

ATC CHEATSHEET

The everyday commands for shipping and operating software with atc. Print it any time with `atc cheatsheet` ; the latest version lives at <https://atc.dev/cheatsheet>.

Most commands take an SSH **target** in `[user@]host[:port]` form, for example `robot@automata0.local`. Add `-v` for verbose logs, or `--json` for machine-readable output.

CHECK WHAT'S RUNNING

```
atc ota status robot@device          # what's active, and daemon
    health
atc ota status robot@device --check   # also run each check hook
atc ota status robot@device --all-channels # each
    channel's delivery with its artifact versions
atc ota ls robot@device               # deliveries and artifacts
atc ota ls robot@device -l            # expand each delivery
atc ota df robot@device               # disk use per channel
atc ota netcheck robot@device         # can atc reach this device?
```

Versions of the tools themselves:

```
atc --version                        # your atc
atc version --json                   # the same values, for a
    script (after 0.1.4)
atcd --version                        # the daemon (run on the
    device)
```

On the device, with no SSH:

```
atcd ctl status                      # daemon serving? (0 = yes, 3
    = down, 5 = up, not answering)
atcd ctl ls                          # installed versions
```

WHAT IS THIS DEVICE'S ID?

```
cat /var/atc/device_id           # the id the server knows this
                                device by
```

Read that file, not `enrollment.json`. The two agree: once the device enrolls, `device_id` holds the id the server minted, and it is mode `0644` so you can read it without root. Before the device enrolls it holds a locally minted UUID instead, which the enrolment then replaces. `enrollment.json` is mode `0600` and holds the device's bearer credential; do not copy it anywhere.

START / STOP / RESTART THE DAEMON

systemd owns the daemon's lifecycle, not atc. The unit is `atcd.service` :

```
sudo systemctl start atcd
sudo systemctl stop atcd
sudo systemctl restart atcd      # after an atc.toml change
systemctl status atcd           # state, plus a log tail
journalctl -u atcd -b -n 100 --no-pager
```

No `atcd.service` on the device? The unit name comes from `[daemon] service_name` in `atc.toml`. Find the real one with `systemctl list-units 'atc*' .`

`ssh robot@device systemctl is-active atcd` can answer `Failed to connect to bus .` A device image with no systemd session support answers that way whatever the daemon does, so the reply says nothing about `atcd`. Ask the daemon itself:

```
atc ota status robot@device      # from your machine
atcd ctl status                  # on the device (0 = up, 3 =
                                down, 5 = not answering)
```

BUILD & PUSH SOFTWARE

Stage an image in your local cache, then push it to a device:

```

atc oci build myapp:1.2.3 -f Dockerfile --platform linux/arm64
atc oci import ghcr.io/team/app:v1      # or pull an existing image
atc oci import myapp:dev --docker        # or take one from local
                                         Docker
atc oci ls                               # what's in the cache

atc ota push robot@device                # push every declared artifact
atc ota push --switch robot@device       # push, then activate
atc ota push -o myapp robot@device       # push one artifact only
atc ota push --dry-run robot@device      # preview, upload nothing

```

Artifacts and versions are declared in a manifest, `atc.toml` or `atc.kdl`. Scaffold an `atc.toml` with `atc ota init`. It carries the lifecycle-hook and long-running-service blocks as comments. `atc ota init --full` also writes the compiled-binary block and the long form of the writable-state rule. Add `--format kdl` to write an `atc.kdl` instead.

`atc ota init` refuses to overwrite a manifest (ATC-E0424). To write the file again, for example with a new `--name`, add `--force`. It replaces a manifest of the same format only.

A bundle's `source` ships every file and every resolvable symlink under it, subdirectories included, at any depth. A dangling symlink stops the push with ATC-E0510. Reorganizing inside it needs no `atc.toml` change: the next push carries whatever tree it finds.

Staging into a directory named `dist`? Anchor the exclude (`rsync --exclude=/dist`), or an unanchored one also drops every nested `dist/` under `node_modules`. Details, plus how a shipped binary should find its sibling data directories: [docs/operating/atc-toml.md](#).

`atc ota push` sends software to a device. `atc oci push` publishes an image to a container registry, which is a different destination.

SWITCH AND ROLL BACK VERSIONS

```

atc ota ls robot@device -d              # versions to choose from
atc ota switch robot@device v1.4.2     # activate one
atc ota rm robot@device v1.4.2         # remove one

```

A rollback restores the earlier delivery. A first delivery has no earlier one, so a veto on it removes `delivery/current` instead, and the push footer says “no earlier delivery to restore”. Units that start from that pointer fail until the next successful push.

A device activates a whole delivery at a time. To switch a single artifact, open that hatch first. Prefer `--for`, so it closes again on its own:

```
atc ota assignment-mode robot@device per_artifact --for 1h
atc ota switch robot@device v1.4.2 -a myapp
atc ota assignment-mode robot@device delivery_pinned # the default
```

atcd keeps its device settings in its store. Read and change them over attach. A change takes effect at atcd’s next restart:

```
atc ota settings ls robot@device # every key,
    its source
atc ota settings set robot@device storage.version_keep_n=4
atc ota settings unset robot@device storage.version_keep_n
```

A `server_only` device refuses `set` and `unset` over attach. Run `atcd settings` on the device as the `atcd` user instead.

The keys that belong to the `systemd` unit, such as `daemon.run_as_user`, change only through a re-install. An install writes only the settings you state: its flags, each `--set`, and the keys the repo’s `atc.toml` states. Every other setting on the device stays:

```
atc bootstrap atcd install robot@device --set
    daemon.hook_run_as_user=robot-hooks
atc bootstrap atcd install robot@device --set
    daemon.service_name=atcd-edge # a unit not named atcd
```

A DEVICE THAT REFUSES EVERY PUSH

A push that crashed part-way through a peripheral’s flash is flagged for recovery, and the device then refuses every later push of that artifact. Nothing local says so. `atc ota recover` reports each such push, the peripherals it holds, and what each is owed, then discharges the ones the daemon can act on:

```
atc ota recover robot@device      # report, then recover what is
    automatic
atc ota recover robot@device --json  # the same report, machine-
    readable
```

Two cases have no safe automatic answer: a journal that cannot say what a committed delivery parked, and a peripheral whose physical state nobody has established. --accept-unknown-state settles those, and only those — it records each as failed, releases its locks, and writes an audit row. A push that owes a real act is refused by name:

```
atc ota recover robot@device --accept-unknown-state
```

UPDATE ATC AND ATCD

```
atc upgrade --check                # is a newer atc available?
atc upgrade                        # update atc on your track
atc upgrade --track nightly        # switch tracks
atc bootstrap source ls            # where updates come from

atc bootstrap atcd install robot@device  # install on a new device
atc bootstrap atcd install robot@device --send-reports  # and send
    fault reports to Agency
atc bootstrap atcd install robot@device --no-link  # a second store:
    leave /usr/local/bin/atcd alone
atc bootstrap atcd upgrade robot@device  # update, then restart it
atc bootstrap atcd upgrade robot@device --from ./atcd  # copy this
    atcd; the device downloads nothing
```

An interactive install asks once whether `atcd` may send the redacted diagnostic report it writes on a fault to Agency. The answer defaults to no. --yes and batch runs do not ask, and change nothing. The answer is the device setting `diagnostics.auto_send` in `atcd`'s store. `atcd` reads it only when it starts. To stop sending, set it to `false` and then restart `atcd`:

```
atc ota settings set robot@device diagnostics.auto_send=false
ssh robot@device sudo systemctl restart atcd
```

Until that restart, the running `atcd` can still send reports.

`atc` also checks once a day whether a newer `atc` exists, and prints one line when there is. `ATC_NO_UPDATE_CHECK=1` turns that off. The check reads the release index for your own machine. It is skipped when stdout is not a terminal, so a script never sees it, and it touches no device.

Two different things get called an update check. A push, or even a status call, can install `atcd` on a device that carries none. That is the auto-bootstrap, and `ATC_NO_UPDATE_CHECK` does not govern it. Set `auto = false` under `[bootstrap]` in `atc.toml` to require an explicit `atc bootstrap atcd install` instead. No `atc` command updates a device's `atcd` on its own; `atc bootstrap atcd upgrade` is the command that does.

KEEP A BUNDLE RUNNING AS A SERVICE

Read this section only for a bundle that must keep a process running. A bundle needs no lifecycle hook. A binary, a script, or a data set ships without one. Such a bundle still needs activation, which is a separate step. `atc ota push --switch` makes the new version current. A push without the flag stages only, unless `[config] autoswitch = true` in `atc.toml` activates it for you.

`atc` ships files and starts no process. A `systemd` unit does that, and three lifecycle hooks in `atc.toml` drive it:

```
[bundle.lifecycle.hook.install.exit]      # write the unit, before the
flip
script = "@self/scripts/install_unit.sh"

[bundle.lifecycle.hook.activate.exit]     # start it, right after the
flip
script = "@self/scripts/start_service.sh"

[bundle.lifecycle.hook.deactivate.enter]  # stop the outgoing version
script = "@self/scripts/stop_service.sh"
```

Hooks run as `atc-hooks`, and bootstrap enables `linger` for that user, so `systemctl --user` works with no `sudo` and the unit returns after a reboot.

A device with no `dbus` or no `systemd --user` cannot use this recipe. `atc` refuses `hook_run_as_user = "root"` under a non-root daemon (`ATC-E0506`). For the two

supported paths, see `docs/operating/atc-toml.md` , “Root work on a device with no user session”.

Point `ExecStart` at the pointer `atc` flips on every push. The install hook reads it out of the one variable every hook receives:

```
# delivery/current/<name> – the pointer, in a form /bin/sh accepts
set -eu
# Print one dotted field of ATC_HOOK_CONTEXT, or the second argument
  when
# it or an object above it is null. With no second argument, a null
  field
# exits 1 and a field under a null object prints "". A missing key
  exits 1.
ctx() {
  awk -v path="$1" -v nargs="$#" -v dflt="{2-}" '
    function fail(why) { print "ATC_HOOK_CONTEXT." path ": " why > "/
      dev/stderr"; exit 1 }
    function find(t, k,  j, c, d, q) {
      k = "\"" k "\""
      for (j = 1; j <= length(t); j++) {
        c = substr(t, j, 1)
        if (q) { if (c == "\\") j++; else if (c == "\\") q = 0 }
        else if (c == "{") d++
        else if (c == "}") { if (--d == 0) return 0 }
        else if (c == "\\") { if (d == 1 && substr(t, j, length(k))
          == k) return j + length(k); q = 1 }
      }
      return 0
    }
    BEGIN {
      s = ENVIRON["ATC_HOOK_CONTEXT"]; n = split(path, key, ".")
      for (i = 1; i <= n; i++) {
        if (substr(s, 1, 4) == "null") { print dflt; exit 0 }
        if (substr(s, 1, 1) != "{") fail("not an object")
        p = find(s, key[i])
        if (p == 0) fail("no such field")
        s = substr(s, p)
      }
      if (substr(s, 1, 4) == "null") { if (nargs < 2) fail("null");
        print dflt; exit 0 }
      if (substr(s, 1, 1) != "\\") fail("not a string")
      for (j = 2; j <= length(s); j++) {
        c = substr(s, j, 1)
        if (c == "\\") { print v; exit 0 }
```

```

        if (c == "\\") { j++; c = substr(s, j, 1); if (c != "\"" &&
        c != "\\\" && c != "/") fail("holds a control escape") }
        v = v c
    }
    fail("not a string")
}'
}
root=$(ctx current_root)
[ -n "$root" ] || { echo "install_unit.sh: current_root is null" >&2;
    exit 1; }
# ...then write ExecStart="$root/bin/myapp" into the unit. Quote it,
double
# any `%` (systemd reads `%n` as a specifier), and refuse a path
holding
# anything outside [A-Za-z0-9/._+:@% -]. A `$` needs no escaping in
an
# executable path. Full guard: docs/operating/atc-toml.md.

```

Read the context with `awk`, not `jq` or `python3`. The meta-atc device image carries neither, and `atcd` pins the hook `PATH` to the standard system directories. The `set -eu` line and the emptiness test are part of the recipe. Without them a missing reader gives an empty `$root`, the hook writes a broken `ExecStart`, and the push still reports the hook as successful.

Use the field. Do not compute the path from the hook's working directory, which is `artifacts/<name>/versions/<sha>`, a sibling of `delivery/` and not a parent. Write the value into the unit at `install.exit`, even before the pointer exists. Do not read files through it there. Until the flip it names the OUTGOING version. To inspect the version being installed, read `self_root.new`, or the `@self` hook's own directory, which is the incoming tree at `install.exit`.

Eight rules, in the order they bite:

- Put `ConditionPathExists=` on the launcher path in the unit's `[Unit]` section, with the same `%`-doubled value you wrote into `ExecStart`. A vetoed first delivery removes `delivery/current`, and a `Restart=always` unit then restart-loops on a missing file. The condition makes it stay inactive until the next successful push.
- Start the service at `activate.exit`, never at `activate.enter`.
`activate.enter` runs before the outgoing version's `deactivate.*`, so a start there is stopped again.
- Use `restart`, not `enable --now`. `enable --now` exits 0 against a unit the old version left running, and the old process keeps the name.

- A hook binds to the version that ships it, script and declaration both. So `deactivate.*` runs the OUTGOING version's copy, and an edit here reaches the deliveries after this one.
- End a hook that only reports with `exit 0`. The exit status is the last command's, and a non-zero one vetoes the delivery.
- Give a wrapper that sources ROS or colcon a `#!/bin/bash` shebang, not `#!/bin/sh`. A device's `/bin/sh` is `dash`, and `setup.bash` reads `$BASH_SOURCE` and uses bash-only syntax. Source `setup.bash`, not `setup.sh`. The POSIX form instead carries the build machine's prefix, which the robot does not have.
- Do not `set -u` in a wrapper that sources ROS's `setup.bash` or a `venv activate`. Both read variables they never set.
- Build a ROS 2 install space you ship without `--symlink-install`. With it, an `ament_python` package gets a `.egg-link` into `build/` on your machine and no `.py` files, so the import fails on the robot.
`atc ota push` warns about it.

Does a hook run before the `install ( check.* , download.* , install.enter )`? Ship its script in the artifact and name it `@self/<path>`. There, `@self` runs the ACTIVE version's copy and skips the hook when that copy is missing, which covers the first delivery. See **Which version's script @self runs**.

Does the hook need a fixed system path that the artifact installs, as a `.deb` does? Add `allow_missing_initial = true` to that hook. The first delivery then skips the hook while the script does not exist yet. Use `allow_missing = true` to skip a missing script on every delivery. See **A hook script that the artifact installs**.

CHECK THE SERVICE AFTER A PUSH

The unit belongs to the `atc-hooks` user. Your login user has a different user manager. So `systemctl --user status myapp.service` over `ssh` answers `Unit myapp.service could not be found`. That message reports the wrong identity, not a failed push. Ask with the hook identity instead:

```
atc shell robot@device -- systemctl --user status myapp.service
```

`atc shell` sets `XDG_RUNTIME_DIR` and `DBUS_SESSION_BUS_ADDRESS` for the hook user. Plain `sudo -u atc-hooks systemctl --user ...` sets neither, and answers `Failed to connect to bus`. Set the variable yourself where `atc shell` is not available:

```
sudo -u atc-hooks env XDG_RUNTIME_DIR="/run/user/$(id -u atc-hooks)" \
systemctl --user status myapp.service
```

Logs need a different command. `journalctl --user as atc-hooks` often reports `No journal files were found`, because the device keeps no separate user journal. The records are in the system journal. Read them from a sudo-capable login:

```
sudo journalctl _SYSTEMD_USER_UNIT=myapp.service _UID="$(id -u atc-hooks)"
```

A `roslaunch` service writes node output to `~/ros/log/latest/`, not to the journal. A node the unit starts directly splits its output: `print()` reaches `stdout`, and `rospy.loginfo` reaches `~/ros/log/<run_id>/`. Python buffers `stdout` on a pipe, so set `Environment=PYTHONUNBUFFERED=1` in the unit to get the first half into the journal. Read the second half as the hook user:

```
atc shell robot@device -- sh -c 'ls -t ~/ros/log/'
```

Full scripts, the `ATC_HOOK_CONTEXT` schema, and the rootless- `systemd` prerequisites: [docs/operating/atc-toml.md](#), “Worked example: a long-running service”. For a root-owned system unit and a `.deb`, see [examples/deb-systemd-service/](#).

TEST YOUR HOOKS WITH NO DEVICE

```
atc bootstrap atcd fetch          # once: get an atcd for this
    machine
atc ota push --local --switch      # push and activate: hooks run
atc ota push --local --no-switch  # stage only
atc ota push --local --switch --local-root /tmp/hook-test
atc ota switch --local-root /tmp/hook-test v2 # activate a staged
    version
```

Run `atc bootstrap atcd fetch` once first. The installer places only `atc`, and `--local` runs a real `atcd`, so the push otherwise stops on `ATC-E0463`. The fetch needs no device, and it tells you if your `PATH` cannot reach the result. Released `atcd` assets are Linux `x86_64` and `aarch64` only.

`--local` pushes to a throwaway daemon on this machine. No target, no SSH, no root. Pass `--switch` or `--no-switch`; the command will not guess. Give `--local-root` an absolute path, or the hooks fail to start. The hooks run as **you**, here, so this is not a sandbox.

To see what you staged, list it. Do not `cat` it:

```
ls -la dist/                # what is in the source
    directory
file dist/myapp             # binary, wrapper script, or
    script
```

A `source` directory usually holds a compiled binary. `cat` on one writes the raw bytes to the terminal, which can leave it unreadable until you run `reset`. `file` answers the question `cat` was asked, and it also catches a libtool wrapper script staged in place of the real binary.

GET A SHELL ON A DEVICE

```
atc shell robot@device      # a shell as the atc-hooks
    user
atc shell robot@device -a myapp # start in that version's dir
atc shell robot@device -- systemctl --user status myapp
```

You get the identity and environment your lifecycle hooks run in, so manual steps behave like the automated ones. A trailing command runs alone and forwards its exit code. This is a shell on the device, not in a container. To run a cached image here, use `atc oci run <ref>`.

`atc shell` is the identity-correct way in. Prefer it over raw `ssh`. The daemon user owns the delivery tree, and the hook user joins its group to traverse it. A login user outside that group can walk the tree by path but cannot list it.

VERIFY A DELIVERY BY HAND

`atc ota status` and `atc ota ls -l` each print an **On-device paths** block with the resolved path of every active artifact:

```
<store-root>/channel/<ns>/<name>/delivery/current/<artifact>  
/var/atc/channel/local/default/delivery/current/myapp
```

Read the two segments after `channel/` as the channel: a namespace and a name. They are `local/default` unless `atc.toml` declares a `[channel]`. The artifact name is the last segment, and the bundle's own files sit inside it.

`atc ota status` prints the live store root, which it reads off the daemon.

`atc ota ls -l` learns no root from the daemon. Over SSH it resolves `atcd`'s default, `/var/atc`. Over `--socket` it prints `<store-root>`, because the daemon's root was chosen by whoever started it. Run `atc ota status <target>` to read the live root of a daemon that runs with `--store-root`.

The block prints paths only. To list what a path holds, follow the symlinks with `find -L <path> -type f`, as the facts below explain.

A bundle that runs and exits needs no path arithmetic. `atc shell -a <name>` lands in the ACTIVE version's directory, with the hook user's identity. Activate the new version first, or these commands run the version before it:

```
atc shell robot@device -a myapp # look around  
atc shell robot@device -a myapp -- ./bin/myapp --version # run it,  
get its code
```

A trailing command forwards its own exit code, so the second form is also a check a script can run.

A **module** is the one kind with a different path, because it has a different pointer. Its payload lives outside the channel tree, and the device selects a version through this link instead:

```
<store-root>/modules/<name>/current  
/var/atc/modules/probe/current
```

There is no `delivery/current/<module>` to look for. The store writes no member link for a module, so the block names the link above instead. A module is still a full member of its delivery in the database, which is what carries it through a rollback.

Four facts about that tree, in the order they bite:

- **delivery/current is a symlink** to a ULID-named delivery directory, and the `<artifact>` inside it is a symlink too. So `find <path> -type f` prints nothing and exits 0: `find` reads the type of its own starting argument, sees a symlink, and never descends. Use `find -L`, `ls -laR -L`, or a resolved path. Keep the `-L` on the `ls` form as well. `-l` makes `ls` report the command-line symlink itself, so a bare `ls -laR` prints one `lrwxrwxrwx` line and lists no file.
- **Delivered files are read-only.** Each one lands `0o440`, or `0o550` when it is executable. No delivered file keeps a write bit. So a listing can look short to a user who does not own the tree, while the program still runs from its full path. Verify by running the program, not by listing the directory.
- **The directories that hold them are not read-only.** They keep the daemon user's own mode, so that user can still create a new file beside a delivered one. Do not predict which way a write goes. Two inputs decide it and neither is a guarantee:
[daemon] `hook_run_as_user` picks the account, and the directory's mode comes from `atcd`'s `umask` when the store was built. A permissive `umask` leaves it group-writable, and `atc-hooks` is in the store group. So the same program can take `EACCES` on one device and succeed on the next. Only the file modes are a guarantee.
- **Enter it with `atc shell <target>`**, which gives you the identity the hooks run as. That is the identity to test a write under, and it is not the owning one. Raw `ssh` as your login user is what makes the facts above look like a failed delivery.

One file in that tree is not yours. A walk of the delivery lists `.atc/lifecycle.pb` beside your payload. It did not come from your `source` directory. `atcd` writes it on the device, and the `atc` CLI never writes it.

`atcd` writes the sidecar for every artifact in a push, whether or not that artifact declares a hook. A bundle that declares none gets a file that declares nothing, and it can be zero bytes long. So an empty sidecar is normal, and it is not a failed write. A pulled delivery is the one exception. The poller's conditions are in [docs/operating/atc-toml.md](#).

The daemon reads the file back later. `atc ota switch`, a rollback, and `atc bootstrap atcd uninstall` each resolve the hooks of a version they did not push, and the sidecar is where they read them. That is why it ships with every version, and not only with a version that declares a hook.

atc.toml is different, and it is usually absent. A push ships every file under `source` and nothing else. Nothing rebuilds the manifest in the version tree. So the manifest lands there only when you keep it INSIDE `source`, where it ships as an ordinary payload file.

The usual layout keeps the manifest beside `source`, and then no `atc.toml` reaches the device.

Exclude the metadata directory when you compare the delivered tree against what you staged. The e2e harness excludes the manifest too, because its own fixtures declare `source = "."` and stage one:

```
atc shell robot@device -a myapp -- find -L . -type f ! -name
    atc.toml ! -path './.atc/*'
```

UNINSTALL

```
atc bootstrap atcd uninstall robot@device          # full
    removal
atc bootstrap atcd uninstall robot@device --keep-store # keep /var/
    atc
```

Full removal stops the unit, disables it, removes it, and purges the store. Remove the local `atc` binary by hand, from `~/local/bin` or `/usr/local/bin`.

DIAGNOSTICS

```
atc ota netcheck robot@device          # connectivity preflight
atc error explain ATC-E0419             # what an error code means
atc diagnose                            # write a redacted support
    bundle
atc diagnose robot@device               # ... plus the
    device's status and atcd log
atc diagnose robot@device --yes         # ... and send it to Agency
    without asking
atc doctor                             # does telemetry from this atc
    reach Agency?
atc diagnose --send <report-dir>        # send the report that a
    failed push wrote
journalctl -u atcd -b --no-pager       # daemon logs (on the device)
```

MORE HELP

atc --help	# every command, grouped by
task	
atc ota push --help	# flags and examples for one
atc completions install	# shell tab-completion
atc bootstrap skill install	# the atc skill for Claude
Code and Codex	

Full docs: <https://atc.dev>