VAAS-X SDK · INDUSTRIAL · CONFIGURATION GUIDE

Configuring Bootstrap for Predictive Maintenance

A practical, step-by-step guide to connecting your own machine fleet, closing the outcome feedback loop, querying across devices, and sizing a deployment — from first machine to production rollout.

10 min	6	~500MB	0
TO FIRST QUERY	SUPPORTED SOURCE TYPES	RAM PER MACHINE, SELF-HOSTED	GPU REQUIRED

VAAS-X Ltd (SC873624) · July 2026 · vaasx.com

Who this is for

You've read the case for catching multi-channel precursor patterns before a threshold alarm fires. This guide is the next step: configuring Bootstrap against your own fleet, from first machine to full production rollout. It assumes a free or paid API key and a Python environment on whatever machine will run the SDK — a line-side gateway, an on-prem server, or a cloud instance, your choice.

Step 1 — install and pick a starting tier

```
pip install vaas-x
```

Every tier includes Bootstrap. What changes between tiers is how many machines you can connect at once and whether you can query across the whole fleet in one call — both directly relevant to a real deployment, covered in full in "Choosing a tier" below. Start on the free tier against one machine; nothing about the configuration changes when you add more devices or upgrade later.

Step 2 — connect your first machine

One `Bootstrap` instance per machine, identified by a `device_id` you choose. A hierarchical naming convention (`<line>_<machine>_<component>`) keeps a large fleet readable in later queries and in the fleet-wide view covered in Step 4.

```
from vaasx import Bootstrap

brain = Bootstrap(api_key="YOUR_API_KEY", device_id="line_3_press_7")
brain.connect("mqtt://press-7.internal/channels")
```

`connect()` accepts whatever your historian or control system already speaks — no separate ingestion pipeline to build:

Source	Typical use
MQTT	Edge gateways, IIoT sensor networks, and OPC-UA-to-MQTT bridges publishing on a broker — the example above
HTTP(S) / WS(S)	REST or streaming APIs from an existing monitoring platform
	Historian exports, batch backfills

Local files (.jsonl / .csv / .json)	
Pandas DataFrame	Data already loaded for analysis in Python
Python generator	Any custom polling loop you already have
Pipe	Wiring Bootstrap directly into an existing local process's own output stream

A PLC or SCADA node that only speaks OPC-UA natively reaches Bootstrap through whichever bridge your historian already uses to get that data onto a network protocol — most industrial OPC-UA deployments already run an MQTT or REST bridge for exactly this reason. There is no schema to write and nothing to label before you start. Bootstrap profiles each channel statistically and classifies its structure on ingestion — the same zero-configuration process validated against the NASA CMAPSS benchmark, where all 24 sensor channels were classified correctly with every identifier stripped and no prior knowledge of what any signal represented.

Connecting a resource-constrained edge sensor — ESP32 example

Step 2 assumes a machine capable of running Python. For a bare sensor node bolted directly onto a machine — an ESP32-class microcontroller reading vibration or temperature at the point of measurement — the SDK isn't what runs on the device. Instead, MicroPython firmware on the ESP32 publishes episodes to your infrastructure directly, over Wi-Fi.

Mode	Fits
Router-direct (HTTP)	Simplest option — the device posts straight to `/episodes/ingest`. Recommended starting point for a single sensor node.
MQTT bridge	The device publishes telemetry to an MQTT broker; a host-side process converts it to episodes. Scales more easily once you're deploying many sensor nodes.
Local edge shard	The device stores and searches episodes locally, syncing to Bootstrap or the router when connectivity allows. Recommended if the sensor's network link is intermittent — shop-floor Wi-Fi frequently is.

Configuration is four values set in the device firmware before flashing: your Wi-Fi credentials, the router or Bootstrap address on your network, and a `device_id` following the same convention as Step 2. Upload with `mpremote`, the standard MicroPython device tool:

```
pip install mpremote
mpremote connect auto fs cp esp32_publisher.py :main.py
```

Whichever mode you choose, the device buffers unsent episodes locally and flushes them automatically once the link recovers — a dropped Wi-Fi connection on the shop floor doesn't lose data, it just delays delivery. A lightweight status endpoint on the device (`http://<device-ip>:9020/health`) confirms it's ingesting before you rely on it.

Everything downstream is identical to any other connected machine: the same `outcome()` feedback loop in Step 3, the same fleet-wide queries in Step 4.

Step 3 — the feedback loop that makes retrieval trustworthy

This is the one step worth treating as a required part of rollout, not an optional extra. Retrieval is weighted toward episodes with a recorded successful outcome by default — but that weighting only works if outcomes actually get recorded. When a flagged pattern is confirmed as a real fault (or ruled out as a false alarm), close the loop:

```
hits = brain.query("vibration rising with temperature drift", k=5)
for hit in hits:
    print(hit["score"], hit["episode"])

# success=True once the fault is confirmed; success=False if it
# turns out to be a false alarm. Either way, future queries on this
# device get more accurate as a direct result of this call.
brain.outcome(episode_id=hits[0]["id"], success=True)
```

A deployment that never calls `outcome()` still retrieves by similarity, but never learns which of those similar-looking patterns actually preceded a fault. Build outcome confirmation into whatever workflow closes a maintenance ticket — that's the natural point where a technician already knows whether the flagged pattern was real.

Step 4 — querying across the fleet, not just one machine

A precursor pattern confirmed on one machine is often relevant to every other machine of the same type. Fleet-wide queries (available from the Developer tier up) fan a single query out across every connected device and merge results by outcome score, so a fault pattern learned on press 7 can surface as a warning on press 12 before it happens there too:

```
from vaasx import HiveMind

fleet = HiveMind(api_key="YOUR_API_KEY")
```

```
fleet.add_shard("line_3_press_7")
fleet.add_shard("line_3_press_12")
hits = fleet.query("vibration rising with temperature drift", k=10)
```

Choosing a tier for a real fleet

Tier	Devices	Fleet-wide queries	Fits
Free	1	No	Single-machine pilot
Developer	Unlimited	Yes	Most real fleet deployments
Professional	Unlimited	Yes	Add vibration audio, thermal imaging, or IMU data alongside sensor telemetry in one memory store
Enterprise	Unlimited	Yes	Sovereign / air-gapped deployment, custom shard topology, dedicated support

Pricing is infrastructure-based per tier, not per query or per episode within your tier's limit — connecting a hundred machines to a Developer-tier account doesn't multiply the bill the way per-token AI pricing would.

Sizing a self-hosted deployment

If you're running on-premise or air-gapped rather than the hosted service, each connected device runs its own process, dominated by a fixed per-process cost rather than how much data that device has accumulated — a machine with 10 stored episodes and one with 10,000 cost roughly the same amount of memory. Measured directly against a live fleet:

Machines	Approx. RAM (device processes only)
1	~0.5 GB
5	~2.5 GB
10	~5.0 GB
15	~7.3 GB
20	~9.8 GB

Budget roughly 500 MB per connected machine, plus about 1 GB of headroom on top of the device total for the OS and any concurrent query burst load.

Deployment model

Profiling, classification, and episode accumulation all run locally, on your own hardware — on-premise, air-gapped, or cloud, your choice. Only the resulting episodes cross the network to reach retrieval. There's no cloud dependency for the equipment itself to keep running, and no GPU requirement: retrieval runs at 14ms mean latency across 1.18 million stored episodes on a standard CPU server.

> **SCOPE, HONESTLY**
>
> What's covered above is what ships and works today: Bootstrap ingestion, retrieval, the outcome feedback loop, and fleet-wide queries from Developer tier up — all production, all benchmarked. Encrypted search over your data is architected into the platform but is currently a roadmap item for custom Enterprise deployments, not part of the standard hosted service — check current availability before relying on it for a compliance requirement.

Configuration checklist

- API key and tier selected against expected device count and fleet-query needs
- `device_id` naming convention agreed before connecting more than one machine
- Data source confirmed against the table in Step 2 — no custom ingestion pipeline needed
- `outcome()` wired into the workflow that closes a maintenance ticket
- Self-hosted only: RAM budgeted at ~500 MB per device plus 1 GB headroom

Get started

Free tier: instant API key, no credit card — `pip install vaas-x`, connect a source, query your first episode in under ten minutes.

Full guide: vaasx.com/docs/getting-started.html | Enterprise: hello@vaasx.com