

Standard Bots

API Manual



Standard Bots API Manual

Standard Bots

Revised September 28, 2026

Table of Contents

1	Introduction	4
1.1	Welcome!	4
1.2	Contacting Standard Bots Support	4
1.3	Intended Use & Limitations	4
1.4	Warnings & Risks	5
1.4.1	Introduction	5
1.4.2	Installation	5
1.4.3	Operator Safety	6
1.4.4	Transportation	6
1.4.5	Cybersecurity	6
2	API & ROS2 Overview	8
2.1	When to Use the SDK and REST API vs ROS2 Bridge	8
3	API	9
3.1	Python SDK	9
3.2	Installing and Authenticating	9
3.3	Connecting to the Robot	10
3.4	How Every Call Works	10
3.5	Control Surface Reference	11
3.6	The Pattern for Any Motion	12
3.7	Use Case A – Reading Live Robot State	12
3.8	Use Case B – Moving the Arm to a Pose	13
3.9	Use Case C – Running a Saved Routine	14
3.10	Use Case D – Controlling the Gripper	15
3.11	Use Case E – Reading and Writing I/O	16
3.12	Use Case F – Recovering from a Fault or E-Stop	17
3.13	Use Case G – Using a Saved Space	17
3.14	Use Case H – Setting the Payload	17
3.15	Use Case I – Triggering a Software Emergency Stop	18
3.16	Troubleshooting	18
3.17	Known Limitations	19
3.18	REST API	19
3.18.1	Authentication and Base URL	19
3.18.2	Route Map	19
4	ROS2	21
4.1	ROS2 Setup	21
4.1.1	Docker Engine Installation and Host Configuration	21
4.1.2	IP Address Configuration	22
4.1.3	Verifying the Network Connection	23
4.1.4	Launching the Docker Container	23
4.1.5	Verifying ROS2 Topic Visibility	23
4.1.6	Executing a Motion Script	24

4.2	ROS2 Python Scripting	24
4.3	Before You Write Code	24
4.4	Topic Reference	24
4.5	The Mandatory Pattern: Enable, Work, Disable	25
4.6	Use Case A – Monitoring Live Joint Angles	26
4.7	Use Case B – Reading the Live TCP Pose	27
4.8	Use Case C – Commanding a Cartesian Move	27
4.9	Use Case D – Closing the Loop on a Moving Target	29
4.10	Use Case E – Streaming a Joint Trajectory	30
4.11	Troubleshooting	30
4.12	Known Limitations	30
4.13	Sample Scripts	30
5	External Control (Beta)	31
5.1	How External Control Relates to the ROS2 Realtime API	31
5.2	How a Session Works	32
5.3	Prerequisites	33
5.3.1	On the Robot	33
5.3.2	On Your Machine	34
5.4	Getting Started with Joint Streaming	35
5.4.1	Step 1 – Preflight: Robot ID, Idle State, and Brakes	36
5.4.2	Step 2 – Start the Session	37
5.4.3	Step 3 – Confirm the Topics Are Live	38
5.4.4	Step 4 – Command the Arm	38
5.4.5	Step 5 – Read the Arm’s State	39
5.4.6	Step 6 – Stop the Session	39
5.5	Command Topics	39
5.5.1	Streaming Joint Positions, with Optional Velocity	40
5.5.2	Streaming Tool-Point Poses, with Optional Twist	41
5.5.3	The Gripper Channel	42
5.6	State Topics	43
5.7	External Control REST Reference	44
5.7.1	Starting a Session	44
5.7.2	Tuning the Stream with servoConfig	47
5.7.3	Stopping a Session	48
5.8	Control and Safety Semantics	48
5.9	What Ends a Session and How to Recover	49
5.9.1	Controller Faults on a Pose Stream	50
5.9.2	Recovering from a Failure	51
5.9.3	Reading Why a Session Ended	52
5.10	Streaming from Python without ROS2	53
5.10.1	RobotStream at a Glance	53
5.10.2	Order of Operations with the HTTP Session	55
5.10.3	Starting and Stopping the Session from the SDK	55
5.10.4	Threading, Errors, and Liveness	56
5.10.5	Velocities and Twist	56
5.10.6	Pose Readback Versus the Streamed Pose	56
5.10.7	Trajectories and Cancel	57
5.10.8	SDK Examples	57
5.11	Direct UDP Interface	58
5.11.1	Transport	58
5.11.2	Reader and Writer Roles	58
5.11.3	Liveness and Heartbeats	59
5.11.4	Messages You Send	59
5.11.5	Messages You Receive	61
5.11.6	Control Responses and Rejection Reasons	61
5.11.7	A Minimal C++ Client	62

5.12 Troubleshooting	63
5.13 Known Limitations	65
6 Additional Resources	67
6.1 Python SDK & REST API Repo	67
6.2 ROS2 Repo	67
6.3 Robot URDF Files	67

Chapter 1

Introduction

1.1 Welcome!

Congratulations on your new Standard Bots collaborative robot.

At Standard Bots, we believe robots can elevate people's lives — but only if people can use them. We created the RO1 to be an affordable robot that anyone can use, yet is still capable of tackling the toughest, most complex challenges.

This manual is meant to include everything needed to get you up and running. But if you ever need help, we're here to assist!

1.2 Contacting Standard Bots Support

- **Email:** support@standardbots.com
- **Phone:** 1-888-9-ROBOTS
- **Address:** Standard Bots, 35 Garvies Point Rd, Glen Cove, NY 11542

1.3 Intended Use & Limitations

The Standard Bots RO1 robot and control box are intended to be used in applications where products need to be picked and placed or applications where a repetitive process can be completed with compatible tooling. The RO1 robot and controller are intended to be used within the environmental constraints outlined in this manual. The RO1 robot and controller are intended to be implemented together, the robot is not intended to be used with any other controller and the controller is not intended to be used with any other robot.

The RO1 robot provides various safety settings and measures that, when implemented correctly and evaluated with a risk assessment, allow the robot to work in an unguarded collaborative environment in close proximity to humans.

The RO1 robot is not designed to work in applications outside the bounds of this manual and any implementation violating the indented use shall be deemed misuse. These applications include but are not limited to:

- Medical applications
- Applications in an explosive environment
- Applications where ingress protection above IP54 is required
- Applications with improper safety integration or where a risk assessment has not been created and evaluated
- Applications requiring a high degree of food safety

Standard Bots expressly denies any liability or expressed or implied warranty claims arising from intentional or unintentional misuse.

1.4 Warnings & Risks

1.4.1 Introduction

This manual is not a comprehensive guide to designing and installing a robot cell. This manual does not cover the selection and integration of 3rd party components that may be required to complete a safe installation. The solution must be designed according to the standards provided by the governing body in the country where the solution will be installed. Standard Bots is not responsible for ensuring the installation of the robot meets the standards for any given installation. The integrator assumes all responsibility for ensuring the robot installation meets all applicable standards and safety guidelines.

1.4.2 Installation

- Make sure to install the robot and all electrical equipment according to requirements and specifications in this manual (see Chapter 2).
- The RO1 control box should only be connected to secure networks.
- **Risk Assessment:** The RO1 is provided as a partially completed machinery. As such, the robot should only be used as part of a cell after a comprehensive risk assessment has taken place.
 - Aside from the robot itself, such a risk assessment may consider the robot's payload, pinching hazards, any toxic or hazardous substances in the cell, and risks involved with sharp end effectors.
 - Risk assessments must cover the entire scope of the integration, including components provided by other suppliers.
 - The risk assessment will help determine any speed or force limits to set up on the robot and what safety accessories (like light curtains or area scanners) can be used. These settings are covered in Chapter 4.
 - Anti-gravity (hand-guided) mode should only be used if risk assessment approves its use.

- Refer to ISO 10218-2 and ISO/TS 15066 for comprehensive guidelines.
- Only trained, instructed, and otherwise qualified personnel may conduct cleaning, maintenance and repair of the robot.

1.4.3 Operator Safety

- Always ensure that the robot is well-maintained and in good mechanical condition before operating it. Always inspect it before use.
- Don't wear jewelry or loose clothing when working with the RO1. Tie back any long hair.
- Never open the control box cabinet door during operation. Never feed wires through the door; feed them through the rubber slot on the bottom of the box.
- Don't expose the robot to strong magnetic fields. This may damage the robot or cause malfunctions.
- Depending on other noise in the environment where the robot is used, hearing protection may be required.
- Never attempt to make any repairs, adjustments, or inspections while the robot is running.
- Operators should stay alert and focused when operating the robot. Avoid distractions and keep attention on the task at hand.

1.4.4 Transportation

WARNING – Proper Transportation Required

To prevent damage during transport, the RO1 robot **must be placed in the supplied Pelican case** with all joints positioned in the **designated “box position.”** Transporting the RO1 outside of these conditions may result in damage to the robot .

1.4.5 Cybersecurity

The Standard Bots RO1 is able to connect to both wired and wireless internet in order to facilitate remote access, troubleshooting and software updates. It is imperative to follow best cybersecurity practices, as you would with any industrial device capable of being accessed online.

1. Secure communication: Make sure the communication channels used by your robot are encrypted and secure, so that unauthorized individuals cannot access the robot's data or control its actions.
2. Strong passwords: Set strong passwords for your robot's user accounts and ensure that they are changed regularly. Use a password manager to help you create and manage secure passwords.
3. Regular updates: Keep your robot's software and firmware up-to-date, as updates often contain security patches that address vulnerabilities.
4. Physical security: Ensure that physical access to the robot and tablet is restricted, and that the robot is stored in a secure location when not in use.
5. Network security: Keep your robot on a separate network or VLAN, and restrict access to that network to only authorized users.

6. Regular testing: Conduct regular security testing to identify any vulnerabilities and ensure that your robot is secure.

By following these best practices, you can help ensure the security of your robot and protect it from potential cybersecurity threats.

Chapter 2

API & ROS2 Overview

This has been verified and is up to date for release/1.23.124

2.1 When to Use the SDK and REST API vs ROS2 Bridge

The robot exposes two complementary developer surfaces:

- **Python SDK and REST API** — request and response control over HTTPS. This is the right choice for orchestration: loading and running a routine, reading state, controlling the gripper, or moving to a named position. Latency is your network round trip, typically tens of milliseconds and up, so it is not meant for tight closed-loop control.
- **ROS2 Realtime API (the bridge)** — the robot publishes live telemetry (joint state, TCP pose, Jacobian, end-effector IMU) and accepts streamed commands (target pose, joint trajectory) as ROS2 topics. Use this when you need to continuously read state and push commands at rate: visual servoing, conveyor or target tracking, teleoperation, force-aware motion, or feeding a trajectory from an external planner.

As a rule of thumb, if you react to input every cycle and stream setpoints, use the bridge. If you issue discrete high-level commands, use the SDK.

NOTE — The ROS2 Bridge Does Not Replace the SDK

The bridge depends on the SDK. You use the Python SDK to unbrake the arm and to switch ROS2 control on and off, so the two are always used together.

Chapter 3

API

All APIs should be treated as Beta software. While software that is controlling the robot through the API is being developed the robot should be safety guarded as a mistake in the software being developed can result in unexpected motion.

3.1 Python SDK

The Python SDK lets you control and communicate with the robot using your own Python code. It gives you access to tools and commands for things like moving the robot, reading sensor data, and connecting to other systems like cameras or cloud services. This is good for users who want more flexibility, whether you're building custom applications, automating tasks, or adding new features beyond what the built-in interface can do.

You can access the Python SDK by going to [API Docs | Standard Bots](#) and selecting Python SDK.

Once there you will be brought to the SDK configure page where it will guide you through installing the SDK and how to create a connection.

The sections that follow, up to the REST API section, cover connecting, reading state, moving the arm, running saved routines, driving the gripper, and reading and writing I/O. No ROS2 is required for any of it.

3.2 Installing and Authenticating

Install the package from PyPI:

```
pip install standardbots
```

To get a token, open **Configure Developer API** under your robot's name in the robot UI, enable the API, and copy the **Authorization Token**. The token is specific to that robot.

Your robot URL is one of the following:

- `https://cb1234.sb.app` — the cloud address for your controller, with your own serial number in place of `cb1234`.
- `http://<robot-ip>:3000` — direct on the LAN.
- `http://localhost:3000` — when running on the controller itself.

3.3 Connecting to the Robot

```
from standardbots import StandardBotsRobot, models

ROBOT_URL = "https://cb1234.sb.app"    # or http://<robot-ip>:3000
ROBOT_TOKEN = "your-api-token"        # from Configure Developer API

sdk = StandardBotsRobot(
    url=ROBOT_URL,
    token=ROBOT_TOKEN,
    # .Simulated hits the built-in sim; .Live drives the real arm
    robot_kind=StandardBotsRobot.RobotKind.Simulated,
)

# The context manager opens and cleans up the connection for you.
with sdk.connection():
    health = sdk.status.health.get_health().ok()
    print(health.health)    # 'ok' / 'warning' / 'error'
```

WARNING – Validate in Simulation First

The `robot_kind` argument decides whether your commands hit the simulator or the real arm. Develop and test against `RobotKind.Simulated`, and check the flag before switching to `RobotKind.Live`. Every robot runs a built-in simulator, so there is no reason to debug new motion code on live hardware.

3.4 How Every Call Works

Every endpoint returns a `Response` object that wraps the data, the HTTP status, and the raw `urllib3` response. You can inspect it directly, or unwrap it with `ok()`, which asserts that the status is exactly 200 and returns the payload, raising otherwise:

```
# Inspect the raw response
response = sdk.movement.brakes.get_brakes_state()
if response.status == 200:
    print(response.data)
else:
    raise Exception("Bad response!")
```

```
# Or, more simply, unwrap with ok()
brakes = sdk.movement.brakes.get_brakes_state().ok()
```

Request and response bodies are built from `models`. Two of them come up constantly:

```
position = models.Position(
    # Meters / Millimeters / Centimeters / Inches / Feet
    unit_kind=models.LinearUnitKind.Meters,
    x=0.4, y=0.0, z=0.3,
)
orientation = models.Orientation(
    kind=models.OrientationKindEnum.Quaternion,
    quaternion=models.Quaternion(x=0.0, y=1.0, z=0.0, w=0.0),
)
```

Scalar numeric model fields are type-checked at runtime and must be floats. Passing an integer such as `z=0` raises `TypeError`. See the gotchas below.

3.5 Control Surface Reference

Everything hangs off the `sdk` object. The groups you will use most:

SDK group	What it does	Key calls
<code>sdk.status</code>	Health and control mode	<code>health.get_health()</code> , <code>control.get_configuration_state_control()</code>
<code>sdk.movement.brakes</code>	Release and engage brakes, E-stop	<code>unbrake()</code> , <code>brake()</code> , <code>get_brakes_state()</code>
<code>sdk.movement.position</code>	Move the arm and read its position	<code>move_tooltip()</code> , <code>move()</code> , <code>get_arm_position()</code>
<code>sdk.poses.pose_retrieval</code>	Read TCP, joint, and flange pose	<code>get_tooltip_position()</code> , <code>get_joints_position()</code> , <code>get_flange_position()</code>
<code>sdk.routine_editor.routines</code>	Run saved routines	<code>list()</code> , <code>load()</code> , <code>play()</code> , <code>pause()</code> , <code>stop()</code> , <code>get_state()</code>
<code>sdk.routine_editor.variables</code>	Read and write routine variables	<code>load()</code> , <code>update()</code>
<code>sdk.equipment</code>	Gripper and end-effector	<code>get_gripper_configuration()</code> , <code>onrobot_2fg7_grip()</code> , <code>control_gripper()</code>
<code>sdk.io</code>	Digital and analog I/O	<code>status.get_io_state()</code> , <code>control.update_io_state()</code>
<code>sdk.recovery</code>	Inspect and clear faults	<code>recover.get_status()</code> , <code>recover.recover()</code>

SDK group	What it does	Key calls
sdk.payload	Set and read payload mass	set_payload(), get_payload()
sdk.camera	Wrist camera frames	bot.get_connected_cameras(), data.get_camera_frame()

These map one-to-one onto the REST routes described in the REST API section below.

3.6 The Pattern for Any Motion

Direct arm-motion calls require the arm to be unbraked and the robot to accept API control. The skeleton is take control, unbrake, move, then release:

```
with sdk.connection():
    # (a) If the robot is in routine-editor control, hand control to the API.
    sdk.status.control.set_configuration_control_state(
        models.RobotControlMode(kind=models.RobotControlModeEnum.Api)
    ).ok()

    # (b) Release the brakes.
    sdk.movement.brakes.unbrake().ok()

    # (c) ... do your motion (see the use cases below) ...

    # (d) Optionally re-engage the brakes when you are done.
    sdk.movement.brakes.brake().ok()
```

Running a saved routine is itself a control action and does not require you to manage the brakes, because the routine handles its own motion. Steps (a) and (b) are only for direct `move_tooltip` and `move` commands.

3.7 Use Case A – Reading Live Robot State

This answers where the arm is and whether it is healthy. It is read-only, so no unbrake step is needed.

```
with sdk.connection():
    # TCP pose: position (x, y, z) + orientation quaternion (i, j, k, w)
    pose = sdk.poses.pose_retrieval.get_tooltip_position().ok().pose
    print("xyz :", pose.x, pose.y, pose.z)
    print("quat:", pose.i, pose.j, pose.k, pose.w)
```

```

# Per-joint angles j0 through j5
joints = sdk.poses.pose_retrieval.get_joints_position().ok().pose
print("joints:", joints.j0, joints.j1, joints.j2,
      joints.j3, joints.j4, joints.j5)

# Health rollup
print("health:", sdk.status.health.get_health().ok().health)

```

NOTE – Two Different Quaternion Field Names

The TCP pose you read back stores orientation in *i*, *j*, *k*, *w*, while the `models.Quaternion` you pass into a move uses *x*, *y*, *z*, *w*. The mapping is *i* to *x*, *j* to *y*, *k* to *z*, and *w* to *w*. Use case B shows the conversion.

`get_arm_position()` is a one-call alternative that returns joint angles and the TCP pose together:

```

with sdk.connection():
    combined = sdk.movement.position.get_arm_position().ok()
    print("joints:", combined.joint_rotations)    # (j0 ... j5)
    tcp = combined.tooltip_position                # PositionAndOrientation
    print("tcp xyz:", tcp.position.x, tcp.position.y, tcp.position.z)

```

3.8 Use Case B – Moving the Arm to a Pose

The safest pattern is to read the current pose and then command a small offset from it, so that you never have to guess an absolute orientation.

```

with sdk.connection():
    sdk.movement.brakes.unbrake().ok()

    # Current TCP pose
    pose = sdk.poses.pose_retrieval.get_tooltip_position().ok().pose

    # Move 5 cm straight up, keeping the same orientation
    sdk.movement.position.move_tooltip(
        position=models.Position(
            unit_kind=models.LinearUnitKind.Meters,
            x=pose.x, y=pose.y, z=pose.z + 0.05,
        ),
        orientation=models.Orientation(
            kind=models.OrientationKindEnum.Quaternion,
            quaternion=models.Quaternion(
                x=pose.i, y=pose.j, z=pose.k, w=pose.w,
            ),

```

```

    ),
    # .Line = straight-line TCP path; .Joint = fastest joint-space path
    movement_kind=models.MovementKindEnum.Line,
).ok()

```

Things to notice in this sample:

- **move_tooltip moves the tool center point** to the target. `move` is a thin alias that calls `move_tooltip` with the same arguments, so both perform a tooltip move. Choose the frame the target is expressed in with `axis_alignment` or `reference_frame`, for example `axis_alignment="tooltip"` versus `"base"`, rather than by picking one method over the other.
- `movement_kind` is either `Line`, a straight TCP path, or `Joint`, the fastest joint-space path, which is curved in Cartesian space.
- **Cap the speed** by passing a speed profile, for example `speed_profile=models.SpeedProfile(max_tooltip_speed=Start slow when validating new targets.`
- **Use small, deliberate offsets.** A 5 cm step is gentle. Larger jumps move the arm more aggressively.

To command the six joints directly instead of a Cartesian target, for example to reach a known configuration, use `set_arm_position` with a joint rotation request:

```

with sdk.connection():
    sdk.movement.brakes.unbrake().ok()

    sdk.movement.position.set_arm_position(models.ArmPositionUpdateRequest(
        kind=models.ArmPositionUpdateRequestKindEnum.JointRotation,
        joint_rotation=models.ArmJointRotations(
            joints=(0.0, -1.2, 1.2, 0.0, 1.57, 0.0),
        ),
        movement_kind=models.MovementKindEnum.Joint,
    )).ok()

```

Joints are a six-tuple of `j0` through `j5` in the robot's joint-angle convention, in radians. Check your joint limits in the API docs before sending absolute targets. `set_arm_position` also accepts `kind=...TooltipPosition`, which is exactly what `move_tooltip` builds for you.

3.9 Use Case C – Running a Saved Routine

This triggers a routine you built on the pendant or web app, optionally overriding its variables, and then watches it run.

```

with sdk.connection():
    # Find the routine you want
    routines = sdk.routine_editor.routines.list(limit=100, offset=0).ok()
    for r in routines.items:

```

```

    print(r.id, r.name)

routine_id = "routine_xxxxxxxxxxx"

# Start it. Always pass a PlayRoutineRequest body; variables optional.
sdk.routine_editor.routines.play(
    body=models.PlayRoutineRequest(variables={"cycle_count": "5"}),
    routine_id=routine_id,
).ok()

# Monitor progress
state = sdk.routine_editor.routines.get_state(routine_id=routine_id).ok()
print("paused:", state.is_paused, "step:", state.current_step_id,
      "cycles:", state.cycle_count)

# Pause and stop
sdk.routine_editor.routines.pause(routine_id=routine_id).ok()
sdk.routine_editor.routines.stop().ok()

```

WARNING – Always Pass a Body to play()

Always pass `body=models.PlayRoutineRequest(...)` to `play()`. Older SDK versions behaved badly when it was omitted, including running live instead of simulated. Passing the body explicitly and keeping the SDK current avoids this.

You can also read and set routine variables out of band with `sdk.routine_editor.variables.load(variable_name=...)` and `.update(body=models.RuntimeVariable(value="..."), variable_name=...)`.

3.10 Use Case D – Controlling the Gripper

Confirm what is attached first, then drive it. The OnRobot 2FG7 has convenience helpers:

```

with sdk.connection():
    cfg = sdk.equipment.get_gripper_configuration().ok()
    print("gripper:", cfg.kind)    # e.g. GripperKindEnum.Onrobot2Fg7

    # Position move to a 50 mm opening
    sdk.equipment.onrobot_2fg7_move(
        value=50, unit_kind=models.LinearUnitKind.Millimeters,
    ).ok()

    # Force grip inward to 20 mm at 40 N
    sdk.equipment.onrobot_2fg7_grip(

```

```

        value=20,
        direction=models.LinearGripDirectionEnum.Internal,
        force=40, force_unit=models.ForceUnitKind.Newtons,
    ).ok()

```

The generic call works for any supported gripper, covering the OnRobot 2FG7, 2FG14, 3FG15, and Screw-driver, the Schunk EGx, and the Robotiq 2F and EPick:

```

sdk.equipment.control_gripper(models.GripperCommandRequest(
    kind=models.GripperKindEnum.Onrobot2Fg7,
    onrobot_2fg7=models.OnRobot2FG7GripperCommandRequest(
        grip_direction=models.LinearGripDirectionEnum.Internal,    # required
        control_kind=models.OnRobot2FG7ControlKindEnum.Move,
        target_grip_width=models.LinearUnit(
            unit_kind=models.LinearUnitKind.Millimeters,
            value=50.0,    # must be a float, not 50
        ),
    ),
)).ok()

```

If a force grip fails, confirm that a gripper is actually configured and powered. `get_gripper_configuration().kind` should not be `NoneConnected`. Force-close errors usually trace back to an unconfigured or unpowered end-effector rather than to your code.

3.11 Use Case E – Reading and Writing I/O

Read first. The response shows you the exact pin keys and value strings your robot uses, so you never have to guess them.

```

with sdk.connection():
    io = sdk.io.status.get_io_state().ok()
    # e.g. {"tool_digital_out_0": "high", "base_digital_in_2": "low"}
    print(io.state)

    # Write: send back the same shape with only the pins you want changed
    sdk.io.control.update_io_state(
        models.IOStateUpdateRequest(state={"tool_digital_out_0": "high"})
    ).ok()

```

If I/O writes do not seem to take effect, confirm that the controller software is current. On older builds, an I/O step inside a routine is a workaround.

3.12 Use Case F – Recovering from a Fault or E-Stop

```
with sdk.connection():
    state = sdk.recovery.recover.get_status().ok()
    # RobotStatusEnum: Idle / RoutineRunning / Failure / Recovering / ...
    print("status:", state.status)
    print("failed:", state.failed)
    if state.failure:
        print("kind          :", state.failure.kind)
        print("reason        :", state.failure.reason)
        print("recovery_type:", state.failure.recovery_type)

    # Clear a software-recoverable fault
    if state.failed:
        sdk.recovery.recover.recover().ok()
```

WARNING – Not Every Fault Clears from Software

The `recovery_type` field tells you what is needed, which may be a guided-mode or manual recovery, a restart, or a physical wrist-button press. A physical E-stop must be released by hand before `recover()` will succeed.

3.13 Use Case G – Using a Saved Space

Spaces are the named positions you save in the app. This drives the arm to a position you taught on the pendant.

```
with sdk.connection():
    spaces = sdk.space.list_global_spaces().ok()
    for s in spaces.items:
        print(s.id, s.name)    # find the Space you want
```

A Space carries its saved coordinates in `s.positions`, where each entry has joint angles, a Cartesian pose, or both. To move there, feed those values into a `move_tooltip` or `set_arm_position` call as shown in use case B, or run a routine that already references the Space. Routine-scoped, non-global Spaces are available through `sdk.routine_editor.routines.list_spaces(routine_id=..., exclude_global_spaces=...)`.

3.14 Use Case H – Setting the Payload

Tell the robot the mass it is carrying so that motion planning stays accurate. Set it after a pick and clear it after a place.

```

with sdk.connection():
    sdk.payload.set_payload(models.PayloadStateRequest(mass=2.0)).ok()    # kg
    print("payload:", sdk.payload.get_payload().ok().mass)

```

3.15 Use Case I – Triggering a Software Emergency Stop

You can command an E-stop from code, for example when a watchdog trips, and then recover afterward using the flow in use case F. This is distinct from the physical button, which must still be released by hand.

```

with sdk.connection():
    sdk.movement.brakes.engage_emergency_stop(
        models.EngageEmergencyStopRequest(reason="watchdog tripped")
    ).ok()

```

3.16 Troubleshooting

- **401 or unauthorized.** The token is bad or missing. Re-copy it from **Configure Developer API**, and remember that it is per-robot.
- **Timeouts or connection refused against a real robot.** Confirm that the Developer API is enabled, that the host is reachable from your machine, and that you are using the right URL form. Intermittent connection refused after a few good calls has been seen on older controller software, so update the controller if you hit it.
- **It moved the real robot when you expected the simulator.** `robot_kind` controls this. Develop on `Simulated`, verify the flag before going `Live`, and keep the SDK current.
- **The arm will not move, or commands are rejected.** The brakes are still engaged, so call `unbrake()`; or there is an active fault, so see use case F; or the robot is in routine-editor control or already running a routine. Check `sdk.status.control.get_configuration_state_control()` and switch to `Api`.
- **Gripper errors on a force grip.** Confirm that a gripper is configured and powered, and use the helper matching your model.
- **Do not guess I/O pin names.** Read `get_io_state()` first to discover the exact keys and values.
- **Pin the SDK version.** The API surface changes between releases, so pin `standardbots==<version>` and re-test on upgrade.
- **Numbers must be floats.** Scalar numeric model fields such as `Position.x`, `.y`, and `.z`, `Quaternion`, `LinearUnit.value`, `ForceUnit.value`, `SpeedProfile.max_tooltip_speed`, and `PayloadStateRequest.mass` are type-checked at runtime. Passing an integer such as `z=0` or `value=50` raises `TypeError: ... must be of type float`. Write `0.0` and `50.0` instead. Tuple fields such as `ArmJointRotations.joints` do accept integers.

3.17 Known Limitations

- **Routines cannot be created, uploaded, or deleted through the public API.** You can list, load, play, pause, and stop existing routines. Routine authoring happens on the pendant or web app.
- **There is no direct velocity control over REST.** For continuous velocity, force-aware, or closed-loop streaming, use the ROS2 Realtime API described later in this chapter.
- **REST is request and response.** Round-trip latency makes it unsuitable for tight closed-loop control. It is for orchestration and discrete commands.
- **Camera access is limited for some depth features.** For full depth streams, some customers use the camera vendor's SDK directly alongside the Standard Bots API.

3.18 REST API

With REST API you will be able to control the movements of your robot. Leverage reference frames, tooltips, and more to accomplish your task.

You can access the REST API by going to [API Docs | Standard Bots](#) and selecting [Move Your Robot](#)

You will then be brought to the [Arm Movement and Position Control Page](#). Here it will walk you through moving the robot via the API.

The Python SDK is a typed wrapper over these same REST routes, so anything the SDK does is also available over plain HTTP. Use the REST API directly when you are working in a language other than Python, calling the robot from a PLC or an existing HTTP client, or debugging with a tool such as `curl`.

3.18.1 Authentication and Base URL

Authentication uses the same per-robot Authorization Token described in the [Python SDK section](#), copied from **Configure Developer API** in the robot UI. The base URL forms are also the same: `https://cb1234.sb.app` for the cloud address, `http://<robot-ip>:3000` on the LAN, or `http://localhost:3000` when running on the controller itself. Routes are served under `/api/v1/`.

3.18.2 Route Map

Each SDK group corresponds directly to a REST route, so the SDK is a reliable guide to the HTTP surface:

SDK group	REST route
<code>sdk.movement.position</code>	<code>/api/v1/movement/position</code>
<code>sdk.movement.brakes</code>	<code>/api/v1/movement/brakes</code>
<code>sdk.poses.pose_retrieval</code>	<code>/api/v1/poses</code>
<code>sdk.routine_editor.routines</code>	<code>/api/v1/routine-editor/routines</code>
<code>sdk.status.control</code>	<code>/api/v1/status/control</code>
<code>sdk.equipment</code>	<code>/api/v1/equipment</code>

SDK group	REST route
<code>sdk.io</code>	<code>/api/v1/io</code>
<code>sdk.recovery</code>	<code>/api/v1/recovery</code>

The ROS2 bridge control state used later in this chapter is also a REST route: GET and POST `/api/v1/movement/ros/state`.

For the exact request and response bodies, query parameters, and the full route list, use the API reference at docs.standardbots.com. The units, enum values, float-versus-integer rules, and quaternion field names documented earlier in this chapter apply to the REST payloads as well.

Chapter 4

ROS2

With ROS you will be able to use fine grained controls and force feedback. At this time for ROS2, the level of control possible is Joint Trajectory Controller and Tooltip Pose in beta. ROS Nodes are not available at this time through the Standard Bots UI. ROS2 is not required for basic applications. The cobot can run predefined programs without ROS 2. ROS 2 is only needed for custom applications, integration with external systems, or advanced programming. Please reach out to learn more as this is in Private Beta.

4.1 ROS2 Setup

This procedure establishes deterministic, low-latency ROS2 communication with the robot. The runtime stack routes real-time commands across a direct Ethernet connection encapsulated within a Docker container. No prior Docker or ROS2 experience is required to complete this deployment.

Before you begin, make sure you have:

- A host computer running Ubuntu 22.04 or 24.04. This procedure has been validated on 24.04. Ubuntu's current release is newer, but 24.04 can still be downloaded from releases.ubuntu.com/noble.
- An Ethernet cable connected directly between the host computer and the control box.
- The bot ID, token, and URL for your robot, available from the API panel on the tablet.

4.1.1 Docker Engine Installation and Host Configuration

Update the system package index and install the foundational dependencies:

```
sudo apt-get update
sudo apt-get install -y ca-certificates curl gnupg
```

Install Docker via the terminal:

```
sudo apt update
sudo apt install git docker.io docker-buildx -y
```

Grant the current user permission to manage Docker without elevated privileges:

```
sudo usermod -aG docker $USER
newgrp docker
```

Clone the Standard Bots ROS2 repository:

```
mkdir ~/robot_projects
cd ~/robot_projects
git clone https://github.com/standardbots/ros2-realtime-api.git
cd ros2-realtime-api
```

With the Ethernet cable connected between the host computer and the control box, identify the name of the active Ethernet interface:

```
ip a
```

The relevant interface name will typically begin with the letter `e` (for example, `eth0`, `enx...`, or `ether1...`). If the correct interface cannot be identified, disconnect the Ethernet cable and re-run the command to determine which interface is no longer present.

Once the interface name has been determined, open `cyclonedds.xml` in the `ros2-realtime-api` folder and update the value within the `<NetworkInterface>` element to reflect that interface name:

```
nano cyclonedds.xml
```

Press **Ctrl + X** to save and exit.

Build the Docker image using the Dockerfile provided in the repository with the build file inside the `ros2-realtime-api` folder:

```
./build.sh
```

4.1.2 IP Address Configuration

On the Ubuntu host, open System Settings and navigate to the wired connection settings. Set the IPv4 method to Manual and assign the following values:

- Address: 192.168.50.101
- Subnet Mask: 255.255.255.0
- Gateway: leave blank

On the robot tablet, open the Network Settings and switch the wired adapter from Automatic (DHCP) to Manual. Configure the following values:

- IP Address: 192.168.50.100
- Subnet Mask: 255.255.255.0
- Gateway: 192.168.50.1

Make sure you press the connect button after entering the values to save the IP addresses.

WARNING – Distinct IP Addresses Required

The host computer must not be assigned the same IP address as the robot (192.168.50.100). Both devices must occupy distinct addresses on the same subnet.

Save the changes, then navigate to the Developer API screen and ensure that **Enable ROS2 Bridge** is toggled ON.

4.1.3 Verifying the Network Connection

Disable the Ubuntu firewall to allow ROS communication to reach the host computer from the robot:

```
sudo ufw disable
```

Issue a ping request from the host to confirm the link is operational:

```
ping 192.168.50.100
```

A successful response will display sequential 64 bytes from 192.168.50.100... lines. Press **Ctrl + C** to terminate the request.

If the request stalls or drops packets and the control box has recently been power-cycled, disconnect the Ethernet cable, wait approximately two minutes for the system to fully initialize, and then reconnect the cable.

4.1.4 Launching the Docker Container

Launch the container using the helper script provided in the repository:

```
./run_shell.sh
```

4.1.5 Verifying ROS2 Topic Visibility

Once inside the container (the prompt will appear as `root@...:/app#`), point the session to the CycloneDDS configuration file:

```
export CYCLONEDDS_URI=file:///etc/standardbots/configuration/cyclonedds.xml
```

List the active topics:

```
ros2 topic list
```

Topics formatted as `/bot_0sapi_.../...` indicate a successful connection to the robot.

If only topics of the form `/None/ro1/hardware/...` are visible, the container can resolve the local middleware but cannot reach the robot. Common causes include:

- The `CYCLONEDDS_URI` environment variable was not exported in the current session.
- The interface name in `cyclonedds.xml` does not match the output of `ip a`.
- A firewall is blocking communication.

4.1.6 Executing a Motion Script

Before executing any script that initiates motion of the arm, ensure that the script running through the ROS2 bridge unbrakes the robot using the Python SDK before sending any movement messages.

WARNING – Absolute Joint Targets

Motion scripts use absolute joint targets rather than relative offsets. All joints will move toward their target positions simultaneously upon execution. Ensure the workspace surrounding the arm is fully clear before launching the script.

Retrieve the bot ID, token, and URL from the API panel on the tablet, then execute:

```
python3 ./src/write_poses.py \
  --bot-id="bot_XXXXXXXXXXXXXXXXXXXX" \
  --token="xxxx-xxxxx-XXXXXXXX-XXXXXXXX" \
  --url="https://cbXXX.sb.app"
```

The arm will trace a smooth trajectory to the target pose.

4.2 ROS2 Python Scripting

The sections that follow cover writing Python scripts against the RO1 ROS2 topics. Complete the ROS2 Setup section above before you start, because they assume the Developer API and the ROS2 bridge are already enabled and that your client environment is running. You will also need the robot URL and API token from the Configure Developer API screen for any script that moves the arm.

4.3 Before You Write Code

- **You enable and disable over the SDK, not over ROS2.** Unbraking and switching ROS2 control on and off go through the SDK over REST. The realtime data itself flows over ROS2 topics. A typical script opens a short SDK connection to change the control state, then does all of the realtime work purely over ROS2 publish and subscribe.
- **Match the publisher's QoS.** The hardware topics publish BEST_EFFORT. A subscriber left on the default RELIABLE profile silently receives zero messages. This is the single most common failure.
- **Never hardcode the bot ID.** Topics are namespaced /<BOT_ID>/ro1/hardware/... Call `detect_bot_id()` to discover it at runtime.

4.4 Topic Reference

All topics are namespaced by your bot ID: /<BOT_ID>/ro1/hardware/...

Topic	Message type	Direction	Use
.../joint_state	sensor_msgs/JointState	Robot to you	Live joint angles and velocities
.../pose	geometry_msgs/PoseStamped	Robot to you	Live TCP pose
.../pose/write	geometry_msgs/PoseStamped	You to robot	Command a target TCP pose
.../joint_trajectory	trajectory_msgs/JointTrajectory	You to robot	Stream a joint trajectory to execute
.../jacobian	std_msgs/Float64MultiArray	Robot to you	Jacobian, for advanced Cartesian control
.../end_effector_imu	sensor_msgs/Imu	Robot to you	Wrist IMU

To see what is actually live, list the topics from a ROS2 shell with `ros2 topic list`.

4.5 The Mandatory Pattern: Enable, Work, Disable

Every motion script follows the same skeleton. Two ideas do all of the work: flip the control state with the SDK over HTTPS, then stream over ROS2. On exit, flip control back off so the arm does not stay live.

Connect the SDK and enable ROS2 control

```
from standardbots import StandardBotsRobot, models

sdk = StandardBotsRobot(
    url=ROBOT_URL,          # from Configure Developer API
    token=ROBOT_TOKEN,
    robot_kind=StandardBotsRobot.RobotKind.Live,  # .Simulated for a sim
)

with sdk.connection():
    sdk.movement.brakes.unbrake().ok()          # release the brakes
    sdk.ros.control.update_ros_control_state(    # hand control to ROS2
        models.ROSControlUpdateRequest(
            action=models.ROSControlStateEnum.Enabled,
        )
    )
    sdk.ros.status.get_ros_control_state().ok()  # confirm it took
```

The SDK connection only needs to be open to change the control state. The realtime data itself flows

over ROS2 topics, so it is normal to close this with block and then do all of your publish and subscribe work outside of it. This is exactly what `write_poses.py` does.

Always disable on exit

Wrap the realtime loop in try and finally, and switch ROS2 control back to Disabled:

```
finally:
    rclpy.shutdown()
    with sdk.connection():
        sdk.ros.control.update_ros_control_state(
            models.ROSControlUpdateRequest(
                action=models.ROSControlStateEnum.Disabled,
            )
        )
```

Both calls hit the same REST endpoint that the SDK wraps: GET and POST `/api/v1/movement/ros/state`.

Match the publisher's QoS

Do not skip this step. The hardware topics publish `BEST_EFFORT`, and a subscriber on the default `RELIABLE` profile will silently receive zero messages:

```
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy

qos = QoSProfile(
    reliability=ReliabilityPolicy.BEST_EFFORT,
    history=HistoryPolicy.KEEP_LAST,
    depth=10,
)
```

4.6 Use Case A – Monitoring Live Joint Angles

Use this when you want to log or visualize what the arm is doing in real time, for telemetry, dashboards, or data capture. This is read-only, so no unbrake or enable step is needed. You only need to subscribe.

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from sensor_msgs.msg import JointState
from detect_bot_id import detect_bot_id

class ReadJointState(Node):
    def __init__(self, robot_id=None):
```

```

super().__init__("read_joint_state")
self.robot_id = robot_id or detect_bot_id(self)
qos = QoSProfile(reliability=ReliabilityPolicy.BEST_EFFORT,
                 history=HistoryPolicy.KEEP_LAST, depth=10)
self.create_subscription(
    JointState,
    f"/{self.robot_id}/ro1/hardware/joint_state",
    self.cb, qos)

def cb(self, msg):
    print([v for v in msg.position])    # swap in your own processing

rclpy.init()
rclpy.spin(ReadJointState())

```

4.7 Use Case B – Reading the Live TCP Pose

This has the same shape as use case A, but subscribes to `.../pose` with `geometry_msgs/PoseStamped`. The message gives you the tool center point position (`msg.pose.position.x`, `.y`, and `.z`) and the orientation quaternion (`msg.pose.orientation.w`, `.x`, `.y`, and `.z`). This is the signal you close the loop on in use cases C and D.

4.8 Use Case C – Commanding a Cartesian Move

Use this to nudge the tool along an axis from wherever it currently is. This is the canonical sample, `write_poses.py`, and the template for any pose-streaming task.

The strategy is to subscribe to the current pose to learn the start pose, then publish a series of slightly shifted target poses to `.../pose/write`.

```

class WritePose(Node):
    def __init__(self, robot_id=None):
        super().__init__("write_pose")
        self.robot_id = robot_id or detect_bot_id()

        qos = QoSProfile(reliability=ReliabilityPolicy.BEST_EFFORT,
                        history=HistoryPolicy.KEEP_LAST, depth=10)

        # READ the current pose
        self.create_subscription(

```

```

PoseStamped, f"/{self.robot_id}/ro1/hardware/pose",
self.pose_callback, qos)

# WRITE target poses
self.publisher = self.create_publisher(
    PoseStamped, f"/{self.robot_id}/ro1/hardware/pose/write", 10)

self.start_pose = None

def pose_callback(self, msg):
    if self.start_pose is None:      # latch the first pose we see
        self.start_pose = msg.pose

def start(self):
    while not self.start_pose:      # wait until we know where we are
        rclpy.spin_once(self, timeout_sec=0.5)

    for i in range(100):
        p = PoseStamped()
        p.header.frame_id = "world"
        # copy the start pose...
        p.pose.position.x = self.start_pose.position.x
        p.pose.position.y = self.start_pose.position.y
        p.pose.orientation = self.start_pose.orientation
        # ...then shift 1 mm along Z each step, 10 cm total
        p.pose.position.z = self.start_pose.position.z + i * 0.001
        self.publisher.publish(p)
        rclpy.spin_once(self, timeout_sec=0.2)    # paces the loop

```

Things to notice in this sample:

- **Latch the start pose.** You cannot command a delta until you know the current pose, so the first callback stores it and the loop builds every target relative to it.
- `frame_id = "world"`. Targets are expressed in the world frame.
- **Small increments.** One millimeter per step keeps motion smooth and safe. A bigger step or a faster loop makes the arm move more aggressively, so change those deliberately. Flip the sign to `- i * 0.001` to reverse direction.
- `spin_once(timeout_sec=0.2)` paces the publish loop and lets callbacks run. Tighten it for a higher command rate.

Because this script moves the arm, it takes the connection arguments it needs in order to unbrake and enable: `--bot-id=<BOT_ID> --token=<TOKEN> --url=<ROBOT_URL>`.

4.9 Use Case D – Closing the Loop on a Moving Target

Use this to follow a part on a conveyor, a detected object, or an external setpoint in real time, as in visual servoing, conveyor tracking, and fast-target following.

Use case C streams a path that was decided in advance. Real applications instead compute each new target from live input. The shape is identical, and only the line that decides the next pose changes:

```
def control_step(self):
    if self.latest_pose is None:
        return
    target = PoseStamped()
    target.header.frame_id = "world"

    # 1) get your external signal (camera detection, conveyor
    #    encoder, network setpoint, and so on) - your code:
    dx, dy, dz = self.get_target_offset()

    # 2) command current pose + correction
    target.pose.position.x = self.latest_pose.position.x + dx
    target.pose.position.y = self.latest_pose.position.y + dy
    target.pose.position.z = self.latest_pose.position.z + dz
    target.pose.orientation = self.latest_pose.orientation

    self.publisher.publish(target)

# drive control_step on a fixed-rate timer, not a for loop:
self.create_timer(0.02, self.control_step)  # 50 Hz example
```

Practical notes for tracking loops:

- Drive publishing from a `create_timer` at a fixed rate rather than a `for` loop. It is steadier, and it decouples your control rate from message arrival.
- Keep per-step corrections bounded by clamping `dx`, `dy`, and `dz`. Large jumps from a noisy detector translate directly into aggressive arm motion.
- Achievable tracking bandwidth depends on the network and on how fast your perception or setpoint source runs. Measure end-to-end latency before committing to a tracking rate for your application.

4.10 Use Case E – Streaming a Joint Trajectory

For planner-driven motion, publish `trajectory_msgs/JointTrajectory` to `.../joint_trajectory` with one or more `JointTrajectoryPoint` entries, each carrying positions plus `time_from_start`. Read `.../jacobian` and `.../end_effector_imu` when you need Cartesian-space or inertial awareness in your controller. The same enable and disable handshake and the same QoS rules described above still apply.

4.11 Troubleshooting

- **A subscriber receives nothing.** This is almost always a QoS mismatch. The hardware publishes `BEST_EFFORT`, and a default `RELIABLE` subscriber gets silence. Use the `BEST_EFFORT` profile shown above.
- **No topics are discovered, or `detect_bot_id` hangs.** Confirm that the ROS2 bridge is enabled on the robot and that your client environment is set up, as described in the ROS2 Setup section. List what is live with `ros2 topic list`.
- **`NameError: name 'os' is not defined`.** `write_poses.py` reads `ROBOT_URL` and `ROBOT_TOKEN` through `os.getenv` but never imports `os`. Either pass `--url` and `--token` on the command line, or add `import os` if you want the environment variable fallback.
- **Messages publish but the arm will not move.** The arm is still braked, or ROS2 control is not Enabled. Confirm that the SDK `unbrake()` and `Enabled` calls succeeded, and that there are no active errors on the Move page.
- **The arm stays live after the script ends.** The `finally` block that sets control to `Disabled` was skipped. Always disable on exit.
- **Motion is too fast or jerky.** Reduce the per-step delta, reduce the loop rate, and clamp corrections in closed-loop code.

4.12 Known Limitations

- Realtime guarantees are bounded by your network and by the bridge. Benchmark latency for any rate-sensitive use case, such as tracking, before committing to specific numbers.
- The bridge controls arm motion. Gripper control, IO, and high-level routines still go through the SDK and REST API.

4.13 Sample Scripts

The `standardbots/ros2-realtime-api` repository cloned during setup contains the samples referenced above:

- `src/write_poses.py` — the Cartesian pose-streaming sample from use case C.
- `src/read_joint_states.py` — the joint state subscriber from use case A.
- `src/detect_bot_id.py` — runtime bot ID discovery.

Chapter 5

External Control (Beta)

The new External Control software is in beta. Reach out to support@standardbots.com for information about joining.

External control lets an outside program drive the arm directly. You open an external-control **session** over HTTP, stream setpoints to the arm for as long as that session lasts, and read the arm's live state back the whole time.

There are three ways to speak it. All three open the same kind of session and are governed by the same safety rules:

- **ROS2 topics** — publish to a small set of topics and subscribe to another set. Works with ROS2 Humble or Jazzy, and uses only standard message types.
- **Python SDK** — the realtime `RobotStream` client. No ROS2 required.
- **Direct UDP** — protobuf over UDP, from C++ or any other language.

The ROS2 bridge and the Python SDK are both thin adapters over the UDP protocol, so everything in the Control and Safety Semantics section applies whichever one you choose. The setpoints are the same on all three, with one exception: only the Python SDK and direct UDP can attach an optional twist to a pose setpoint, because `geometry_msgs/PoseStamped`, the ROS2 pose message, has no velocity field.

WARNING — Your Code Moves the Arm

External control streams motion commands straight from your program to the joints, with no routine, planner, or operator in between. Guard the cell and keep people clear while you are developing against it. External control is Beta and access is granted case by case, so contact support@standardbots.com to have it enabled.

5.1 How External Control Relates to the ROS2 Realtime API

The ROS2 Setup and ROS2 Python Scripting sections earlier in this chapter describe the ROS2 Realtime API: topics under `/<BOT_ID>/ro1/hardware/...`, switched on and off with `sdk.ros.control.update_ros_control_state`

which wraps GET and POST `/api/v1/movement/ros/state`.

External control is a newer, session-based interface. Its topics live under `/<robotId>/external/...`, it is switched on and off with POST `/api/v1/external-control/start` and POST `/api/v1/external-control/stop`, and it adds two non-ROS transports alongside the ROS2 one.

Which interface your robot exposes depends on the software it is running. On a robot running the external-control stack, the older `movement/ros/state` endpoint returns 500 with `ControlSourceManager` is enabled. That response is the signal to use `external-control/start` instead.

The topics listed in the Command Topics and State Topics sections below are the complete `/external/*` set. The Jacobian and wrist IMU topics described in the Topic Reference section belong to the `/ro1/hardware/...` tree, not to this one.

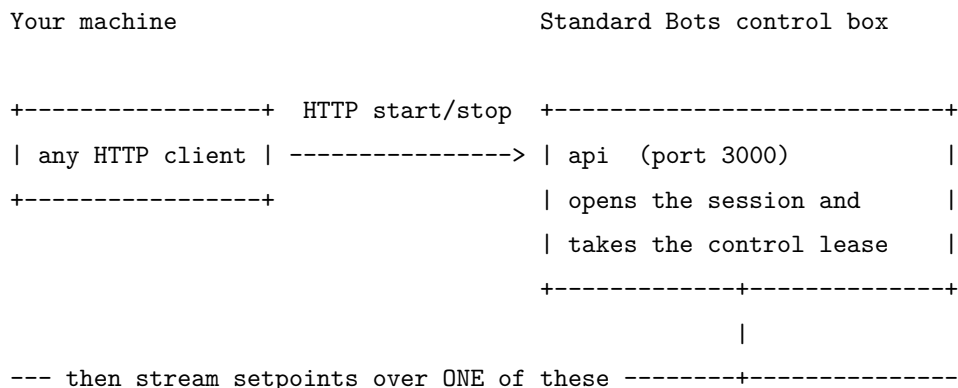
5.2 How a Session Works

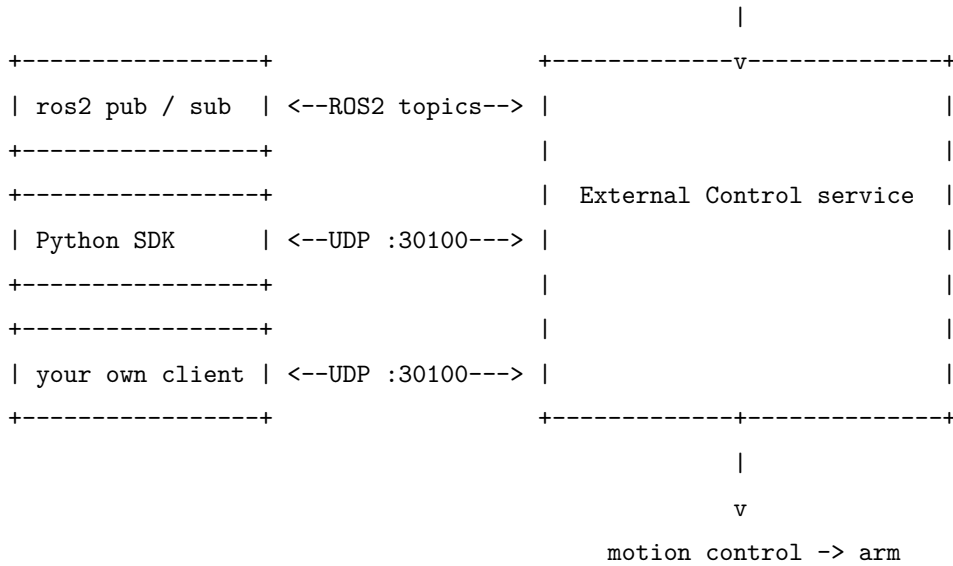
1. POST `/api/v1/external-control/start` with the stream type you want, for example `joints` for live joint streaming. The robot takes control and, for a ROS2 session, brings up the ROS2 bridge.
2. Publish setpoints to `/<robotId>/external/<modality>` and read state from `/<robotId>/external/robot_*`.
3. POST `/api/v1/external-control/stop` when you are done. The robot releases control and stops safely.

Control is held for the lifetime of the session. There is **no per-message engage handshake**, and control is held until you call `stop` or the session ends on its own. If you stop publishing, what happens depends on the stream type. On a robot with the *Streaming controller* capability, described in the Prerequisites section, a `joints` or `pose` session decelerates the arm to rest after the command timeout, 500 ms by default, and the session ends. A `jointTrajectory` session holds position.

A session that ends on its own leaves the robot in a failure state, not idle, so you have to recover the robot before you can `start` again. The What Ends a Session and How to Recover section covers every case.

Every session has the same shape no matter how you drive it. One HTTP call opens the session and takes the control lease, you stream setpoints over one of three interchangeable transports, and a second HTTP call ends it.





These are alternatives, not layers you stack, so pick one per session. All three reach the same external control service, and the ROS2 and Python SDK front ends are thin clients over the same protobuf-over-UDP protocol that the third one speaks directly.

Transport	Reach for it when
ROS2 topics	You already have a ROS2 stack to publish from. See the Command Topics and State Topics sections.
Python SDK	You want live control from Python, without ROS2. See the Streaming from Python without ROS2 section.
Direct UDP	You are working in C++ or another language. See the Direct UDP Interface section.

The ROS2 front end is brought up per session by a `start` with `controlledBy` set to `rosHumble` or `rosJazzy`, and it is torn down by `stop`. The UDP listener serves the other two transports. Either way, only one external client can hold the writer slot at a time.

5.3 Prerequisites

5.3.1 On the Robot

External control depends on robot-side capabilities that Standard Bots configures, so contact support@standardbots.com if your robot does not have them turned on. The first three are required for any session. The last two vary from robot to robot and change how the same API behaves. `start` succeeds either way, so confirm with Standard Bots which of them your robot has before you build on them. Their italicized names are used throughout the External Control sections that follow.

Capability	What it enables
Control arbitration	Lets an external client hold the control lease that permits commanding the arm.
External-control sessions	Enables the session that <code>start</code> and <code>stop</code> create.
ROS2 bridge	Publishes the <code>/external/*</code> topics and relays them to the arm. Only needed for ROS2.
<i>Streaming controller</i>	Routes <code>joints</code> and <code>pose</code> streams through the tracked streaming controller. <code>servoConfig</code> tuning, the command watchdog, and the <code>velocity</code> and <code>twist</code> fields work only with it. Without it, they are accepted and ignored.
<i>Trajectory execution</i>	Executes multi-point trajectories: the <code>.../trajectory</code> and <code>.../cancel</code> topics, the UDP <code>trajectory_command</code> and <code>cancel_trajectory</code> messages, and the SDK's <code>send_trajectory()</code> and <code>send_cancel()</code> . Where it is off, they are accepted and ignored.

NOTE – Streaming Controller or Trajectory Execution, Not Both

A robot has either *Streaming controller* or *Trajectory execution*, never both. `start` succeeds either way, so ask Standard Bots which one your robot has.

The bridge supports ROS2 Humble and Jazzy. The `controlledBy` field of `start`, set to `rosHumble` or `rosJazzy`, selects which one a session launches, so match it to the ROS2 distribution on your machine.

5.3.2 On Your Machine

Whichever transport you drive the arm with, you need:

- An **API token** for the robot's HTTP API. Every `start` and `stop` call takes one. This is the same per-robot Authorization Token used everywhere else in this chapter, copied from **Configure Developer API** in the robot UI.
- **Network reachability** to the robot's control box on port **3000**.

If you are using ROS2, you also need:

- **ROS2 Humble or Jazzy** with the standard message packages `geometry_msgs`, `sensor_msgs`, `trajectory_msgs`, and `std_msgs`. All of them come with a normal ROS2 install.
- **CycloneDDS** as the RMW implementation, matching the robot:

```
export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
```

- **The robot's external ROS domain.** The bridge publishes the `/external/*` topics on the robot's *external* DDS domain, which defaults to 1. Your robot may be configured differently. Match it:

```
export ROS_DOMAIN_ID=1
```

- **A route to the robot's wired customer Ethernet port, which has to be up with an IP address.** The external DDS domain is bound to that port only. The robot's Wi-Fi is not on it unless an operator adds it in the robot's ROS2 network settings. The bridge binds to the wired port too, so if the port has no address, a ROS2 session ends within a couple of seconds of start.
- **DDS discovery** on that subnet. Multicast is used for participant discovery only, and unicast for everything else, so your machine has to be able to reach the robot's wired IP address directly.

WARNING – Domain, RMW, or Network Mismatch Is the Most Common Failure

A mismatch is the single most common reason topics never appear. If `ros2 topic list` shows nothing after start, check `ROS_DOMAIN_ID`, `RMW_IMPLEMENTATION`, and that you are on the robot's wired subnet. If `/external_control_ros_bridge` *does* appear in `ros2 node list` but no `/external/*` topics do, your machine can reach only one of the addresses the robot advertises. Move to the wired subnet, or have the operator restrict the robot's ROS2 interfaces to the one you can reach.

If you are using the Python SDK, you also need:

- **The `standardbots` package**, installed with `pip install standardbots`. It contains both the REST client and the realtime client, and it needs **Python 3.9 or later**, or 3.10 or later for the `servo_overshoot.py` example. The realtime client is the `standardbots.realtime` subpackage:

```
from standardbots import StandardBotsRobot, models
from standardbots.realtime import (
    RobotStream, JointState, CartesianPose, CartesianTwist,
    Point, Quaternion, Vector3,
)
```

- **A package version that carries the API** described in the Streaming from Python without ROS2 section. Versions are numbered `2.<YYYYMMDD>.<n>`, after the robot release they ship with. `twist`, `velocities`, and per-joint limits need the release that carries them, so ask Standard Bots which version that is and pin it.

The realtime client speaks UDP directly, so none of the ROS2 or DDS setup above applies.

If you are writing your own client, you also need `protoc` and the `external_control.proto` contract, and UDP port **30100** on the control box has to be reachable. The contract is append-only, so fields that a robot does not know about are simply absent. Again, no ROS2 or DDS is involved.

5.4 Getting Started with Joint Streaming

This walkthrough uses `streamType: "joints"`, which is live joint streaming. Pose streaming and `jointTrajectory` are covered in the Command Topics section.

5.4.1 Step 1 – Preflight: Robot ID, Idle State, and Brakes

The robot has to be **idle**, meaning no routine is running and it is not in **Failure**, and its **brakes disengaged** before you start a session. You can do this from the robot UI, with the Python SDK, or entirely over the same HTTP API. Every call below takes the same `Authorization: Bearer <token>` header as `start`.

First, get the `robotId`. It namespaces every external-control ROS2 topic in this section, and it is what `<robotId>` stands for throughout:

```
curl http://<robot-ip>:3000/api/v1/identity/bot_identity \
  -H "Authorization: Bearer <token>"
```

The response identifies the robot:

```
{
  "robotId": "...",
  "alias": "...",
  "robotOperationMode": "...",
  "applicationUrl": "..."
}
```

If `ros2 topic list` instead shows topics under `/local/external/*`, the robot has no identity configured, so contact Standard Bots.

Next, check that the robot is idle:

```
curl http://<robot-ip>:3000/api/v1/recovery/status \
  -H "Authorization: Bearer <token>"
# => { "status": "Idle", "failed": false }
```

`status` is the state that start tests. `Idle` means the robot is ready, and anything else is refused with 409 `robot_not_idle`. `RoutineRunning` means a routine or another session is running, so stop it first. `Failure` means a previous session ended on a fault, so recover the robot, as described in the [Recovering from a Failure](#) section, and then start.

Then check the brakes, and disengage them if needed:

```
curl http://<robot-ip>:3000/api/v1/movement/brakes \
  -H "Authorization: Bearer <token>"
# => { "state": "engaged" } or { "state": "disengaged" }
```

```
curl -X POST http://<robot-ip>:3000/api/v1/movement/brakes \
  -H "Authorization: Bearer <token>" \
  -H "Content-Type: application/json" \
  -d '{"state": "disengaged"}'
```

From Python, `sdk.movement.brakes.get_brakes_state()` and `sdk.movement.brakes.unbrake()` do the same thing.

Re-check the brakes after an E-stop or a power loss. GET /api/v1/status/health and GET /api/v1/status/control-mode are the general health and mode checks if you need to diagnose before starting.

5.4.2 Step 2 – Start the Session

```
curl -X POST http://<robot-ip>:3000/api/v1/external-control/start \
  -H "Authorization: Bearer <token>" \
  -H "Content-Type: application/json" \
  -d '{"controlledBy": "rosJazzy", "streamType": "joints"}'
```

The response carries the identifier of the session that was created:

```
{ "routineId": "abc123..." }
```

The request body takes these fields:

- `controlledBy` is required. It is `rosHumble` or `rosJazzy`, matching the ROS2 distribution on your machine. Use `api` instead for the Python SDK or for your own UDP client.
- `streamType` is required. It is `pose`, `joints`, or `jointTrajectory`, and it selects which command topic is live for the session, as described in the Command Topics section.
- `servoConfig` is optional tuning for joints and pose sessions, and it is ignored on `jointTrajectory`. Every field inside it is optional, so send only what you want to change:

```
curl -X POST http://<robot-ip>:3000/api/v1/external-control/start \
  -H "Authorization: Bearer <token>" \
  -H "Content-Type: application/json" \
  -d '{
    "controlledBy": "rosJazzy",
    "streamType": "joints",
    "servoConfig": {
      "tauMs": 20,
      "commandTimeoutMs": 500,
      "rampInDurationMs": 300,
      "positionSmoothingTimeConstantMs": 20
    }
  }'
```

Full semantics, defaults, and ranges are in the Tuning the Stream with `servoConfig` section.

WARNING — `start` Returns Before the Session Is Live

`start` returns before the control lease is taken and, for a ROS2 session, before the bridge is up, which takes a few seconds. Setpoints sent before then are dropped, never queued. On ROS2, wait until the /external/* topics appear and .../robot_joints is flowing, as shown in Step 3. On UDP or the Python SDK, poll GET /api/v1/recovery/status a second after start. `RoutineRunning` means the session is up, and `Failure` means it died on the way up, so recover the robot and then start again.

5.4.3 Step 3 – Confirm the Topics Are Live

With `ROS_DOMAIN_ID` and `RMW_IMPLEMENTATION` set to match the robot:

```
ros2 topic list | grep external
# /<robotId>/external/trajectory
# /<robotId>/external/pose
# /<robotId>/external/joints
# /<robotId>/external/gripper
# /<robotId>/external/cancel
# /<robotId>/external/robot_joints
# /<robotId>/external/robot_pose
# /<robotId>/external/robot_gripper
# /<robotId>/external/status
```

`<robotId>` is your robot's identity, which is also the topic namespace. If you do not know it, `ros2 topic list` reveals it.

The bridge node is `/external_control_ros_bridge`. If the node is listed but no `/external/*` topics are, see the mismatch warning in the On Your Machine section. If the node never appears and the session is gone within seconds of `start`, the robot could not launch the bridge, as described in the Troubleshooting section.

5.4.4 Step 4 – Command the Arm

Stream joint setpoints as positions in radians, with the joints ordered `joint0` through `joint5`. Note `--rate` rather than `--once`: joint streaming is a *continuous* stream of setpoints, and the arm tracks the latest one.

```
ros2 topic pub --rate 100 \
  /<robotId>/external/joints sensor_msgs/msg/JointState "{
  name: ['joint0','joint1','joint2','joint3','joint4','joint5'],
  position: [0.0, -1.57, 1.57, 0.0, 1.57, 0.0]
}"
```

Leave this command running, and run Steps 5 and 6 in a second terminal. Killing the publisher stalls the stream, and the session ends after `commandTimeoutMs`, 500 ms by default. The robot is then in `Failure`, so `stop` in Step 6 returns `stopped: false`, and you have to recover the robot, as described in the Recovering from a Failure section, before you can start again.

Joint order is positional, and `name` is ignored, as described in the Command Topics section. The arm eases from wherever it is to this target over `rampInDurationMs`, 300 ms by default, and then holds it for as long as the publisher keeps running. To actually move the arm, republish with changing `position` values at a steady rate. `velocity` is optional and is covered in the Command Topics section.

WARNING – Keep the Deltas Between Setpoints Small

Sending a single setpoint far from the arm's current position is safe here only because the ramp-in smooths the approach. Once you are streaming for real, keep the delta between consecutive setpoints small.

5.4.5 Step 5 – Read the Arm's State

```
# live joint state
ros2 topic echo /<robotId>/external/robot_joints

# engaged=true/false, for the bridge's writer slot only
ros2 topic echo /<robotId>/external/status
```

.../status says only whether the bridge holds the writer slot. It prints nothing on a healthy session, and it does not change on a protective stop or an E-stop, as described in the State Topics section. To find out why a session ended, use `GET /api/v1/recovery/status`, as described in the Reading Why a Session Ended section.

5.4.6 Step 6 – Stop the Session

From the second terminal, call `stop` while the publisher from Step 4 is still running:

```
curl -X POST http://<robot-ip>:3000/api/v1/external-control/stop \
-H "Authorization: Bearer <token>"
```

The response confirms the teardown:

```
{ "stopped": true }
```

`stop` returns after teardown, once the arm has settled, so an immediate `start` is safe. Once `stop` has returned, you can stop the publisher. `stopped` is `false` when there was nothing to stop: the robot was idle, an operator's routine is running, which `stop` never ends, or the session already ended on its own and the robot is in `Failure`. In that last case, recover the robot and then `start`.

5.5 Command Topics

These are the topics you publish to. All names sit under the namespace `/<robotId>/external`. Pick the `streamType` at start that matches the topic you will publish, as the warning below explains.

Topic	Message type	Fields used
.../trajectory	trajectory_msgs/JointTrajectory	joint_names, points[].positions in radians, points[].time_from_start
.../joints	sensor_msgs/JointState	position in radians, optional velocity in rad/s

Topic	Message type	Fields used
.../pose	geometry_msgs/PoseStamped	pose.position in meters, pose.orientation as a quaternion
.../gripper	std_msgs/Float64MultiArray	data = [position] or [position, velocity]
.../cancel	std_msgs/Empty	none

- .../trajectory takes a multi-point path, which the robot executes with the timing you supply. joint_names, using joint0 through joint5, and points are both required, and every point needs one position per name, or the message is dropped. Only positions and time_from_start are used. This topic needs *Trajectory execution*. On other robots it is accepted and ignored: nothing moves, and nothing appears on .../status.
- .../joints takes a single joint-space setpoint. Stream these continuously. position is required and velocity is optional, while effort, name, and header are ignored. See the Streaming Joint Positions, with Optional Velocity section below.
- .../pose takes a Cartesian setpoint. Only pose is read, header is ignored, and no TF transform is applied. See the Streaming Tool-Point Poses, with Optional Twist section below.
- .../gripper is normalized to the range 0 to 1, from 0 closed to 1 open, and it works in any session. See The Gripper Channel section below.
- .../cancel aborts the in-flight .../trajectory. It is a no-op otherwise, or on a robot without *Trajectory execution*. It uses depth-1 volatile QoS, so a cancel published before the bridge is up is never replayed.

Joint order is positional: index 0 is joint0 and index 5 is joint5, with positions in radians. .../joints ignores name and does not check the count. Only .../trajectory reads joint_names.

WARNING – One Motion Modality per Session

Use one motion modality per session, matching streamType. The control lease is granted for the modality you pass to start, and publishing to another motion topic is rejected by the robot’s control arbitration. If you start with streamType: jointTrajectory but publish to .../pose, those pose commands are rejected. To switch modality, stop the session and start a new one with the new streamType. Gripper and cancel are not gated this way and work in any session.

5.5.1 Streaming Joint Positions, with Optional Velocity

For live joint control, use streamType: joints and publish sensor_msgs/JointState messages carrying position in radians, ordered joint0 through joint5, streamed continuously at a steady rate of roughly 100 to 250 Hz.

With *Streaming controller*, your targets are low-pass filtered, the arm ramps into the first one rather than jumping to it, each arrival is blended over a smoothing window, and a stalled stream is watchdogged. All of these behaviors are tunable per session through servoConfig. Without the capability, positions are applied as they arrive, with no tuning and no watchdog.

velocity is optional. With *Streaming controller*, the arm advances the target at your velocity between setpoints, a first-order hold, which removes the stepped motion you get when streaming below the control rate. Omit the field and each setpoint is held until the next one arrives. Without the capability, velocity is ignored and the positions still apply.

- **Send one velocity per position, or none at all.** A count mismatch, or a NaN or an infinity, drops the whole setpoint silently. Sustained, that stalls the stream, and the session ends. All-zero velocities are legal.
- **This velocity is not the same field as a trajectory point's.** On `.../trajectory`, a point's velocities and accelerations are dropped before the message leaves the bridge, because the wire format carries only positions and timing, so populating them is inert rather than dangerous. Shape a trajectory through its positions and `time_from_start` instead.
- **Do not emulate live control with a single-point trajectory.** Publishing a `jointTrajectory` of one point with `time_from_start = 0` asks the controller to arrive instantly, which produces a velocity spike. Use `streamType: joints` for live setpoints, and reserve `jointTrajectory` for real multi-point paths.
- **Keep the delta between consecutive setpoints small.** A stalled stream decelerates to rest after the session's command timeout, 500 ms by default and tunable per session with `servoConfig.commandTimeoutMs`.
- **Tune the smoothing to your stream rate.** `tauMs`, 20 ms by default, sets how far the arm trails your stream. Raise it if a jittery stream produces visible chatter, and lower it if tracking feels laggy. `positionSmoothingTimeConstantMs`, also 20 ms by default, spreads each setpoint's arrival correction over a window instead of a single control cycle. Set it to about twice your stream period. At 0, each setpoint is applied at once, so setpoints that arrive early or late become velocity spikes.

5.5.2 Streaming Tool-Point Poses, with Optional Twist

For Cartesian control, use `streamType: pose` and stream **tool center point** setpoints continuously. This section assumes *Streaming controller*. Without it, a pose is an unvalidated flange target in the base frame, held indefinitely with no watchdog, and `twist` is ignored.

Frame and point. Poses are expressed in `servoConfig.referenceFrame`, which is either `base`, the default, or `world`, which also applies the robot's configured base placement. The point driven is the robot's Active Tooltip, or the flange if no tool is configured. The tool offset is read at `start`, so set the tooltip before starting.

WARNING – The Pose Readback Is the Flange

`.../robot_pose` and the UDP `ServerMessage.pose` report the **flange**, as described in the State Topics section. With a tool configured, do not seed your stream from the readback plus offsets, because your first target would be off by the tool length.

Rate and hold. Pose streaming is designed for rates from 10 Hz to 1 kHz. Between setpoints, the target advances along the setpoint's **twist**, a first-order hold. With no twist the target stands still, so a

slow stream produces a staircase. Supply a twist, or stream at 100 Hz or faster. A gap longer than `commandTimeoutMs` ends the session.

Twist. `linear` is the tool point's velocity in m/s, and `angular` is the tool frame's angular velocity in rad/s, both expressed in the `referenceFrame` axes. The twist has to be the derivative of the poses you stream, or the motion becomes a sawtooth. With a tool, keep in mind that the flange's velocity is not the tool point's velocity while the tool rotates. Twist is available on UDP and the Python SDK only, because the ROS2 `geometry_msgs/PoseStamped` message has no velocity field.

Ramp-in and smoothing. The arm ramps from its current tool pose to your first setpoint over `rampInDurationMs`, 300 ms by default and clocked from the first setpoint. It then tracks the stream through a first-order low-pass filter with the time constant `tauMs`, 20 ms by default. The ramp is linear in time, so covering 0.3 m in 300 ms is an approach of roughly 1 m/s, which is enough to trip the joint acceleration check. Size the ramp to the distance, or start the stream at the arm's current tool pose. With `rampInDurationMs: 0`, a far first setpoint ends the session at once with `TRACKING_ERROR_EXCEEDED`.

- **Send orientation as a unit quaternion.** Within $1e-3$ of unit norm, it is normalized. Further off than that, or unset, which reads as all zeros, the setpoint is dropped. A twist containing a NaN or an infinity drops the setpoint too.
- **Dropped setpoints are silent.** If every setpoint is dropped from the start, the watchdog never arms: there is no motion, no fault, and the session stays open. If drops begin later, the stream stalls and the session ends. Check `orientation` before suspecting the network.
- **Keep consecutive setpoints close.** A step beyond the jump bound ends the session with `STREAM_JUMP`. The ramp-in absorbs only the first setpoint.
- **`motionScalingLimits` makes the arm lag rather than jump.** Once the lag exceeds the tracking bound, the session ends with `TRACKING_ERROR_EXCEEDED`.
- **Stopping is Cartesian.** A stall or a `stop` decelerates the tool point along its current direction, with orientation held. Stop publishing once you call `stop`, because a large jump during the stop escalates it to a joint-space stop.

The bounds behind these fault codes are listed in the Controller Faults on a Pose Stream section.

WARNING – A Non-Zero Twist Keeps the Target Moving

A non-zero twist keeps the target moving after your last setpoint, through the `commandTimeoutMs` window and the stop that follows. Send a zero twist when you intend to hold.

5.5.3 The Gripper Channel

`.../gripper`, and its UDP counterpart `gripper_command`, is a jog-shaped channel, not a pick command. It is gated on the writer slot, not on `streamType`.

Field	Range	Meaning
position	0 to 1	0 is fully closed and 1 is fully open, on the gripper's configured stroke. The same scale applies to Robotiq 2F, OnRobot 2FG and 3FG, and Schunk EGx grippers.
velocity	0 to 1	Fraction of the gripper's speed range. 0, or omitting the value, uses the gripper's default.

Force is the gripper's configured default, and a setpoint never waits for grip detection. Setpoints coalesce, so you can stream at any rate.

NOTE – No Gripper, No Channel

With no enabled, connected gripper, or with a Robotiq gripper that is still activating, commands are dropped silently and `.../robot_gripper` never publishes, so confirm that it produces data first. `.../robot_gripper` updates at 30 Hz during a session and 10 Hz otherwise.

5.6 State Topics

These are the topics the robot publishes and you subscribe to. They are republished once per robot state frame: about **200 Hz** on a real arm, up to about 400 Hz where pose and joint updates arrive as separate frames, and **40 Hz** in the simulator. They use the default reliable, volatile QoS with a depth of 10, and nothing is latched. `header.stamp` is the bridge's receive time, not a robot timestamp.

Topic	Message type	Contents
<code>.../robot_joints</code>	<code>sensor_msgs/JointState</code>	Live name from <code>joint0</code> to <code>joint5</code> , position in radians, velocity in rad/s, and effort in newton-meters, which is a filtered torque estimate.
<code>.../robot_pose</code>	<code>geometry_msgs/PoseStamped</code>	Live flange pose in the robot base frame, with position in meters and orientation as a quaternion. No tool offset and no base-placement transform are applied, regardless of <code>servoConfig.referenceFrame</code> . <code>header.frame_id</code> is the fixed label "world", but treat the data as base-frame.

Topic	Message type	Contents
.../robot_gripper	std_msgs/Float64MultiArray	data = [position], normalized to the range 0 to 1. Republished with every frame, and the value refreshes at 30 Hz during a session. Published only once the robot has reported a gripper. With none, the topic exists but never publishes.
.../status	std_msgs/String	engaged=<true/false> reason=<reason>, telling you whether the bridge holds the writer slot. Published on transitions only , and not latched. reason is auto-engaged when the bridge takes control, or send_<op>-failed when a command could not be delivered and the bridge dropped control. The bridge re-takes control on your next command.

What the .../status topic does and does not tell you:

- On a healthy session it prints **nothing**. The one engaged=true message goes out as the bridge comes up, before any subscriber has matched, so judge liveness by .../robot_joints flowing instead.
- It does **not** change on a protective stop, an E-stop, a collision, or a fault, and nothing is published at teardown. The /external/* topics simply disappear. Read the cause from GET /api/v1/recovery/status, as described in the Reading Why a Session Ended section.
- If another client holds the writer slot at start-up, the bridge stays up with every topic present, drops your commands while retrying on each one, and publishes nothing until it succeeds. See the Troubleshooting section.

5.7 External Control REST Reference

The base URL is the robot's control box on port **3000**, the same base URL used by the rest of the API. Both endpoints require an API token in the Authorization: Bearer <token> header.

5.7.1 Starting a Session

POST /api/v1/external-control/start starts an external-control session. It takes the control lease and, for rosHumble and rosJazzy, brings up the ROS2 bridge. It waits up to 2 seconds for the arm to be still, and then returns once the session routine is running. If the routine is not running within 15 seconds, the call fails with 500 cannot_play_routine. The call returns **before** the lease is held and the bridge is up, as

described in Step 2 of the Getting Started with Joint Streaming section.

A request body that sets every field looks like this:

```
{
  "controlledBy": "rosJazzy",
  "streamType": "joints",
  "servoConfig": {
    "tauMs": 20,
    "commandTimeoutMs": 500,
    "rampInDurationMs": 300,
    "positionSmoothingTimeConstantMs": 20,
    "referenceFrame": "base",
    "motionScalingLimits": {
      "jointVelocityMax": [1.0, 1.0, 1.0, 2.0, 2.0, 2.0],
      "jointAccelerationMax": [4.0, 4.0, 4.0, 4.0, 4.0, 4.0]
    }
  }
}
```

The request body has three top-level fields:

Field	Required	Values
controlledBy	yes	api, rosHumble, Or rosJazzy
streamType	yes	pose, joints, Or jointTrajectory
servoConfig	no	Streaming tuning. Honored on joints and pose, and ignored on jointTrajectory. Every field inside it is individually optional, so omit what you do not want to change. See the Tuning the Stream with servoConfig section.

A missing or unknown value for controlledBy or streamType is a 400 request_failed_validation that names the field. The two required fields are checked before servoConfig.

A successful call returns 200 with the session identifier:

```
{ "routineId": "string" }
```

Every error has the same body shape:

```
{ "error": "<code>", "message": "<detail>" }
```

These are the errors you are most likely to see:

Status	Meaning	Fix
400 <code>brakes_must_be_disengaged</code>	The brakes are engaged. This is checked first, so an E-stopped robot reports it.	Disengage the brakes, as shown in Step 1, then retry.
400 <code>request_failed_validation</code>	<code>controlledBy</code> or <code>streamType</code> is missing or not an allowed value, or a <code>servoConfig</code> field is out of range, the wrong type or shape, or null. The message names every offending field.	Correct the named field.
409 <code>robot_not_idle</code>	The message says which case applies: a routine or another session is running (<code>state: RoutineRunning</code>), a previous session ended on its own and the robot is in <code>Failure</code> , the arm has not come to rest within 2 seconds, or another session is already starting.	If something is running, stop it. On <code>Failure</code> , recover the robot and then start. Otherwise, wait a moment and retry.
500 <code>cannot_play_routine</code>	The session routine could not start, because the robot was already in <code>Failure</code> when play began, or the routine ended before it ran. Nothing is left running.	Check <code>GET /api/v1/recovery/status</code> , recover the robot if needed, and retry.
500 <code>internal_server_error</code>	An unexpected robot-side error, or <code>Routines</code> service is unavailable while the robot's services start.	Retry, and if the error persists, contact Standard Bots.
503 <code>service_initializing</code>	The robot software is still starting.	Retry after a few seconds.

A failed start leaves nothing running. A validation 400 names every offending field in its message, separated by semicolons. The message is prefixed with `Invalid request:` for the two required fields, or with `Invalid servoConfig:` for the tuning. For example, a message that names two `servoConfig` fields reads as follows, wrapped here to fit the page:

```
Invalid servoConfig: tauMs: Number must be less than or equal to 1000;
motionScalingLimits.jointVelocityMax: must be 6 values, one per joint
```

5.7.2 Tuning the Stream with servoConfig

`servoConfig` is optional tuning of the *Streaming controller*. It is honored on `joints` and `pose`, validated and then ignored on `jointTrajectory`, and ignored entirely on robots without the capability. Each field is optional and defaults independently, so you can send just the one you want to change. A field that belongs to the other stream type still has to be valid but has no effect, so one request shape works for both.

Field	Default	Range	Applies to
<code>tauMs</code>	20 ms	1 to 1000	<code>joints</code> , <code>pose</code>
<code>commandTimeoutMs</code>	500 ms	1 to 2147483647	<code>joints</code> , <code>pose</code>
<code>rampInDurationMs</code>	300 ms	0 to 2147483647	<code>joints</code> , <code>pose</code>
<code>positionSmoothingTimeConstantMs</code>	20 ms	0 to 1000	<code>joints</code>
<code>referenceFrame</code>	"base"	"base" Or "world"	<code>pose</code>
<code>motionScalingLimits</code>	absent	Two six-value arrays	<code>pose</code>

Each field works as follows:

- `tauMs` is the time constant of the first-order low-pass filter applied to your targets, which are joint positions on `joints` or the tool pose on `pose`. A smaller value is snappier and a larger one is smoother, and the arm trails your stream by roughly the time constant you set.
- `commandTimeoutMs` is the stream-staleness window, armed by your first setpoint. If no fresh setpoint arrives inside it, the arm decelerates to rest and the session ends.
- `rampInDurationMs` is the soft start. The arm eases from its current position, or its current tool pose, to your first setpoint over this window, clocked from that setpoint, so a session that begins with the arm far from your first target does not jump. Setting it to 0 disables the ramp. The Streaming Tool-Point Poses, with Optional Twist section explains how to size it.
- `positionSmoothingTimeConstantMs` is the window over which each setpoint's deviation from the position-plus-velocity extrapolation is absorbed, instead of in a single 1 ms control cycle. 0 applies the deviation immediately.
- `referenceFrame` is the frame of your poses and twists. "world" also applies the robot's configured base placement. If no base placement is configured, the two frames coincide.
- `motionScalingLimits` sets per-joint speed and acceleration caps that the controller slows the stream to. It takes a `jointVelocityMax` array in rad/s and a `jointAccelerationMax` array in rad/s^2 , each holding six values for `joint0` through `joint5`, and every value has to be finite and greater than 0. Both arrays are required together. Omit the object for no per-joint cap.

All millisecond fields are integers. Out-of-range, wrong-type, wrong-length, and non-finite values are rejected with a 400 rather than clamped.

WARNING – Omit a Field to Get Its Default, and Never Send `null`

A `null` is a validation error, not a fallback. The Python SDK serializes every unset field, including `servoConfig` itself, as `null`, as described in the Starting and Stopping the Session from the SDK section.

NOTE – Misspelled Fields Fail Silently

Unrecognized keys inside `servoConfig` are dropped without an error, so a misspelled field name fails silently and you get the default instead. Check your spelling against the table above.

`motionScalingLimits` slows the stream rather than rejecting it. Every cycle, the commanded joint velocity is scaled so that no joint exceeds `jointVelocityMax`, and then rate-limited so that no joint exceeds `jointAccelerationMax`. The arm lags the stream until it catches up, and once the lag exceeds the tracking bound, the session ends with `TRACKING_ERROR_EXCEEDED`.

Independently of those caps, the arm's own per-joint commanded-acceleration limit, 5 rad/s^2 on RO1, still applies, and exceeding it ends the session with `JOINT_COMMAND_OUT_OF_LIMITS`. If you hit it, raise `tauMs` or tighten `jointAccelerationMax`. A small `tauMs` combined with a tight `jointAccelerationMax` can overshoot a stopped target. The overshoot is bounded, and the `servo1_overshoot.py` SDK example predicts it.

While a session runs, the **Settings > External control > Streaming** page in the robot UI shows the effective tuning.

5.7.3 Stopping a Session

`POST /api/v1/external-control/stop` stops the running external-control session, releases the lease, and tears down the bridge. It takes no request body, and it returns 200 with:

```
{ "stopped": true }
```

`stopped` is `false` when there was nothing to stop: the robot was idle, an operator's routine is running, which `stop` never ends, or the session already ended on its own and the robot is in `Failure`. In that last case, recover the robot and then `start`.

`stop` returns after teardown and settle, capped at 2 seconds, so an immediate `start` is safe. It ends the running session no matter who started it, whether that was you, another client, or the robot UI. There is no session owner and no owner heartbeat, so a session never ends just because its client went away. Watch for a session ending under you, where the topics vanish and `GET /api/v1/recovery/status` leaves `RoutineRunning`, rather than assuming the session is yours until you call `stop`.

An operator's own routine is stopped from the robot UI or with `POST /api/v1/routine-editor/routines/{id}/stop`, not with this endpoint.

5.8 Control and Safety Semantics

- **Control is held for the session's lifetime.** Starting the session acquires control once, and there is no per-command engage or disengage. Stopping the session releases it.
- **A stalled joints or pose stream is fatal** on robots with *Streaming controller*. Once your first setpoint is accepted, a gap longer than `servoConfig.commandTimeoutMs`, 500 ms by default, decelerates the arm and ends the session. The deceleration is in joint space on `joints`, and along a Cartesian

ramp on pose. Recover the robot, and then start again. To idle mid-session, keep republishing the current setpoint, with a zero twist on pose, or raise `commandTimeoutMs` at start. `jointTrajectory` has no watchdog. The failure reason reads `CLIENT_STREAM_TIMEOUT`, and nothing else is sent, so watch your own publish rate.

- **Single writer.** There is one session per robot, so a second start, from any client or from the robot UI, gets 409 `robot_not_idle`. There is one writer per session. A ROS2 bridge holds the writer slot for the whole session, while in an `api` session UDP clients pass the slot between them with `take_control` and `release_control`, as described in the Direct UDP Interface section.
- **The robot UI can see and stop the session.** The UI shows the running session's `controlledBy`, `streamType`, and effective `servoConfig`, read-only, on the **Settings > External control > Streaming** page, with a **Stop** button that does the same as `stop`. **Pause** ends the session too. Gripper actuation from the UI is refused with “Streaming active” while a session holds the gripper.
- **Every end is a controlled stop.** A `stop`, a fault, and a robot-side pause all release control and bring the arm to a controlled stop. On pose it is a Cartesian stop, at 1 m/s² and 2 rad/s² by default, with orientation held. Stop publishing once you call `stop`. A dropped client does not end the session. Only the watchdog, a fault, or `stop` does.
- **Safety always wins, and it ends the session.** Joint, velocity, and torque limits, collision detection, and E-stop are enforced by the robot regardless of external control. Any of them, any pause from the robot UI, and any controller fault on the stream, such as a singularity, a stream jump, a tracking error, or a joint command out of limits, ends the session, and there is no resume. `.../status` does not change, `.../robot_joints` goes still, and the topics vanish. Read the cause from `GET /api/v1/recovery/status`.

5.9 What Ends a Session and How to Recover

Only `stop` ends a session cleanly. Every other end leaves the robot in `Failure`, not `Idle`, and “recover” below means the procedure in the Recovering from a Failure section. The last case in this list does not end the session at all.

- **A `stop`**, whether yours, another token holder's, or the **Stop** button in the robot UI. The arm makes a graceful stop, and `stop` returns once it has settled, typically within a few seconds. The session ends, `stop` returns `{ "stopped": true }`, and the robot is `Idle`, so you can start again.
- **A joints or pose stream that stalls past `commandTimeoutMs`.** The arm decelerates to rest, along a Cartesian ramp on pose, and the session ends. `failure.reason` reads `External Control Step failed: The command stream stopped, so the streaming session ended (code=CLIENT_STREAM_TIMEOUT ...)`. Recover, then start.
- **A controller fault on the stream.** The controller makes its fast stop, in joint space, and the session ends. `failure.reason` embeds one of the codes `STREAM_JUMP`, `TRACKING_ERROR_EXCEEDED`, `IN_SINGULARITY`, `COMPUTATION`, `JOINT_COMMAND_OUT_OF_LIMITS`, `TRAJECTORY_OVER_LIMITS`, or `TC_CONTROLLER_ERROR`. Recover, fix the stream, and start. The faults that apply to a pose stream are detailed in the Controller Faults on a Pose Stream section below.

- **A collision, a protective stop, a control-system stop, or Pause in the robot UI.** The arm stops, the robot pauses, and the session ends at once, because a paused session cannot resume. `failure.kind` is `StepPlayFailure`, and `failure.reason` reads `External control session ended: <cause>`. Start a new session to resume streaming. Recover, then start. Whether you can recover the robot yourself depends on `recovery_type`, described below.
- **A joint reaching its limit.** The arm stops and the session ends. `failure.kind` is `OutOfJointLimitsFailure`. A person has to recover the robot at the robot itself, in guided mode, before you start again.
- **An E-stop or an arm power-off.** Arm power is cut and the session ends. Read the state from `GET /api/v1/recovery/status`. Release the E-stop or restore power, with no `recover` call needed, re-check the brakes as shown in Step 1, and then start.
- **Loss of internal communication on the robot.** The arm stops and the session ends. Read the state from `GET /api/v1/recovery/status`, `recover`, and `start`.
- **Your UDP client going silent for 5 seconds or sending `release_control`, or the bridge process dying.** Nothing happens to the arm directly: the writer slot is freed, or the topics vanish. A watchdogged stream will already have hit the stall case above, and a `jointTrajectory` session keeps holding. The session **continues**. On UDP, you find out from the reply to your next `take_control`, as described in the Direct UDP Interface section. On ROS2, the `/external/*` publishers disappear while REST still reports `RoutineRunning`. Send `take_control` again, or `stop` and then `start` for a fresh bridge.

5.9.1 Controller Faults on a Pose Stream

Each of these conditions stops the arm and then ends the session. The bounds are robot-side defaults, and the tuning fields are described in the Tuning the Stream with `servoConfig` section.

Code in <code>failure.reason</code>	Condition and notes
<code>STREAM_JUMP</code>	Two consecutive setpoints are too far apart. The bound is 0.1 m or 0.5 rad. The first setpoint is exempt, because the ramp-in absorbs it.
<code>TRACKING_ERROR_EXCEEDED</code>	The target ran away from the arm. The bound is 0.4 m or 1.5 rad between the target and the tracked pose. Causes include an unreachable pose, <code>motionScalingLimits</code> set too tight, and a far first setpoint with <code>rampInDurationMs: 0</code> .
<code>IN_SINGULARITY</code>	A singularity. A session may start at a singular pose and hold there, but once anything has moved, a target at or through a singularity faults. Plan clear of singularities, because the controller does not steer around them.
<code>COMPUTATION</code>	The stop ramp has no IK solution. The stream is integrated in joint space, and IK runs only on the deceleration ramp. If there is no solution there, the arm stops faster, in joint space.

Code in <code>failure.reason</code>	Condition and notes
<code>JOINT_COMMAND_OUT_OF_LIMITS</code>	The commanded joint acceleration is above the robot's fixed per-joint limit. The message reads "A commanded joint position is outside the arm's limits", but on a pose stream this code means acceleration. Raise <code>tauMs</code> or add <code>motionScalingLimits</code> .
None	A commanded joint position is beyond a joint's range, or more than 0.2 rad from the measured position. This is a control-system stop, not a controller fault. The command is not scaled or clamped: the arm's own validator refuses it, the arm stops gracefully, and the session ends with no code. The robot UI shows Robot encountered a control system issue: Invalid trajectory point.... Stay inside the joint ranges yourself.

5.9.2 Recovering from a Failure

While the robot is in Failure, `stop` returns `{ "stopped": false }` and `start` returns 409 `robot_not_idle` with `state: Failure`. Nothing recovers automatically. Read the failure, recover, and start again:

```
curl http://<robot-ip>:3000/api/v1/recovery/status \
  -H "Authorization: Bearer <token>"
# => { "status": "Failure", "failed": true,
#      "failure": { "kind": "StepPlayFailure",
#                  "reason": "External Control Step failed: ...",
#                  "recovery_type": "Recoverable", ... } }
```

```
curl -X POST http://<robot-ip>:3000/api/v1/recovery/recover \
  -H "Authorization: Bearer <token>"
# => { "status": "Idle", "failed": false }
```

```
curl -X POST http://<robot-ip>:3000/api/v1/external-control/start \
  -H "Authorization: Bearer <token>" \
  -H "Content-Type: application/json" \
  -d '{"controlledBy": "rosJazzy", "streamType": "joints"}'
```

`recovery_type` tells you whether `recover` will succeed on its own:

recovery_type	Typical causes	Who recovers
Recoverable	A controller fault, a stalled stream, a pause, a position or velocity protective stop, or a collision deviation.	You. Call <code>recover</code> , then <code>start</code> .
GuidedMode, ManualRecoveryMode, Or NotRecoverable	A collision, a torque protective stop, or a joint at its limit.	A person at the robot, first. <code>recover</code> alone does not succeed.

An E-stop or an arm power loss needs no `recover`: the failure clears once the E-stop is released and power is back. The Recover button in the robot UI and `sdk.recovery.recover.recover()` in the Python SDK do the same as `recover`.

5.9.3 Reading Why a Session Ended

GET `/api/v1/recovery/status` keeps the reason until you recover:

```
{
  "status": "Failure",
  "failed": true,
  "failure": {
    "kind": "StepPlayFailure",
    "reason": "External Control Step failed: ...",
    "recovery_type": "Recoverable",
    "is_recoverable_with_wrist_button": false
  }
}
```

The full reason in that example is a single line, wrapped here to fit the page:

```
External Control Step failed: No new setpoint arrived in time, so the
arm stopped (code=STALENESS_TIMEOUT severity=FAULTED
clearableBy=CLEARABLE_BY_CLIENT)
```

`failure.reason` takes one of two shapes. Match on its text, not on `kind`, which is `StepPlayFailure` even for a collision.

When a controller fault or a stalled stream ended the session, the reason has this shape, where a stalled stream reads `code=CLIENT_STREAM_TIMEOUT`:

```
External Control Step failed: <message> (code=<ErrorCode>
severity=<Severity> clearableBy=<ClearableBy>)
```

When an operator pause or a pausing safety failure ended the session, the reason has this shape, where a pause reads the routine was paused:

```
External control session ended: <reason>. Start a new session to
resume streaming.
```

Both shapes are single lines in the actual response. Poll `status` to detect the end of a session: it moves from `RoutineRunning` to `Failure` or `Idle`.

5.10 Streaming from Python without ROS2

If you are not using ROS2, start the session with `controlledBy: "api"` and stream setpoints with the Standard Bots Python SDK's realtime `RobotStream` client instead of publishing ROS topics. The semantics in the Control and Safety Semantics section apply unchanged. Install the `standardbots` package as described in the On Your Machine section. The realtime client and its value types import from one module:

```
from standardbots.realtime import (
    RobotStream, CartesianPose, CartesianTwist, JointState,
    Point, Quaternion, Vector3,
)
```

Session tuning still happens on the HTTP `start` call, not on the client. `RobotStream.start()` takes no configuration, so `servoConfig` has to go in the `external-control/start` body regardless of which of the three transports you then use to stream. The SDK can make that HTTP call for you, as shown in the Starting and Stopping the Session from the SDK section below.

5.10.1 RobotStream at a Glance

```
# Always UDP port 30100; the port is not configurable.
stream = RobotStream("<robot-ip>")

stream.start() # open the socket, subscribe as a reader
joints, ts_us = stream.get_joints() # block for the next state broadcast

# ... POST /api/v1/external-control/start with controlledBy "api" ...
stream.take_control() # become writer; RuntimeError if rejected
stream.send_joints(list(joints.position))
stream.release_control()
# ... POST /api/v1/external-control/stop ...

stream.stop() # unsubscribe; releases control if held
```

with RobotStream(ip) as stream: calls start() on entry and stop() on exit, but you still call take_control() and release_control() yourself.

The calls work as follows:

- RobotStream(host) takes the host only. It binds an ephemeral local port, and the robot replies to that port.
- start(), stop(), and is_running() handle the lifecycle and take no arguments. stop() sends Unsubscribe, which also releases control if you are the writer.
- take_control(timeout=2.0) promotes you to writer. It waits up to timeout for the first state broadcast, and then up to timeout again for the robot's answer. A rejection raises RuntimeError("TakeControl rejected by server: <reason>"), with the reasons listed in the Control Responses and Rejection Reasons section.
- release_control() returns you to being a reader. It raises RuntimeError if the robot rejects it with not the current writer, and only warns if the stream is not running or the robot does not reply.
- send_joints(positions, timestamp_us=0, velocities=None) sends one setpoint in radians, ordered joint0 through joint5. velocities are in rad/s, one per position, as described in the Velocities and Twist section below.
- send_pose(pose, timestamp_us=0, twist=None) sends one pose as CartesianPose(position=Point(x, y, z), orientation=Quaternion(x, y, z, w)) in meters, with an optional twist=CartesianTwist(linear=Vector3, angular=Vector3) in m/s and rad/s. Which frame and which point it drives is described in the Pose Readback Versus the Streamed Pose section below.
- send_trajectory(points, joint_names=(), timestamp_us=0) sends points = [[q0, ..., q5], time_from_start_s, ...]. Always pass joint_names=("joint0", ..., "joint5"), as described in the Trajectories and Cancel section below.
- send_gripper(position, velocity=0.0, timestamp_us=0) is normalized to the range 0 to 1, where velocity=0 means the gripper default. It works in any session.
- send_cancel() aborts the in-flight trajectory, and is a no-op otherwise.
- get_joints(timeout=5.0) returns (JointState, timestamp_us). It blocks for the **next** broadcast, and raises TimeoutError if none arrives within timeout. JointState carries name, position in radians, velocity in rad/s, and effort in newton-meters.
- get_pose(timeout=5.0) returns (CartesianPose, timestamp_us), which is the flange pose in the base frame, in meters, as described in the Pose Readback Versus the Streamed Pose section below.
- get_gripper(timeout=5.0) returns (position, timestamp_us), normalized to the range 0 to 1. It times out on a robot with no gripper.
- on_joints(cb), on_pose(cb), and on_gripper(cb) call cb(value, timestamp_us) on every broadcast, from the receive thread.

timestamp_us on every send_* call is an optional client send-time in microseconds, where 0 means unset. The timestamp_us returned with state is the robot's state timestamp, as a Python float.

5.10.2 Order of Operations with the HTTP Session

The robot's UDP listener runs without a session, as described in the Direct UDP Interface section, so `start()`, the `get_*` calls, and the `on_*` callbacks all work **before** the HTTP start. Use them to read the arm's position and seed your first setpoint. `take_control()` also succeeds without a session, but setpoints sent outside a session are discarded with no error. Work in this order:

1. Create the stream with `RobotStream(ip)` and call `stream.start()`, then read the arm with `get_joints()` or `get_pose()`.
2. POST `/api/v1/external-control/start` with `controlledBy: "api"` and your `streamType` and `servoConfig`, either with `curl` or from the SDK as shown below.
3. Call `stream.take_control()`, and then stream setpoints at a steady rate.
4. Call `stream.release_control()`, then POST `/api/v1/external-control/stop`, and finally call `stream.stop()`.

WARNING – Stop the HTTP Session Before Closing the Stream

Closing the stream first lets the setpoint stream go stale, so a watchdogged session ends with a stream-timeout failure instead of a clean stop. See the What Ends a Session and How to Recover section.

5.10.3 Starting and Stopping the Session from the SDK

```
from standardbots import StandardBotsRobot, models

sdk = StandardBotsRobot(url="http://<robot-ip>:3000", token="<token>",
                        robot_kind=StandardBotsRobot.RobotKind.Live)

routine_id = sdk.external_control.start(models.StartExternalControlRequest(
    controlledBy=models.ExternalControlControlledBy.Api,
    streamType=models.ExternalControlStreamType.Joints,
    # Populate every field; see the warning below.
    servoConfig=models.ExternalControlServoConfig(...),
)).ok().routineId

# ... stream with RobotStream ...

# .stopped is False if no session was running
sdk.external_control.stop().ok()
```

`.ok()` returns the parsed body, or raises on any non-200 response with the robot's error code. The error table in the Starting a Session section applies unchanged.

WARNING – The SDK Sends Unset Fields as `null`

The SDK sends every unset field as JSON `null`, and the robot rejects `null` in `servoConfig`. Omit-

ting `servoConfig`, or any field inside it, fails start with 400 `request_failed_validation`. Either populate **every** `ExternalControlServoConfig` field, including the ones your stream type ignores, as `servo_sphere_demo.py` does, or make the two HTTP calls with another client and use the SDK only for `RobotStream`. Note that `motionScalingLimits` takes six positive float values per array, not int values.

5.10.4 Threading, Errors, and Liveness

`start()` runs one daemon receive thread that updates the latest state and runs your `on_*` callbacks. Keep callbacks short, and never call `get_*`, `take_control()`, or `release_control()` from inside one, because they wait on the very thread you are on. An exception raised in a callback is logged and swallowed.

`send_*` is fire-and-forget. It raises `RuntimeError` before `start()` or after `stop()`, and `OSError` on socket errors, but never for a dropped setpoint, because the robot reports none. Stream only after `take_control()` returns. The `get_*` calls raise `TimeoutError` after `timeout`, 5 seconds by default, which is the usual first symptom of a wrong IP address or a blocked UDP port 30100.

Liveness is handled for you. `Subscribe` is re-sent every second until the first broadcast arrives, and a `Heartbeat` goes out after 2 seconds of silence, inside the robot's 5-second eviction window. Heartbeats keep the *subscription* alive, not the *setpoint stream*, so `commandTimeoutMs` still applies.

Frames arrive at about 200 Hz, with `pose` and `joints` together. `gripper` is present only once a connected gripper reports, so `get_gripper()` timing out while everything else works means there is no gripper, not that there is no connection. Broadcasts stop when the arm's state stops, and there is no substitute heartbeat.

`on_*` registrations are permanent. There is no way to unregister one, and a function registered twice runs twice. Calling `stop()` and then `start()` on the same object gives you a clean slate with a new local port, so call `take_control()` again. `is_running()` goes false after `stop()` or after a socket failure.

5.10.5 Velocities and Twist

`send_joints(velocities=...)` and `send_pose(twist=...)` always put the field on the wire, but the arm uses it only with *Streaming controller*, as a first-order hold between setpoints, as described in the Command Topics section. On those robots, a `velocities` list whose length differs from `positions`, or any non-finite entry in `velocities` or `twist`, drops the whole setpoint silently. Sustained, that stalls the stream, and the session ends. On other robots the field is ignored.

5.10.6 Pose Readback Versus the Streamed Pose

`get_pose()` and `on_pose` return the wrist-flange pose in the robot base frame, on every robot. Without *Streaming controller*, `send_pose()` targets that same point. With it, `send_pose()` commands the **active tool center point** in `servoConfig.referenceFrame`, so seeding your stream from `get_pose()`, as `robot_stream_poses.py` does, moves the arm by the tool offset when the stream starts.

Seed from the REST tooltip pose instead, `sdk.movement.position.get_arm_position().ok().tooltip_position`, as `servo_sphere_demo.py` does, with `referenceFrame` set to that pose's frame.

5.10.7 Trajectories and Cancel

`send_trajectory()` and `send_cancel()` act only on robots with *Trajectory execution*. On other robots the call returns normally and the arm holds, with nothing to tell the client. Where they do act, pass `joint_names=("joint0", ..., "joint5")` and exactly that many positions per point, or the trajectory is dropped silently.

5.10.8 SDK Examples

Runnable examples live in the `examples/` directory of the package source. They hard-code `ROBOT_IP = "localhost"`, and the demo also hard-codes `ROBOT_URL` and `ROBOT_TOKEN`, so edit them before running.

Example	What it shows
<code>robot_stream_joints.py</code>	A joints session that swings joint 0 with <code>send_joints</code> at 50 Hz. For your own streams, use a steady rate of roughly 100 to 250 Hz, as described in the Streaming Joint Positions, with Optional Velocity section.
<code>robot_stream_poses.py</code>	A pose session that offsets the flange pose from <code>get_pose()</code> along X, Y, and Z. See the Pose Readback Versus the Streamed Pose section.
<code>robot_stream_trajectory.py</code>	A jointTrajectory session that sends one 20-point <code>send_trajectory</code> out and back. Needs <i>Trajectory execution</i> .
<code>robot_stream_gripper.py</code>	<code>send_gripper</code> from open to closed to open, with <code>on_gripper</code> feedback.
<code>servo_sphere_demo.py</code>	A complete pose-streaming run: unbrake and home over REST, the HTTP start from the SDK with every <code>servoConfig</code> field, streaming targets, and the HTTP stop.
<code>servo_overshoot.py</code>	Runs offline and predicts the worst-case overshoot for the limits used in the sphere demo.

Only `servo_sphere_demo.py` starts the HTTP session itself. The `robot_stream_*` scripts need a session of the matching `streamType` started first, as shown in the Getting Started with Joint Streaming section, and the gripper script needs a session of any type.

Not using Python either? Both the ROS2 bridge and the Python SDK sit on top of a documented protobuf-over-UDP protocol that you can speak directly from C++, or from any other language. That protocol is described next.

5.11 Direct UDP Interface

The ROS2 bridge and the Python SDK's `RobotStream` are both thin clients over one lower-level transport: a protobuf-over-UDP stream served directly by the robot. If you are using neither ROS2 nor the Python SDK, for example because you are writing a C++ client, you talk to this interface directly.

The contract is `external_control.proto`, package `external_control`. Ask Standard Bots for the file, because the Python SDK ships only pre-generated bindings and not the source. Compile it for your language with `protoc`. The Python SDK, in `standardbots/realtime/robot_stream.py` and `protocol.py`, is the reference implementation.

5.11.1 Transport

Property	Value
Protocol	UDP
Server endpoint	<robot-ip>:30100, on the control box.
Client endpoint	Any local IPv4 port. Bind an ephemeral port with <code>bind("", 0)</code> , and the server replies to your datagram's source address. That IP address and port are your identity for the subscription, the writer slot, and liveness, so use one socket for the whole session . A new socket is a new, unsubscribed client, and the old writer slot frees only when the old identity is evicted, after 5 seconds. IPv6 is not supported.
Wire format	One serialized protobuf message per datagram: <code>ClientMessage</code> from you to the robot, <code>ServerMessage</code> from the robot to you. There is no length prefix and no framing, so the UDP payload <i>is</i> the protobuf.
Datagram size	One setpoint or state snapshot per packet, comfortably under one MTU. The reference client reads with <code>recvfrom(65535)</code> .

Reading needs no session. `Subscribe` alone delivers state broadcasts whenever the robot's external-control service is up.

Moving the arm needs a session. `POST /api/v1/external-control/start` with `controlledBy: "api"` takes the control lease, and then `take_control` claims the UDP writer slot. The two are independent: `take_control` succeeds without a session, but the robot then drops every arm setpoint and sends nothing back. `gripper_command` is the exception, as described in the Messages You Send section below. Call `POST /api/v1/external-control/stop` when you are done.

5.11.2 Reader and Writer Roles

A client is first a reader, and then optionally promotes itself to the single writer:

1. `Subscribe` makes you a reader and starts the `ServerMessage` broadcasts.

2. TakeControl requests exclusive write. The server replies with a ControlResponse.
3. JointCommand, PoseCommand, **and the other setpoints** are applied, but only from the active writer.
4. ReleaseControl puts you back to being a reader. You still receive state.
5. Unsubscribe stops the broadcasts, and releases control first if you are the writer.

Two rules govern this:

- TakeControl **needs a prior** Subscribe, **and** Subscribe **is not acknowledged**. The first broadcast is your only confirmation, and none arrives while the arm's state is not flowing. UDP may reorder datagrams, so a TakeControl sent right after Subscribe can be rejected with `not subscribed`. Wait for one broadcast first, and on `not subscribed`, re-send Subscribe and retry once.
- **There is one writer at a time, across transports**. In a ROS2 session the bridge holds the slot, so a UDP `take_control` is answered with `rejected by orchestrator` (another transport has control). In an api session the first `take_control` wins, and after a `release_control` or an eviction the next one wins, within the same session. Setpoints from a client that is not the writer are dropped silently. Releasing control, unsubscribing, or being evicted frees only the writer slot, and the HTTP session stays up.

5.11.3 Liveness and Heartbeats

The server evicts any client whose last **accepted** message is more than **5 seconds** old. An accepted message is a subscribe, heartbeat, `take_control`, or `release_control`, or a setpoint from the **current writer**. Setpoints from a client that is not the writer, and malformed datagrams, refresh nothing.

- Send a Heartbeat at least every 5 seconds. The reference client sends one every 2 seconds when idle, which is a good interval to match.
- Eviction is **silent**. Broadcasts stop, and your next `take_control` is answered with `not subscribed`. Re-send Subscribe to come back.

NOTE – Eviction Is Not a Stop

Eviction frees only the UDP writer slot, and the HTTP session and its lease stay up. A watchdogged stream will already have ended the session long before 5 seconds pass. A `jointTrajectory` session, or a pose session without *Streaming controller*, holds position and stays open until you stop.

5.11.4 Messages You Send

ClientMessage is a `oneof` envelope, so set exactly one field per datagram.

Field	Purpose
<code>subscribe</code>	Register as a reader. Idempotent, and not acknowledged.
<code>take_control</code>	Request exclusive write. Requires a prior <code>subscribe</code> .
<code>release_control</code>	Relinquish write and stay a reader.
<code>unsubscribe</code>	Stop broadcasts. Releases control first if you are the writer.

Field	Purpose
heartbeat	Keepalive, as described above.
joint_command	A joint-space setpoint.
pose_command	A Cartesian pose setpoint, with an optional <i>twist</i> .
trajectory_command	A whole multi-point path. <i>Trajectory execution</i> robots only.
gripper_command	A gripper setpoint.
cancel_trajectory	Abort the in-flight trajectory. <i>Trajectory execution</i> robots only, and a no-op otherwise.

`take_control` and `release_control` are each answered with a `ControlResponse`.

The command messages carry these payloads:

- `joint_command` carries `position[]` in radians, ordered `joint0` through `joint5`, an optional `velocity[]` in rad/s, and `timestamp_us`. An empty `position[]` is dropped. With *Streaming controller*, a `velocity[]` has to carry one finite entry per position or the setpoint is dropped silently. Without the capability, `velocity[]` is ignored.
- `pose_command` carries `pose`, with `position` in meters and orientation as a unit quaternion, an optional `twist`, with `linear` in m/s and `angular` in rad/s, and `timestamp_us`. With *Streaming controller*, the pose is the tool center point in `servoConfig.referenceFrame` and the twist is carried forward between setpoints. The setpoint is dropped silently when `pose` is unset, when any value is non-finite, or when the orientation's norm is more than $1e-3$ away from 1. Without the capability, the pose is the flange in the base frame and the twist is ignored. `ServerMessage.pose` is always the flange.
- `trajectory_command` carries `joint_names[]`, `points[]` of { `position[]`, `time_from_start_seconds` }, and `timestamp_us`. Note that `time_from_start_seconds` is a single double in seconds, unlike the ROS2 `sec` and `nanosec` split. It needs *Trajectory execution*, and on other robots it is accepted and ignored. It is dropped silently when `joint_names` or `points` is empty, when a point's `position` count differs from `joint_names`, or when the arm's executor is not ready. The arm owns execution timing and does not re-interpolate your points.
- `cancel_trajectory` has no payload. It cancels in-flight trajectories on *Trajectory execution* robots, and is a no-op elsewhere. Session teardown issues the same cancel.
- `gripper_command` carries `position`, from 0 closed to 1 open, and `velocity`, where 0 means the gripper default, both in the range 0 to 1, plus `timestamp_us`. It is applied from the current writer regardless of the session and `streamType`, so **it still actuates the gripper even when no session is running**. It is dropped silently when no gripper is connected, as described in The Gripper Channel section.

`timestamp_us` is an optional client send-time in microseconds, and 0 means unset.

These mirror the ROS2 command topics described in the Command Topics section. The one addition is the optional `twist` on `pose_command`, which ROS2 cannot send, because `geometry_msgs/PoseStamped` has no velocity field.

5.11.5 Messages You Receive

The server sends a `ServerMessage` to every subscribed reader on each arm-state update, and a separate one as the direct reply to `take_control` and `release_control`. Broadcasts follow the robot's state rather than a timer, at **200 Hz** by default, which is a per-robot setting. Sends closer together than **2 ms** are coalesced into one frame that carries the latest value of every field, so `pose` and `joints` may be one tick apart.

WARNING – No Arm State, No Frames

The robot sends no idle heartbeat. If `ServerMessage` broadcasts stop arriving while your subscription is alive, the robot's arm-state stream has stalled, so treat it as a link-down.

Field	Contents
<code>pose</code>	Live flange pose in the robot base frame. It is not the tool point, and it is not converted to <code>servoConfig.referenceFrame</code> .
<code>joints</code>	Live <code>JointState</code> : name, position in radians, velocity in rad/s, effort in newton-meters.
<code>gripper</code>	<code>GripperState.position</code> , normalized to the range 0 to 1. Present only once a connected gripper reports state. Updates at 10 Hz when idle and 30 Hz during a session.
<code>timestamp_us</code>	Server send time in microseconds, on a monotonic clock. Use it for ordering and intervals only, and never compare it to your own clock.
<code>control_response</code>	Present <i>only</i> on a reply to <code>take_control</code> or <code>release_control</code> , as <code>{ success, reason }</code> . A reply carries nothing else.

Branch on `has_control_response()`: a reply answers your request, and a broadcast carries state. Never update your cached pose or joints from a reply.

5.11.6 Control Responses and Rejection Reasons

`take_control` and `release_control` are acknowledged with a `ControlResponse` carrying success and reason:

- `success = true` with an empty `reason` means the request was granted. An idempotent re-take by the current writer also returns success.
- `success = false` with `not subscribed` means `take_control` arrived without a prior subscribe, including after a silent eviction.
- `success = false` with `another client already has control` means another UDP client is the writer.
- `success = false` with `rejected by orchestrator` (another transport has control) means the ROS2 bridge, or another transport, holds control.
- `success = false` with `not the current writer` means `release_control` came from a client that is not the writer.

NOTE – Older Servers May Not Reply at All

Older servers may not send a `ControlResponse`. If you get no reply within a short timeout of roughly 2 seconds, treat the request as best-effort succeeded, which is what the reference client does for backward compatibility.

5.11.7 A Minimal C++ Client

Generate the protobuf bindings with `protoc --cpp_out=. external_control.proto`, then drive a POSIX UDP socket. Error handling is elided here:

```
#include <arpa/inet.h>
#include <sys/socket.h>
#include <unistd.h>
#include "external_control.pb.h"    // protoc-generated

int fd = ::socket(AF_INET, SOCK_DGRAM, 0);
sockaddr_in robot{};
robot.sin_family = AF_INET;
robot.sin_port = htons(30100);
::inet_pton(AF_INET, "192.168.1.100", &robot.sin_addr);

auto send_msg = [&](const external_control::ClientMessage& m) {
    std::string buf = m.SerializeAsString();    // no length prefix
    ::sendto(fd, buf.data(), buf.size(), 0,
             (sockaddr*)&robot, sizeof(robot));
};

// After the HTTP start with controlledBy:"api":
// 1) Subscribe 2) TakeControl
{ external_control::ClientMessage m;
  m.mutable_subscribe(); send_msg(m); }
// ...Subscribe is unacknowledged: wait for one state
// broadcast before sending TakeControl...
{ external_control::ClientMessage m;
  m.mutable_take_control(); send_msg(m); }
// ...read ServerMessage.control_response().success() before streaming...

// 3) Stream joint setpoints (position-only; add velocity only if
// your robot has the Streaming controller capability)
{
  external_control::ClientMessage m;
```

```

auto* jc = m.mutable_joint_command();
for (double q : {0.0, -1.57, 1.57, 0.0, 1.57, 0.0})
    jc->add_position(q);
send_msg(m);                                     // repeat at a steady rate
}

// Receive state and control responses
char rx[65535];
sockaddr_in from{}; socklen_t flen = sizeof(from);
ssize_t n = ::recvfrom(fd, rx, sizeof(rx), 0,
                      (sockaddr*)&from, &flen);

external_control::ServerMessage sm;
sm.ParseFromArray(rx, static_cast<int>(n));
// sm.has_control_response()
// ? reply to take_control / release_control (no state)
// : periodic state broadcast (pose, joints, gripper if any)

// On exit: ReleaseControl, then Unsubscribe, then the HTTP stop.

```

Stream `joint_command` at a steady rate of roughly 100 to 250 Hz with small per-tick position deltas, exactly as you would on the ROS2 `.../joints` path. The same safety and limit enforcement applies.

5.12 Troubleshooting

- **start returns 409 robot_not_idle.** The message says which case applies. `state: RoutineRunning` means a routine or another session is running, so stop it. `state: Failure` means the previous session ended on its own, so recover the robot and then start. Otherwise, the arm has not come to rest, so retry after a moment.
- **start returns 400 brakes_must_be_disengaged.** Disengage the brakes as shown in Step 1, and re-check them after an E-stop or a power loss.
- **A call to the older `movement/ros/state` endpoint fails with 500 and `ControlSourceManager` is enabled.** That endpoint belongs to the older ROS2 Realtime API, and a robot running external control rejects it. Use `external-control/start` instead.
- **POST `/api/v1/recovery/recover` returns 500 `recovery_error`.** The E-stop is still pressed, or the arm is still powering off. Release the E-stop, or wait for arm power, and then retry.
- **recover succeeded after a collision, but start is refused.** The arm is still against the obstruction, and recover does not move it. Move the arm off the obstruction under guided control from the robot UI, and then start.
- **ros2 topic list shows no `/external/*` topics.** Check `ros2 node list`. If nothing from the robot appears, `ROS_DOMAIN_ID` or `RMW_IMPLEMENTATION` does not match the robot, or you are not on the robot's wired subnet. If `/external_control_ros_bridge` is listed but there are no topics, your ma-

chine reaches only one of the robot's advertised addresses. Move to the wired subnet, or have the operator restrict the robot's ROS2 interfaces.

- **A ROS2 session ends within seconds of start, no topics ever appear, and stop returns stopped: false.** The robot could not launch the bridge, either because the wired customer port has no IP address or because the robot lacks the ROS2 distribution you asked for. Recover the robot, check the wired link, and try the other `controlledBy` value. If both fail, contact Standard Bots.
- **The topics exist but the arm does not move, or one modality is rejected.** You are publishing to a topic that does not match the session's `streamType`, or the robot is stopped, which `GET /api/v1/recovery/status` shows. If `.../status` is also silent, the bridge never obtained control because another client holds the writer slot, typically a UDP client that sent `TakeControl`. Release or stop that client, or stop and start again.
- `.../status` **shows** `engaged=false reason=send_<op>-failed`. One command could not be delivered, and the bridge re-takes control on your next message. If it recurs, the robot is rejecting your commands, most likely because you are publishing to a topic that does not match the session's `streamType`.
- `.../trajectory` or `.../cancel` **does nothing, with no error anywhere.** Your robot lacks *Trajectory execution*. Use `joints` or `pose` instead.
- **The arm stops and every `/external/*` topic vanishes.** The bridge was torn down, and nothing is published at teardown. Check `GET /api/v1/recovery/status`. Failure means the session ended on a fault, a stall, or a pause, and `failure.reason` names which, so recover the robot and then start. `RoutineRunning` means the bridge process died while the session stayed up, so stop and then start for a fresh bridge. `Idle` means someone called stop, so start again.
- **The arm jerks, draws high torque, or E-stops while streaming joints.** The position steps between setpoints are too large, or you sent a single-point `jointTrajectory` at `time_from_start = 0`. Stream `sensor_msgs/JointState` at a steady rate with small per-tick deltas, and never use a one-point trajectory as a setpoint.
- **Setpoints are silently ignored even though you are the writer, on ROS2 or UDP.** A malformed setpoint is dropped with no reply. The usual causes are a non-unit or unset orientation, a NaN or infinite position or twist, and a velocity that is not one per position or is non-finite. If every setpoint from the first one is dropped, the watchdog never arms and the session stays open. If drops begin later, the stream stalls and the session ends.
- **A pose session ends with STREAM_JUMP, TRACKING_ERROR_EXCEEDED, or IN_SINGULARITY.** `STREAM_JUMP` means consecutive setpoints are too far apart, so interpolate on your side. `TRACKING_ERROR_EXCEEDED` means the target ran away from the arm, because of an unreachable pose, `motionScalingLimits` set too tight, or a far first setpoint with `rampInDurationMs: 0`. `IN_SINGULARITY` means the path crosses a singularity, so re-plan it. The bounds are listed in the Controller Faults on a Pose Stream section.
- **The arm keeps moving briefly after your pose stream stops.** The last setpoint carried a non-zero twist. Send a zero twist when you intend to hold.
- **The first pose lands off by the tool length.** You seeded your stream from the flange readback, while the stream commands the tool center point. See the Pose Readback Versus the Streamed

Pose section.

- **The arm decelerated and the session ended, and nobody hit an E-stop.** The stream stalled past `commandTimeoutMs`, or the robot was paused from its UI. `failure.reason` in `GET /api/v1/recovery/status` says which, reading either `code=CLIENT_STREAM_TIMEOUT` or the routine was paused. Check your publisher, recover the robot, and then start.
- **start keeps answering 409 with state: Failure after recover.** `recovery_type` says who can recover the robot. Recoverable means you can call `recover` yourself. Anything else needs someone at the robot to release the E-stop, restore arm power, or recover in the UI or with the wrist button.

On the direct UDP interface:

- **No ServerMessage ever arrives.** The host or port is wrong, you never sent `Subscribe`, you did not bind a local port for the reply, or the robot's arm-state stream is not flowing. A session is *not* required to receive state. `Subscribe` is not acknowledged, so with no broadcasts you cannot tell "not subscribed" apart from "robot not publishing". Check the robot first.
- **take_control is answered not subscribed right after Subscribe.** The `TakeControl` overtook the unacknowledged `Subscribe`. Wait for one broadcast first, and on `not subscribed`, re-send `Subscribe` and retry once. After a long silence, the same answer means you were evicted, as described below.
- **Setpoints are ignored and the arm does not move.** You are not the writer, which `ControlResponse.reason` tells you, or no session is running. The writer slot is granted independently of the session, and without a session every arm setpoint is dropped silently, except gripper commands. Call `start` with `controlledBy: "api"`, and then stream.
- **Broadcasts stop.** If a later `take_control` is answered `not subscribed`, you were evicted after 5 seconds without an accepted message. Send a `Heartbeat` at least every 5 seconds, remembering that setpoints from a client that is not the writer do not count, and then re-send `Subscribe`. If broadcasts stop while you are heartbeating, the robot's arm-state stream has stalled, so check the robot UI.

5.13 Known Limitations

- This interface drives the arm directly from your setpoints. It does **not** run MoveIt planning, collision-aware path planning, or trajectory time-scaling for you. That is your responsibility, or you can use the robot's higher-level planning APIs instead.
- Several behaviors are robot-side capabilities, as described in the Prerequisites section. Without *Streaming controller*, `servoConfig` is accepted and ignored and there is no command watchdog. Without *Trajectory execution*, `.../trajectory` and `.../cancel`, and their UDP counterparts, are accepted and ignored.
- Pose commands drive the tool center point in `servoConfig.referenceFrame` with *Streaming controller*, and the flange in the base frame without it. The readback is always the flange. No TF transform is applied, and `frame_id` is ignored.
- One control modality per session, set by `streamType`. Switch by restarting the session. Gripper and cancel are available in any session.

- Only one session, and one writer within it, can run at a time, and the robot UI can always see and stop it.
- GET `/api/v1/recovery/status` is the durable record of why a session ended.
- Topic names are fixed at `/<robotId>/external/<topic>`, or at `/local/external/<topic>` on a robot with no identity configured. They cannot be remapped from your side, so remap on your own nodes instead.

Chapter 6

Additional Resources

6.1 Python SDK & REST API Repo

`sb-rest-api`

6.2 ROS2 Repo

`ros2-realtime-api`

6.3 Robot URDF Files

StanardBots URDF Files



Standard Bots Company
80 Pratt Oval
Glen Cove, NY 11542 USA

www.standardbots.com
1-888-9-ROBOTS