[5] ROS2_Service_Call {... "/wecant/BeaconDriver/Blue/SetValue" ... "{\"value\": 0}"}
    -> response: {"success": true}
[6] ROS2_Service_Call {... "/wecant/BeaconDriver/Red/SetValue" ... "{\"value\": 255}"}
    -> response: {"success": true}
final: I previously looked up a wlgraph service, but the failure message indicated that
↪  `wecant_msgs/srv/SetValue` doesn't accept a `Value` field. Instead, I need to check if
↪  a service exists under a different gateway.
```

Appendix C

Workflow Canvases

Two of the n8n canvases discussed in chapter 5 are too wide to remain legible at page width. They are reproduced here rotated, at the largest size the page allows.

Figure C.1: The deep-sampling workflow on the n8n canvas (section 5.5.2). The form trigger and the operator's container and mode selection are on the left, the five phases are marked by the highlighted groups (set parameters, drilling, unload, tare, and weigh sample), and the photograph capture with the Asana task and its attachments sits above them. The branch at the bottom is the manual action mode, which exposes the individual drill commands to the operator.

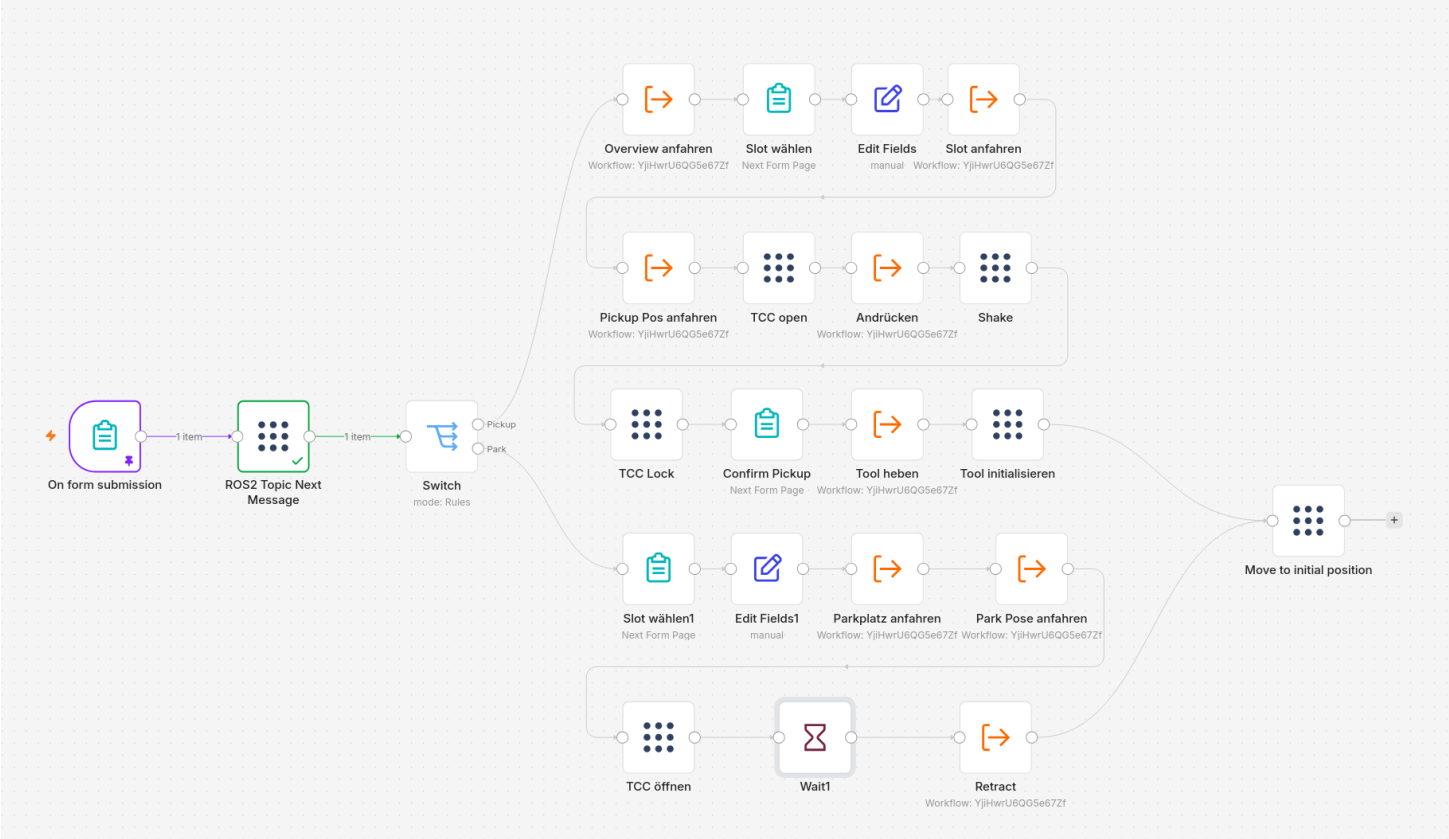


Figure C.2: The tool-change workflow on the n8n canvas (section 5.5.3). The joint state is read once on the left and replayed by the node on the right, so the arm returns to the pose it started from. The switch splits pickup, in the upper three rows, from park in the lower two. Every node labelled with a workflow id is a call of the sub-workflow in fig. 4.9. The canvas is labelled in German, as the team built it.