Copyright 2026 Synthiam Inc.

# Third-Party Integrations

## Overview

ARCx does not ask you to throw away the robot software you already have. A robot built on ROS 2, a machine running Viam, a Python program you wrote last year, a dashboard on a tablet: each can work alongside ARCx and its robot skills, sharing the same sensors, the same cameras and the same motors. ARCx becomes the place where the parts meet, and its interface becomes the way you see and drive all of them.



This section explains what each integration does, what you need, and what to expect, so you can decide which one fits before you start. The robot skill manuals linked from each page have every setting.

---

## Contents

1. The Robot Bus: the common ground
2. Which integration do I need?
3. What ARCx adds
4. The pages in this section

---

## The Robot Bus: the common ground

Every integration in this section ends up on the **Robot Bus**, ARCx's shared layer of topics, services and frames. Robot skills, scripts and ARCx itself publish what they sense on topics (the lidar, the robot's pose, what the camera sees, what was heard), offer what they can do as services (drive, speak, move a servo, run a script), and place their sensors in a tree of frames.

Because everything meets there, an integration is never a dead end. A ROS topic carried onto the bus can be read by a Python program, recorded and replayed, watched in **Views** →

**Robot Bus**, or handed to an AI assistant. A Viam sensor's readings can drive an ARCx script. A service your own program provides can be called by a ROS node through the ROS Connector. You connect each system once, and everything else on the bus can use it. The Robot Bus page in this section explains it in full, including how programs in any language connect to it directly.

## Which integration do I need?

| You have | Use | What you get |
|---|---|---|
| A robot running **ROS 2** (or ROS 1) | The **ROS 2 Connector** robot skill | ROS topics and services joined to ARCx: a ROS navigation stack driving the robot, a ROS lidar in ARCx's mapping, ARCx's camera and sensors in ROS, ROS services called from ARCx scripts. |
| A machine running **Viam** | The **Viam Robot Controller** robot skill | Every component on the machine in ARCx: its base as the movement panel, its servos, arms, grippers and motors on servo ports, its cameras as video sources, its sensors as variables. |
| Your own program in **Python, JavaScript or any language** | The **Robot Bus** API, built into ARCx | Read every topic, publish your own, call and provide services, look up frames, over a WebSocket or plain HTTP. Ready made Python and JavaScript clients are served by the robot. |
| An **AI assistant** (Claude, ChatGPT, Cursor and others) | The **MCP server**, built into ARCx, under AI Integration | The assistant can see, drive and configure the robot, including everything on the Robot Bus. |

You can use several at once. A typical mixed robot might have the ROS 2 Connector bringing in a navigation stack, a Python program providing a custom planner, and an AI assistant asked to "drive to the kitchen and tell me what you see", all on the same bus.

## What ARCx adds

Joining another platform to ARCx gives that robot everything ARCx already does:

- **An interface you do not have to build.** Workspaces, live camera views, joysticks, servo sliders, Custom UI touch screens and Remote ARCx access from any browser.
- **Robot skills from the store.** Face and object tracking, speech recognition and synthesis, conversational AI, animations, navigation and many more work on the other platform's hardware once it is mapped into ARCx.
- **Scripting without a toolchain.** JavaScript, Python and Blockly run inside ARCx, with the other platform's data and commands available to them.
- **Record and replay.** The Robot Bus recorder captures what the robot sensed, from any source, and plays it back at your desk.

ARCx does not need to be installed on the other robot, and the other platform does not need to change. Both integrations talk to their platform over the network using that platform's own standard interface, so the robot keeps running the software it runs today.

# The pages in this section

- **Robot Bus**: the topics, services, frames and recorder every integration shares, and how programs in any language connect.
- **ROS 2**: what the ROS 2 Connector robot skill does, what you need on the ROS side, and what to expect.
- **Viam**: what the Viam Robot Controller robot skill does, how it connects to a Viam machine, and what to expect.

## Robot Bus

The Robot Bus is how the parts of an ARCx robot talk to each other and to the outside world. Robot skills, scripts and ARCx itself publish what they sense on **topics**, offer what they can do as **services**, and place their sensors in a tree of **frames**. Programs outside ARCx - written in Python, JavaScript, C#, Go, anything that speaks WebSocket or HTTP - join the same bus as equals: they read every topic, publish their own, call services and offer services of their own.

If you know ROS, this is the same idea as ROS topics, services and tf, without the install. If you know Viam, it is the same idea as a robot's API, without the cloud. And because the ROS Connector and Viam Robot Controller robot skills put ROS 2 and Viam robots on the bus too, ARCx is where the three meet.

---

# Contents


1. What is on the bus
2. Topics
3. Services
4. Frames
5. Recording and playback
6. The Robot Bus view
7. Scripts
8. Programs outside ARCx
9. The WebSocket protocol
10. REST and streaming
11. ROS 2 and Viam
12. AI assistants
13. Settings and security
14. Compared with ROS 2 and Viam


---

# What is on the bus

As soon as ARCx starts, everything that already flows through ARCx's core services is on the bus, with no change to any robot skill:

| Topic | What it carries |
|---|---|
| `/arcx/nms/pose` | The robot's pose from the navigation messaging system: `x`, `y` in metres from where tracking began, `yaw` in radians counter-clockwise (ROS REP 103) |
| `/arcx/nms/scan` | Each scan frame: the pose it was taken at, and parallel `angles` (radians) and `ranges` (metres) arrays, like a ROS LaserScan |
| `/arcx/nms/status` | The navigation status |
| `/arcx/nms/fix` | A GPS fix from a GPS robot skill, like ROS's |

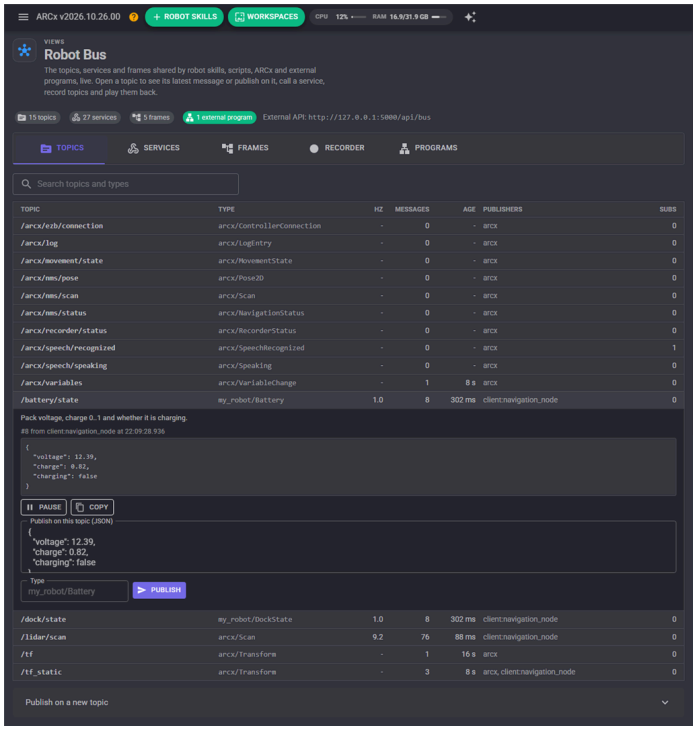
| | |
|---|---|
| | NavSatFix: latitude, longitude, altitude, fix type, satellites, HDOP, accuracy, speed and course (see Where odom is on the earth) |
| `/arcx/nms/geo_reference` | Where `odom`'s origin is on the earth and the compass bearing of its X axis (90 is ROS's east-north-up), whenever a GPS skill sets or moves it |
| `/arcx/vision/{camera}/detections` | What a camera detected in a frame: each object's name, rectangle, confidence, and - when a depth camera is aligned - its distance and position |
| `/arcx/speech/recognized` | What speech recognition heard |
| `/arcx/speech/speaking` | What the robot is saying |
| `/arcx/movement/state` | The movement direction and speeds |
| `/arcx/kinematics/{arm}/pose` | Where an arm's end effector is |
| `/arcx/variables` | Every change to a global variable |
| `/arcx/ezb/connection` | A controller connected or disconnected |
| `/arcx/log` | Warnings and errors from the ARCx log |
| `/tf`, `/tf_static` | Frames as they change |
| `/arcx/recorder/status` | What the recorder is doing |

Robot skills add their own topics and services, and so does every external program that connects.

---

# Topics

A topic is a named stream of messages, such as `/arcx/nms/pose` or `/my_robot/battery`. Anyone publishes on a topic and anyone subscribes to it; neither side needs to know the other exists.



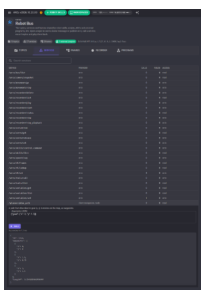
• **Names** look like file paths: they start with a slash and segments are letters, digits, `_`, `-` and `.`. Case does not matter.
• **Messages are JSON.** Every language reads JSON, and a message is turned into JSON once when it is published, however many subscribers receive it. A message also carries its topic, a **type** (free text such as `arcx/Pose2D` that tells readers which fields to expect), a sequence number, a timestamp and who published it.
• **The latest message is kept.** Any program can read a topic's current value at any time without waiting for the next message, and a new subscriber can ask for it straight away (like a latched ROS topic).
• **Patterns** subscribe to many topics at once: `*` matches one segment (`/arcx/vision/*/detections`) and `**` any number (`/arcx/**`, or `**` for everything).
• **Slow subscribers never slow the robot.** Each subscriber has its own queue; when it falls behind, its oldest messages are dropped. A subscriber can also ask for at most N messages a second per topic, and then gets the newest message of each topic at that rate - ideal for a dashboard or a phone on a slow network.

---

## Services

A service is a named call: send JSON arguments, get a JSON result (or an error). It is the bus's way of asking for something to be done. ARCx provides these:

| Service | What it does |
|---|---|
| `/arcx/bus/list` | Every topic, service, frame and connected program |
| `/arcx/movement/go`, `/arcx/movement/stop` | Drive through whichever movement panel the project has |
| `/arcx/speech/say` | Speak text |
| `/arcx/servo/get`, `/arcx/servo/set`, `/arcx/servo/release` | Read, move and release servos |
| `/arcx/script/run` | Run a JavaScript or Python script and wait for its output |
| `/arcx/variables/get`, `/list`, `/set` | Global variables |
| `/arcx/skills/list`, `/arcx/skills/control_command` | The project's robot skills and their control commands |
| `/arcx/camera/snapshot` | A JPEG from any video source |
| `/arcx/tf/lookup`, `/arcx/tf/frames`, `/arcx/tf/set` | The transform tree |
| `/arcx/recorder/...` | `start`, `stop`, `status`, `list`, `play`, `stop_playback`, `delete` |
| `/arcx/tools/list`, `/arcx/tools/call` | Every MCP tool ARCx offers, including the tools robot skills add |

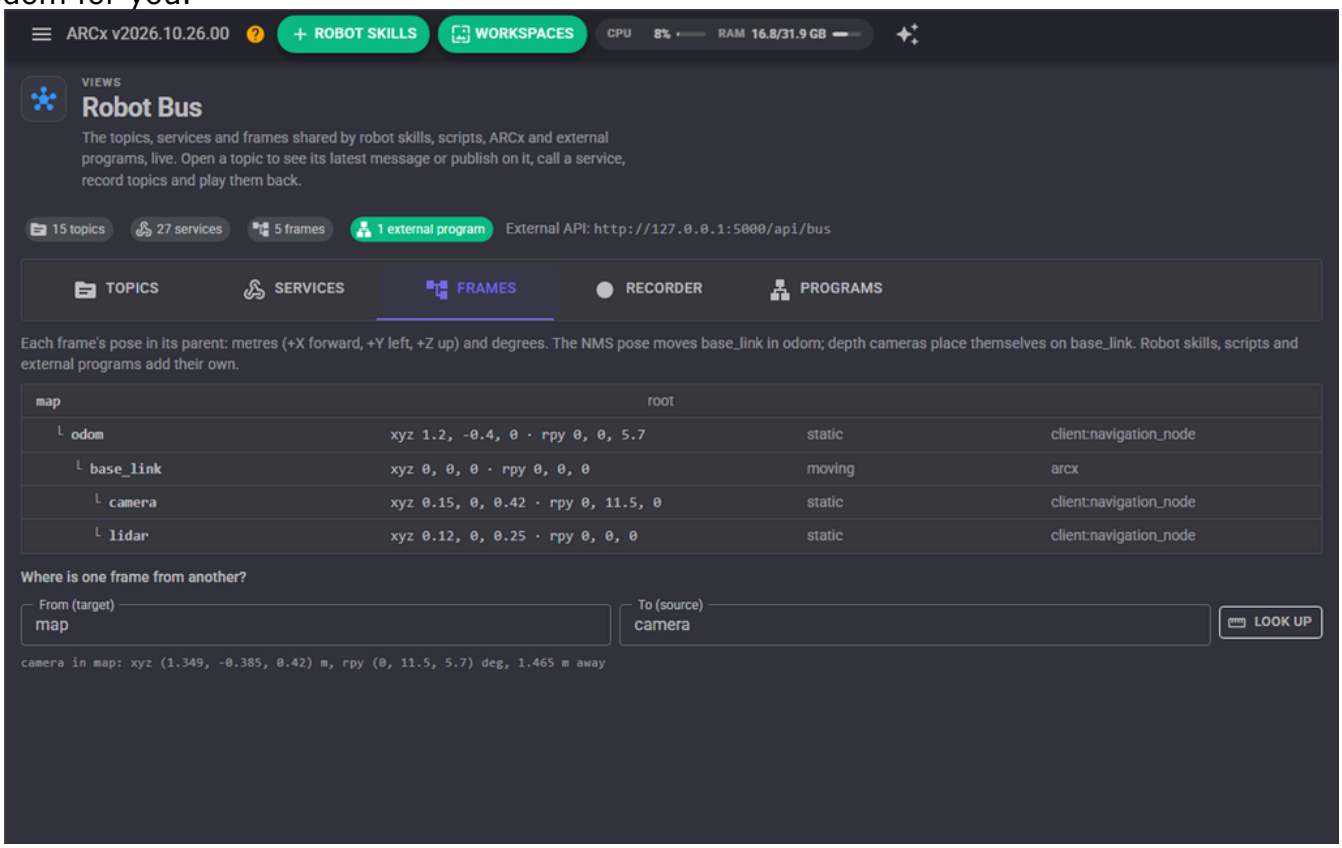


Services marked read only (lists, lookups, status) can be called by external programs even when they are not allowed to control the robot. Every service in the list has a description and, for ARCx's own, a JSON Schema of its arguments: `/arcx/bus/list` returns them.
A program outside ARCx can **provide** a service too. Python computing inverse kinematics, a planner on another computer, a ROS node behind the ROS Connector: once it advertises a service, every script, robot skill and program on the bus can call it by name.

---

# Frames

Frames say where things are. Each frame has a parent and a pose in that parent: metres and radians, +X forward, +Y left, +Z up, angles counter-clockwise (ROS REP 103, the same convention as the NMS). Lookups work between any two connected frames, however many levels apart: ask where the `camera` is from `odom`, and the bus walks camera → base_link → odom for you.




ARCx keeps two frames itself: `base_link` (the robot's body) moves in `odom` with every NMS pose, and every depth camera places itself on `base_link` from its mount settings. Scripts, robot skills and programs add their own: a camera on a pan/tilt, an arm's base, a tag on the floor.

## Where odom is on the earth

With a GPS robot skill in the project (BN-880 GPS Compass, for one), `odom` is tied to the earth. `/arcx/nms/geo_reference` says where its origin is and which compass direction its X axis points, and is published again whenever the GPS skill moves it. That happens when wheel encoders or a tracking camera report the pose and their frame drifts: the GPS skill keeps the tie following them, which is ROS's map to odom correction. Convert a pose to latitude and longitude with the newest geo reference, not one you kept.

```
{ "latitude": 43.6531968, "longitude": -79.3832052, "altitude": 112.4, "bearing": 0, "frame": "odom", "source": "Bn880 Gps Compass" }
```

`bearing` is degrees clockwise from true north: 0 means `odom`'s X points north (Y west), 90 means ROS's east-north-up (X east, Y north). The fields are `null` when the map is no longer tied to the earth.

Every fix the receiver gives arrives on `/arcx/nms/fix`, like ROS's NavSatFix. It is the receiver's own report; the robot's fused position is the pose.

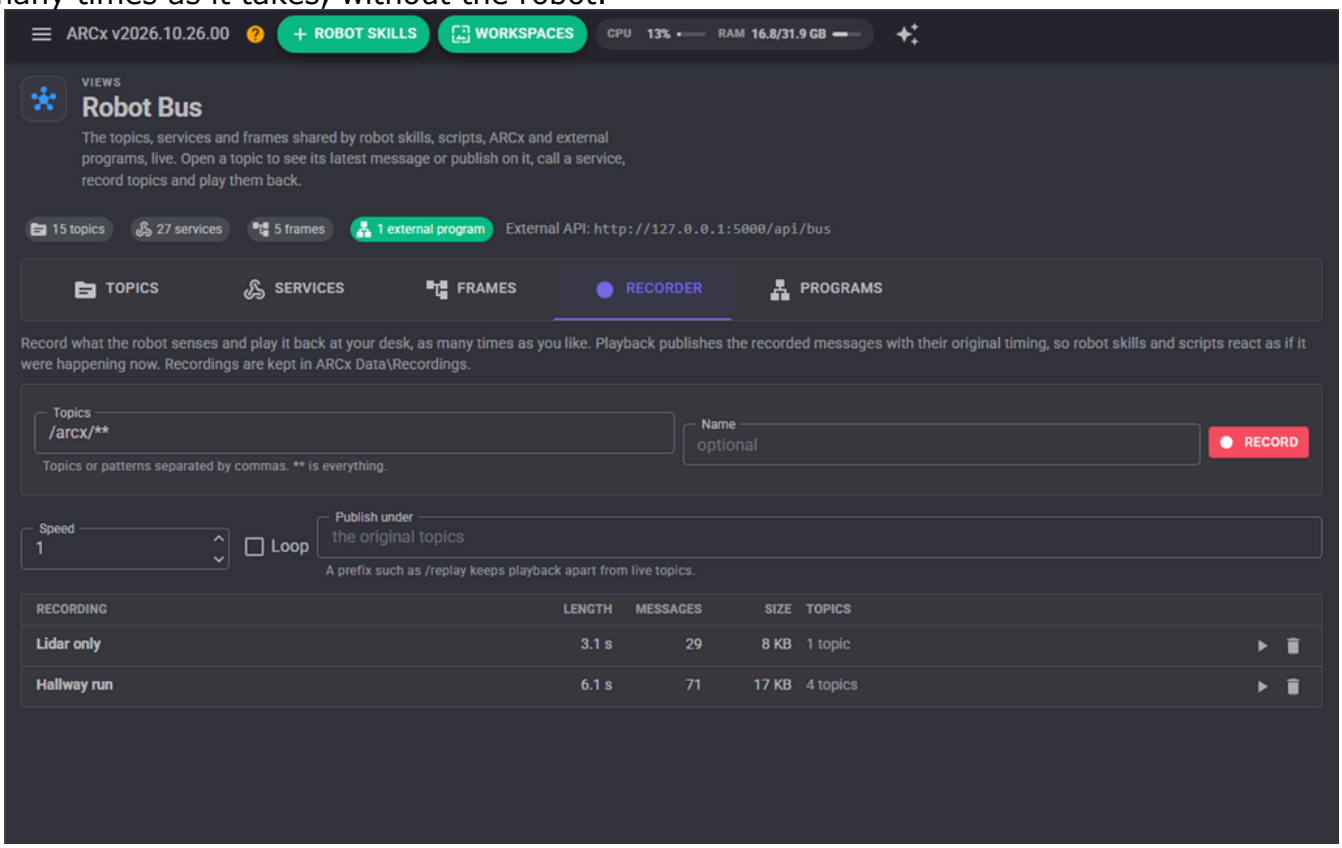
```
{ "latitude": 43.65339683, "longitude": -79.38348417, "altitude": 112.4, "fixType": "Fix3D",
"hasFix": true, "satellites": 10, "hdop": 0.93, "horizontalAccuracy": 1.5, "speed": 0.04,
"course": null, "utc": "2026-11-07T01:36:50.22Z", "source": "Bn880 Gps Compass" }
```

`fixType` is `NoFix`, `DeadReckoning`, `Fix2D`, `Fix3D`, `Differential`, `RtkFloat` or `RtkFixed`. `horizontalAccuracy` is metres, one standard deviation, estimated from the HDOP when the receiver gives no estimate of its own. `course` is degrees clockwise from true north, the direction of travel. Anything the receiver did not report is `null`.

---

## Recording and playback

The recorder writes chosen topics to a file in `ARCx Data\Recordings` and plays them back later with their original timing - the lidar, the pose, what the camera saw, what was heard. A navigation or vision problem recorded once on the robot can be replayed at your desk as many times as it takes, without the robot.




- Choose topics with patterns: `/arcx/nms/**,/arcx/vision/**`, or `**` for everything.
- Playback publishes the recorded messages on their original topics, so robot skills and scripts react as if it were happening now. Give a prefix such as `/replay` to keep playback apart from live topics instead.
- Playback runs at any speed from 0.05x to 100x and can loop.
- Service calls are not recorded: they are commands, and replaying "drive forward" at a desk is never what anyone wants.

Recordings are gzip compressed JSON lines (`.arcxbag`), so any language can read them: the first line is a header and each following line is `{"t": milliseconds, "topic", "type", "src", "seq", "data"}`.

---

## The Robot Bus view

**Views, Robot Bus** shows the bus live:

- **Topics**: every topic with its type, rate, message count, age, publishers and subscriber

count. Open one to watch its latest message, pause it, copy it or publish on it.
• **Services**: every service with its provider and call counts. Open one to read its description and argument schema, and call it with arguments you type.
• **Frames**: the transform tree, and a lookup between any two frames.
• **Recorder**: record, stop, and play or delete recordings.
• **Programs**: the external programs connected, what they subscribe to and provide, and how much they have sent and received.

---

## Scripts

JavaScript and Python scripts have a `Bus` namespace. JavaScript names start lower case, Python names upper case.

```
// Publish, read and wait
Bus.publish("/my_robot/status", { state: "patrolling", lap: 3 });

var pose = Bus.getLatest("/arcx/nms/pose");
print(pose.x + ", " + pose.y);

var heard = Bus.waitForMessage("/arcx/speech/recognized", 10000);

if (heard != null)
  print("Heard " + heard.text);

// Call a service, including one an external program provides
Bus.call("/arcx/speech/say", { text: "Hello" });
var ik = Bus.call("/planner/solve", { x: 0.2, y: 0.1, z: 0.3 }, 5000);

// Frames
Bus.setTransform("base_link", "camera", 0.1, 0, 0.3, 0, 0.2, 0, true);
var p = Bus.transformPoint("odom", "camera", 1.5, 0, 0);

// GPS: the newest fix, when a GPS robot skill is in the project
var fix = Bus.getLatest("/arcx/nms/fix");

if (fix != null && fix.hasFix)
  print(fix.latitude + ", " + fix.longitude + " ±" + fix.horizontalAccuracy + " m");
```

```
pose = Bus.GetLatest("/arcx/nms/pose")
print(pose["x"], pose["y"])

Bus.Publish("/my_robot/status", {"state": "docked"})
Bus.Call("/arcx/speech/say", {"text": "Hello"})
```

The rest: `getField(topic, "pose.yaw")`, `getAge`, `getRate`, `exists`, `getTopics(pattern)`, `getServices(pattern)`, `lookupTransform`, `getFrames`, `startRecording`, `stopRecording`, `startPlayback`, `stopPlayback`. The script editor's API list documents each one.

---

## Programs outside ARCx

Turn on **Options, Preferences, Robot Bus, Allow external programs**. A robot run without a screen can turn it on with the `ARCX_ROBOT_BUS=1` environment variable instead, and set the access token with `ARCX_ROBOT_BUS_TOKEN`.
The bus is then at `http://<address>:<port>/api/bus`, and ARCx serves ready made clients:
• **Python**: `http://<address>:<port>/sdk/arcx_bus.py` - one file, no dependencies, Python 3.8 or later. It is also a command line tool.
• **JavaScript**: `http://<address>:<port>/sdk/arcx-bus.js` - for web pages (including Custom UI pages) and Node.js 22 or later.

```
from arcx_bus import RobotBus
```

```
bus = RobotBus("http://192.168.1.20:5000", name="planner")

bus.subscribe("/arcx/nms/scan", lambda m: plan(m["data"]), max_rate_hz=5)

def solve(args):
    return {"joints": inverse_kinematics(args["x"], args["y"], args["z"])}

bus.provide("/planner/solve", solve, description="Joint angles for a target point")

bus.call("/arcx/movement/go", {"direction": "Forward", "speed": 120})
```

```
python arcx_bus.py --url http://192.168.1.20:5000 topic list
python arcx_bus.py --url http://192.168.1.20:5000 topic echo /arcx/nms/pose
python arcx_bus.py --url http://192.168.1.20:5000 topic hz /arcx/nms/scan
python arcx_bus.py --url http://192.168.1.20:5000 service call /arcx/speech/say
'{"text":"hi"}'
python arcx_bus.py --url http://192.168.1.20:5000 record start /arcx/nms/** --name hallway
```

The full client reference is `/sdk/README.md` on the robot.

---

# The WebSocket protocol

Every client is a thin layer over one WebSocket at `/api/bus/ws` (add `?token=` when a token is set and the client cannot send headers, and `?name=` to name the program). Every frame is a JSON object with an `op`. A request may carry an `id`; the reply carries the same one.

| From the program | Reply |
|---|---|
| `{"op":"hello","name":"my_node"}` | `{"op":"welcome","protocol":1,"client":"client:my_node","allowControl":true}` |
| `{"op":"subscribe","id":1,"topic":"/arcx/nms/pose","latest":true,"maxRateHz":10}` | `{"op":"ok","id":1,"subscription":"s1"}`, then messages |
| `{"op":"unsubscribe","subscription":"s1"}` | `{"op":"ok"}` when it had an id |
| `{"op":"publish","topic":"/my/topic","type":"my/Type","data":{...}}` | none unless it fails, or `ok` with `seq` when it had an id |
| `{"op":"advertise","topic":"/my/topic","type":"...","description":"..."}` | as publish |
| `{"op":"get","id":2,"topic":"/arcx/nms/pose"}` | `{"op":"result","id":2,"ok":true,"data":{message}}` |
| `{"op":"call","id":3,"service":"/arcx/speech/say","args":{"text":"hi"},"timeoutMs":10000}` | `{"op":"result","id":3,"ok":true,"data":...,"ms":12.5}` |
| `{"op":"list","id":4,"pattern":"/arcx/**"}` | `{"op":"result","id":4,"ok":true,"data":{topics,services,frames,clients}}` |
| `{"op":"advertise_service","service":"/my/add","description":"...","readOnly":true}` | `ok`; then ARCx sends `service_request`s |
| `{"op":"service_response","id":"r7","ok":true,"data":{...}}` | the answer to a `service_request` |
| `{"op":"ping","id":5}` | `{"op":"pong","id":5}` |

From ARCx: `{"op":"message","topic","type","seq","stamp","src","data"}` for each subscribed message (`stamp` is Unix time in milliseconds), `{"op":"service_request","id":"r7","service","args","caller"}` for a call to a service the program provides, and `{"op":"error","id","error"}`.

---

## REST and streaming

Everything that does not need a live connection is plain HTTP, so `curl`, a spreadsheet, a PLC or a microcontroller can take part:

| Request | What it does |
|---|---|
| `GET /api/bus` | What this is and how much is on it |
| `GET /api/bus/list?pattern=` | Topics, services, frames, programs |
| `GET /api/bus/topics/arcx/nms/pose` | The latest message on `/arcx/nms/pose` |
| `POST /api/bus/topics/my/topic?type=my/Type` | Publish the JSON body on `/my/topic` |
| `GET /api/bus/services` | The services |
| `POST /api/bus/services/arcx/speech/say` | Call `/arcx/speech/say` with the JSON body as its arguments |
| `GET /api/bus/frames`, `GET /api/bus/transform?target=odom&source=camera` | The transform tree |
| `GET /api/bus/stream?topic=/arcx/**&rate=5` | Server-Sent Events: one event per message, for `curl -N` or a browser `EventSource` |

```
curl http://127.0.0.1:5000/api/bus/topics/arcx/nms/pose
curl -X POST -d '{"text":"hello"}' http://127.0.0.1:5000/api/bus/services/arcx/speech/say
curl -N "http://127.0.0.1:5000/api/bus/stream?topic=/arcx/vision/**"
```

---

## ROS 2 and Viam

ARCx does not ask a ROS or Viam robot to change. Two robot skills bring them onto the bus:

- **ROS Connector** talks to a ROS 2 (or ROS 1) robot through rosbridge. Its Robot Bus bindings carry a ROS topic onto a bus topic or a bus topic out to ROS, and while it is connected it offers services that publish to any ROS topic, call any ROS service and list the ROS topics. A ROS navigation stack, a MoveIt planner or a perception node then works with every ARCx robot skill, script and program on the bus.
- **Viam Robot Controller** connects to a viam-server. The readings of its sensors and other components are published on the bus, and it offers services that list the Viam robot's resources and send DoCommand to any of them.

---

## AI assistants

The MCP server (see the MCP manual) has bus tools: `bus_list`, `bus_get_latest`, `bus_wait_for_message`, `bus_publish`, `bus_call_service` and `bus_lookup_transform`. An AI assistant sees exactly what robot skills and programs see, including topics and services that external programs provide.

---

## Settings and security

Under **Options, Preferences, Robot Bus**:

- **Allow external programs**: off by default. Robot skills and scripts inside ARCx use the bus whatever this says. The `ARCX_ROBOT_BUS=1` environment variable turns it on regardless.
- **Access token**: when set, programs must send `Authorization: Bearer <token>` or `?token=<token>`. Set one whenever ARCx is bound to a network address rather than the default loopback. `ARCX_ROBOT_BUS_TOKEN` sets it from the environment.
- **Allow external programs to control the robot**: off leaves programs able to subscribe, read and call read only services, but not publish, provide services or call a service that acts.

A web page from another site is refused even when a program would be allowed, so visiting a web page cannot reach your robot. The bus is never reachable through Remote ARCx: it is for programs on the robot's own network.

---

## Compared with ROS 2 and Viam

| | **ARCx Robot Bus** | **ROS 2** | **Viam** |
|---|---|---|---|
| Install | Part of ARCx on Windows, Linux (Ubuntu 24.04) and macOS | ROS 2 distribution, usually Ubuntu | viam-server plus a cloud account |
| Publish/subscribe | Topics, patterns, latest value, per-subscriber rate limits | Topics, QoS profiles | Component streams |
| Request/response | Services, provided by anyone | Services and actions | Component APIs and DoCommand |
| Transforms | Frame tree with lookups between any frames | tf2 | Frame system |
| Record and replay | Built in, any language reads the files | rosbag2 | Data capture to the cloud |
| Languages | Any: WebSocket, REST, SSE; Python and JavaScript clients served by the robot | C++, Python, others through client libraries | SDKs for Python, Go, TypeScript, C++, Flutter |
| Wire | JSON over WebSocket and HTTP | DDS | gRPC and WebRTC |
| Visual tools | The Robot Bus view, the whole ARCx interface | rqt, RViz | The Viam app |
| Works with the others | ROS Connector and Viam Robot Controller put them on the bus | rosbridge | Modules |

The bus is built for robots that are put together from parts made by different people: a robot skill from the store, a Python program, a ROS node, an AI assistant. JSON costs more bytes than DDS or protobuf; for the rates robots run at (lidars at tens of Hz, poses at hundreds) it does not matter, and every message can be read by a person.

# ROS 2

The **ROS 2 Connector** robot skill joins ARCx to a robot running ROS 2. ROS nodes can drive the robot and use its sensors, and ARCx can use what ROS publishes, so the two work as one robot. Keep your ROS navigation, perception and planning, and add ARCx's interface, robot skills, scripting and AI on top, or the other way round: build in ARCx and bring in one ROS package you need.



The full list of settings and options is in the robot skill's manual: https://synthiam.com/Support/Skills/Communication/ROS-2-Connector?id=23469

---

## Contents


1. Why use ROS 2 through ARCx
2. How it connects
3. What you need
4. What you can do with it
5. How a setup goes
6. What to expect
7. ROS and the Robot Bus


---

## Why use ROS 2 through ARCx

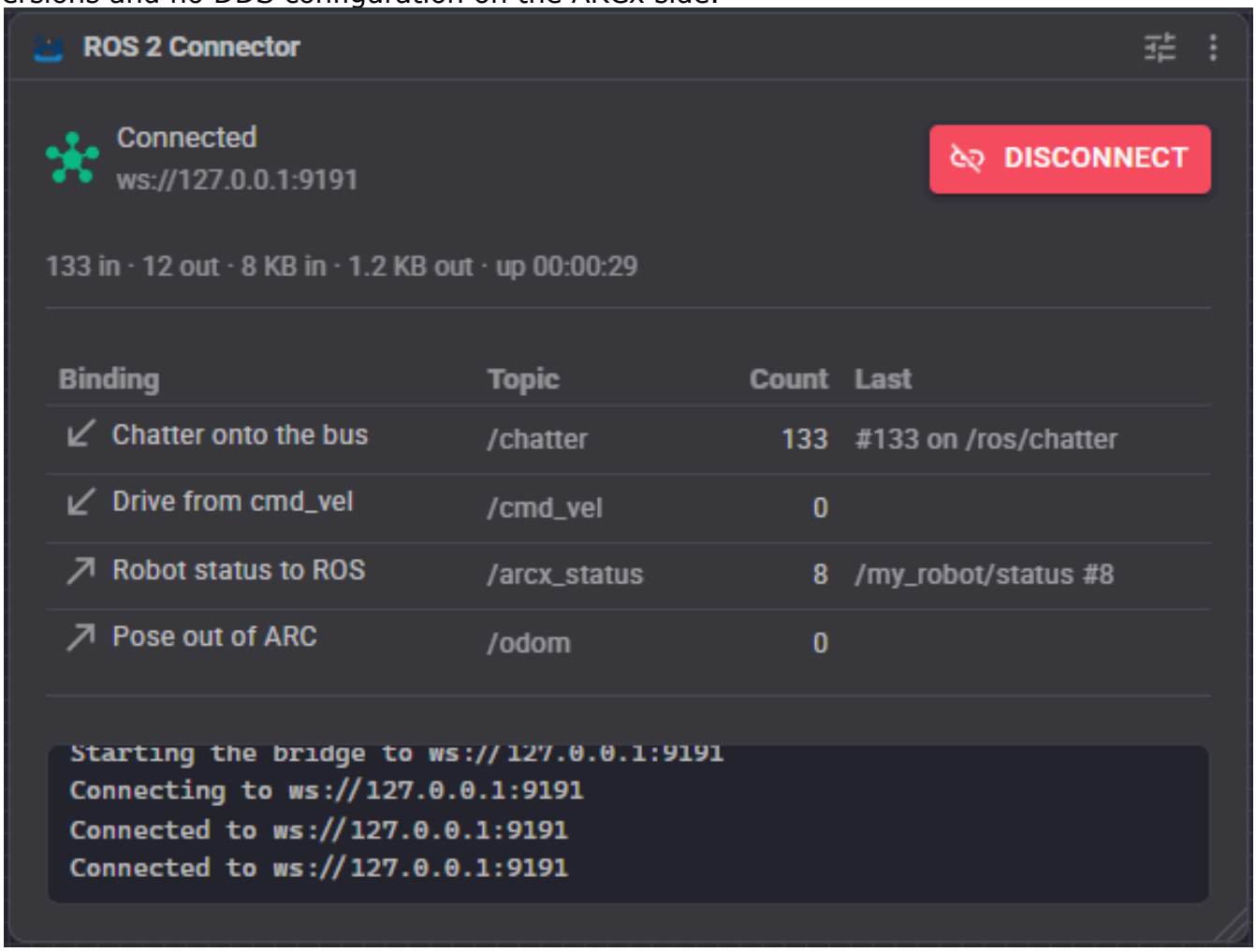
ROS 2 is powerful, but it is made for advanced users. Working with it normally means a Linux computer, the command line, writing nodes in C++ or Python, building packages, editing launch and parameter files, and keeping library versions matched across machines. Its tools, such as RViz and rqt, are built for developers to inspect and debug a robot, not for someone to operate one. There is no point-and-click way to make a ROS robot speak, follow a face, run an animation or show a control screen on a tablet; each of those is more code to write.

That puts a ROS robot out of reach for most people who would use it: students, teachers, artists, researchers outside robotics, operators and customers.

ARCx changes that. Once the ROS 2 Connector joins the two, the ROS robot is driven from ARCx's interface: joysticks, camera views, servo sliders, touch screens built with Custom UI, and access from any browser. Behaviours come from robot skills added from the store, and scripts are written in Blockly, JavaScript or Python inside ARCx. The ROS side keeps running exactly as it does today, so the engineers who built it keep their tools, and everyone else gets a robot they can actually use.

## How it connects

The skill talks to ROS through **rosbridge**, the standard ROS package that lets software outside ROS take part over a WebSocket. That has one big advantage: **ARCx does not need ROS installed.** ARCx can run on a Windows laptop, a Mac or a Linux computer and reach a ROS 2 robot across the network, with no ROS distribution, no matching library versions and no DDS configuration on the ARCx side.




## What you need

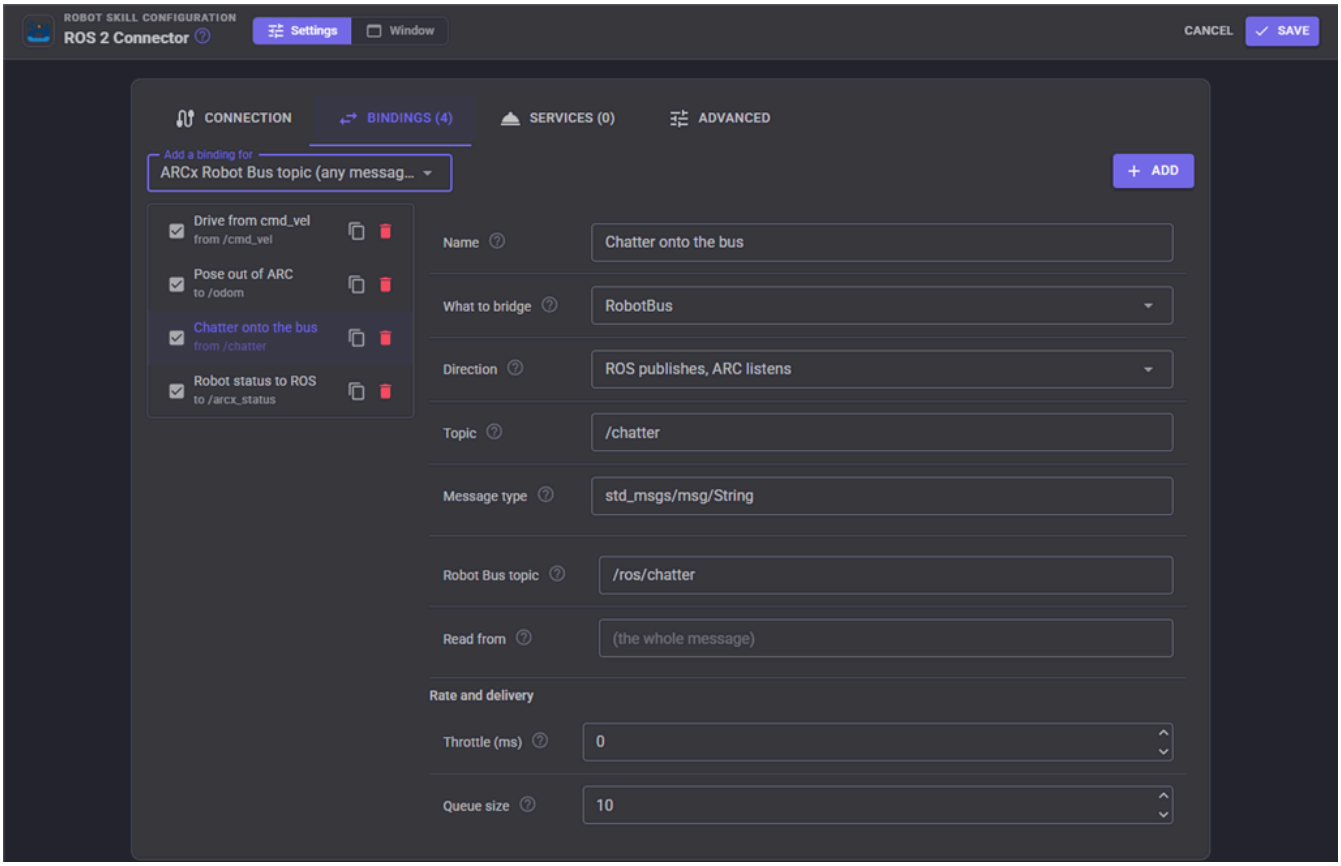
- A ROS 2 machine (the robot, or a computer on its network) running `rosbridge_server` from the `rosbridge_suite` package. One command starts it: `ros2 launch rosbridge_server rosbridge_websocket_launch.xml`. It listens on port 9090 by default.
- A network path between that machine and the computer running ARCx.
- The ROS 2 Connector robot skill added to your ARCx project, pointed at the rosbridge address.

ROS 1 robots running rosbridge work too, with a compatibility switch in the skill.

## What you can do with it

The skill is built around **bindings**. Each binding wires one part of ARCx to one ROS topic, in one direction, and you add as many as you need. Common ones come ready to use:

• **Let ROS drive the robot.** A movement binding on `/cmd_vel` lets a ROS 2 navigation stack, a teleop node or your own controller drive whatever movement panel the ARCx project has, from a hobby servo robot to a motor controller.
• **Share the lidar and the pose.** A ROS lidar can feed ARCx's mapping and navigation robot skills, and ARCx's own lidar and odometry can be published to ROS in standard message types.
• **Move servos from ROS.** Joint states drive ARCx servos, and ARCx servo positions can be published back.
• **Send the camera to ROS.** ARCx video sources can be published as compressed images.
• **Trigger anything in ARCx from a message.** A ROS message can run a script, run a control command on any robot skill, or set an ARCx variable, so a ROS node can make the robot speak, play an animation or start tracking.
• **Answer ROS service calls.** ARCx can offer ROS services that nodes call and wait on, such as a behaviour tree node asking the robot to say something and waiting until it has.
• **Any message type, including your own.** Custom bindings build messages from a template or pick incoming messages apart, so you are not limited to the types the skill knows.




## How a setup goes

1. Start rosbridge on the ROS machine.
2. Add the ROS 2 Connector robot skill and enter the rosbridge host and port. Press **Connect**: the skill's window shows the link and counts every message in and out.
3. Add the bindings you need. Each starts with sensible defaults (a movement binding already points at `/cmd_vel`, for example), so most need only a topic name checked.
4. Watch the counts move in the skill's window, and turn on **Auto connect** so the robot comes up connected by itself.

The skill can also list the ROS graph's topics, services and nodes, which helps when you are not sure what a robot publishes.

## What to expect

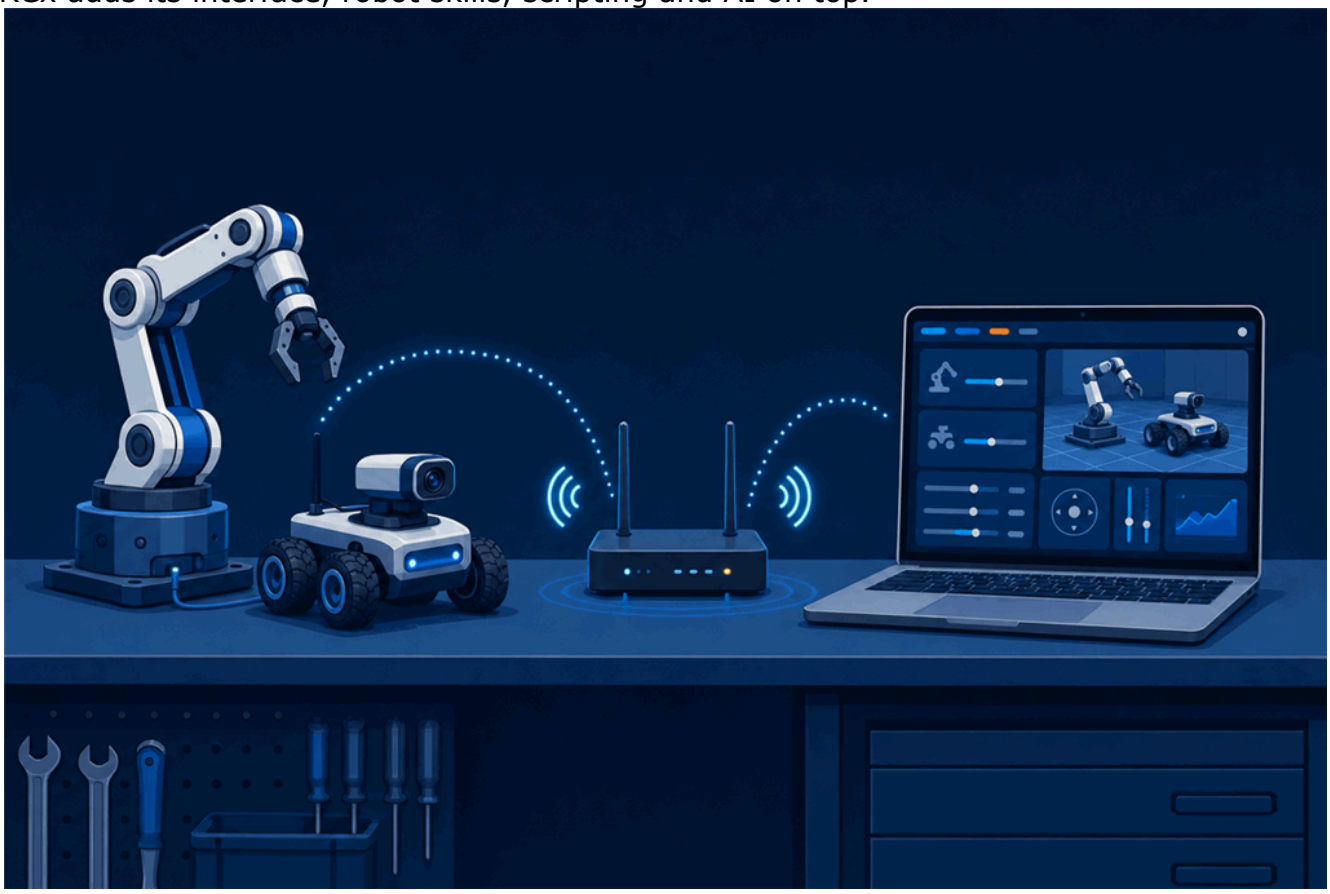
• **It is a bridge, not a ROS replacement.** ROS keeps doing what ROS does well; the skill joins it to ARCx. You still write and run ROS nodes the normal ROS way.
• **Everything travels as JSON.** rosbridge carries messages as JSON over a WebSocket, which is simple and works across any network, but costs more than native DDS. Robot control, poses, lidar scans and compressed camera images at normal rates are comfortable. Very heavy streams, such as raw point clouds or uncompressed high resolution video at full frame rate, are better kept inside ROS.
• **Safe by default.** If the connection to ROS drops, the skill can stop the robot's movement, so a robot driven from ROS does not keep going when ROS goes away. It reconnects by itself when the link returns.
• **Numbers that JSON cannot carry.** JSON has no infinity or NaN, so a laser scan reading with no return is sent as just beyond the maximum range instead. The skill's manual explains where this matters.

## ROS and the Robot Bus

The ROS 2 Connector also puts ROS on ARCx's **Robot Bus**. A Robot Bus binding carries a ROS topic onto the bus whole, in its own message shape, or sends a bus topic out to ROS. While connected, the skill also offers bus services that publish to any ROS topic, call any ROS service, mirror a ROS topic onto the bus on request, and list the ROS graph.
That means a Python program, a JavaScript dashboard or an AI assistant connected to the same ARCx can reach your ROS graph without knowing ROS is there, and ROS data can be recorded and replayed with the Robot Bus recorder like any other.
For every binding kind, setting and script command, see the ROS 2 Connector manual: https://synthiam.com/Support/Skills/Communication/ROS-2-Connector?id=23469

## Viam

The **Viam Robot Controller** robot skill makes ARCx the interface for a machine running Viam. It connects to `viam-server` on the robot, finds every component and service the machine is configured with, and puts each one into the part of ARCx that already knows what to do with it. Your Viam machine keeps its configuration, its modules and its drivers; ARCx adds its interface, robot skills, scripting and AI on top.



The full list of settings and options is in the robot skill's manual: https://synthiam.com/Support/Skills/Communication/Viam-Robot-Controller?id=23598

## Contents


1. Why use Viam through ARCx
2. How it connects
3. What you need
4. What the machine becomes in ARCx
5. How a setup goes
6. What to expect
7. Viam and the Robot Bus


## Why use Viam through ARCx

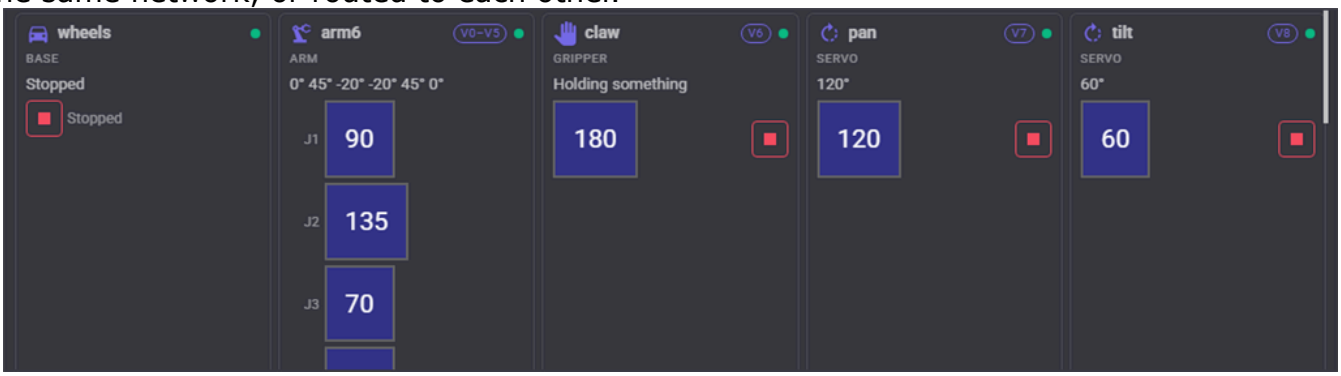
Viam is made for advanced users. A machine is described in JSON configuration, and making it do anything useful normally means writing a program with one of Viam's SDKs in Python, Go, TypeScript or C++. The control panel in Viam's app is built for engineers to test components one at a time, not for someone to operate a robot day to day. There is no point-and-click way to make a Viam robot follow a face, respond to speech, run an animation or show its own control screen on a tablet; each of those is more code to write. That puts a Viam machine out of reach for most people who would use it: students, teachers, artists, operators and customers.

ARCx changes that. Once the Viam Robot Controller connects, the machine is driven from ARCx's interface: joysticks, camera views, servo sliders, touch screens built with Custom UI, and access from any browser. Behaviours come from robot skills added from the store, and scripts are written in Blockly, JavaScript or Python inside ARCx. The machine keeps its Viam configuration exactly as it is, so the engineers who built it keep their tools, and everyone else gets a robot they can actually use.

## How it connects

The skill speaks Viam's own API, the same gRPC interface Viam's SDKs use, directly to `viam-server` on the robot. Nothing is installed on the robot and no driver is written: if the machine works in Viam, its components are available to ARCx.

The connection is **direct, over your network**, from the ARCx computer to the robot's address and port. It does not go through Viam's cloud, so ARCx and the robot need to be on the same network, or routed to each other.




## What you need

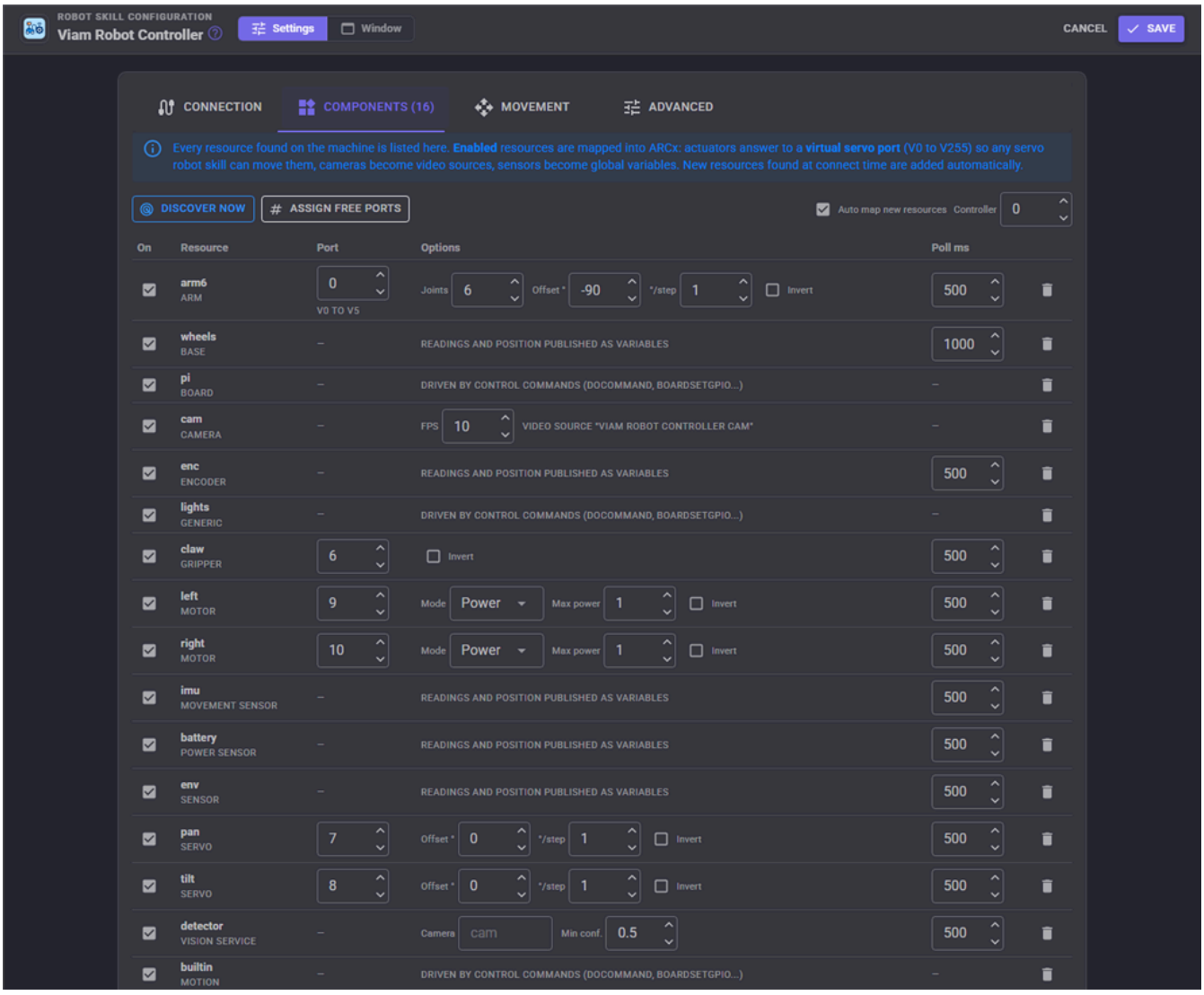
- A machine running `viam-server`, reachable from the ARCx computer. Its default port is 8080.
- For a machine set up through app.viam.com: an **API key** for the machine, created in the machine's settings. A `viam-server` started from a local configuration without authentication needs no key.
- The Viam Robot Controller robot skill added to your ARCx project, with the machine's

address and key.

## What the machine becomes in ARCx

When the skill connects, it lists every resource on the machine and maps each one:

| On the Viam machine | In ARCx |
|---|---|
| A base | The project's movement panel, so joysticks, movement scripts, tracking skills and navigation drive it |
| Servos, motors, grippers, arms | Servo ports, so any servo robot skill, animation or script moves them |
| Cameras | ARCx video sources, so face, colour, glyph and object tracking work on them |
| Vision services | Their detections drawn on the camera and handed to the tracking skills |
| Sensors, movement sensors, power sensors, encoders | Variables that scripts and robot skills read |
| Boards and anything else | Commands, including DoCommand to reach any resource's own commands |




New resources found on the machine are added automatically, and you choose which are enabled and which servo port each actuator answers to.

## How a setup goes

1. Make sure `viam-server` is running on the robot and note its address and API key.
2. Add the Viam Robot Controller robot skill, enter the address and key, and use **Test connection** to confirm ARCx can reach the machine and see what it found.
3. Connect. The skill's window shows one live card per resource: a servo's angle, a motor's power, an arm's joints, a camera's picture, a sensor's readings.
4. Choose what the movement panel drives (a Viam base, or a pair of motors), and from then on every ARCx way of driving the robot works.

## What to expect

- **The machine stays the source of truth.** Configuration, modules and drivers stay in Viam. ARCx reads what the machine offers and sends it commands; change the machine in Viam, press **Discover**, and ARCx follows.
- **Safety sessions.** The skill can open a Viam safety session, so if ARCx or the network drops, `viam-server` stops every actuator ARCx was commanding. A **Stop all** button stops everything on the machine at once.
- **Network speed matters for cameras.** Each camera frame is a call over the network. A handful of frames per second is plenty for tracking, and the rate is set per camera.
- **Local network, not the cloud.** Reaching a machine only through Viam's cloud connection is not supported; ARCx needs a network route to the machine's address.

## Viam and the Robot Bus

Everything the skill reads from the machine is also published on ARCx's **Robot Bus**, whole and structured: each resource's readings on its own topic, and the machine's health on a status topic. While connected, the skill also offers bus services that list the machine's resources and send DoCommand to any of them.

That means a Python program, a JavaScript dashboard or an AI assistant connected to the same ARCx can use the Viam machine without a Viam SDK, and its sensor data can be recorded and replayed with the Robot Bus recorder like any other.

For every setting, option and control command, see the Viam Robot Controller manual: https://synthiam.com/Support/Skills/Communication/Viam-Robot-Controller?id=23598