TestPulse Pro — User Guide

Evidence-first AAA diagnostics for your own lab. Point TestPulse at your RADIUS or TACACS+ server, give it the logs and captures you already have, and get a **named cause with the evidence behind it** — never "RADIUS is flaky".

Every diagnosis answers four things, and refuses to collapse them into one verdict:

Observed outcome	what actually happened across the evidence
Cause family	a named, machine-readable class — not free text
Why	the specific counters and log lines that support it
Fix direction	what to change next

Everything in this guide was run against the shipped package. Where a command's output is shown, that is its real output.

---

1. Install

Python 3.10 or newer. Nothing else is required — no cluster, no container, no second service.

```
python3 -m venv .venv && source .venv/bin/activate
pip install testpulse
```

What it runs on

The wheel is `py3-none-any` — pure Python, no compiled extension, nothing platform-specific in the package itself. **The API, the viewer, the diagnosis engine and the CLI run anywhere Python 3.10+ runs.** What differs between operating systems is not TestPulse; it is the external tools TestPulse shells out to (§A.20), and every one of them is optional.

	Linux	macOS	Windows
API + viewer + `testpulse-diagnose`	✓	✓	✓
Device log collection over SSH	✓	▲	✗ (paramiko, no `ssh.exe` needed)
Packet capture	`dumpcap` + `setcap`	`dumpcap` + ChmodBPF	Wireshark + **Npcap**
`testpulse-capture-setup`	✓	reports, cannot grant	Linux/macOS only

	Linux	macOS	Windows
The shipped RADIUS suite	`freeradius-utils`	`brew install freeradius-server`	no native `radtest`; use WSL
Bring your own `.pcap`			

Two rows are worth reading twice. **`testpulse-capture-setup` is the only entry point that is not portable** — it grants `CAP_NET_RAW`, a Linux capability, and imports `pwd`/`grp` to check your groups, so on Windows it fails at import rather than doing something useless. Nothing else in the package does. And **the shipped test suite needs `radtest`/`radclient`**, which have no native Windows build; everything that reads evidence you already have works there regardless.

The recommendation, plainly: Ubuntu 22.04 LTS. That is what this build is developed and regression-tested on, it is the one platform where every optional tool installs with a single `apt` line, and it is the only one where `testpulse-capture-setup --grant` can finish the job for you. macOS is a close second and fully supported for everything except granting capture rights, which Wireshark's own installer does instead. On Windows, prefer **WSL2 with Ubuntu** over native Python — not because the package fails natively, but because the AAA tooling around it assumes a POSIX box and you will spend your time on that rather than on your network.

Sizing is modest: a clean install measured **115 MB** of virtualenv, the API holds about **96 MB** resident, and 2 GB of RAM with any modern CPU is comfortable. Add `[rag]` and the virtualenv grows to ~539 MB, or ~5.3 GB with `[rag-local]` (see A.15).

Optional extras. **None are needed to start**, and each one is a feature that degrades cleanly when absent rather than a hidden requirement:

Extra	Adds
`testpulse[ai]`	AI narration using **your own** Anthropic/OpenAI key
`testpulse[mcp]`	MCP server, to drive TestPulse from an AI client
`testpulse[pcap]`	Deeper packet analysis (scapy/dpkt)
`testpulse[windows]`	Windows endpoint collection (WinRM, Event Log)
`testpulse[rag]`	Retrieval over your own run history
`testpulse[postgres]`	Postgres instead of the bundled SQLite
`testpulse[all]`	All of the above

---

2. First run

TestPulse **refuses to start without a credential**. That is deliberate: an unauthenticated API bound to `0.0.0.0` is a mistake you get to make exactly once.

Why there is a token, and who issues it

You generate it. Nobody issues it to you. There is no licence server, no account, and no activation call — TestPulse Pro runs entirely on your machine, and a vendor-issued key would mean phoning home, which this build never does.

The token exists for one specific reason: `testpulse-api` serves the viewer **and** the API from a single process on a single origin. There is no reverse proxy in front to add a credential, and baking one into a public download would be handing the key to everyone who downloads it. So the only credential that can protect that origin is one you create yourself at start-up:

```
export TESTPULSE_API_TOKEN="$(python3 -c 'import
secrets;print(secrets.token_urlsafe(32))')"
testpulse-api
```

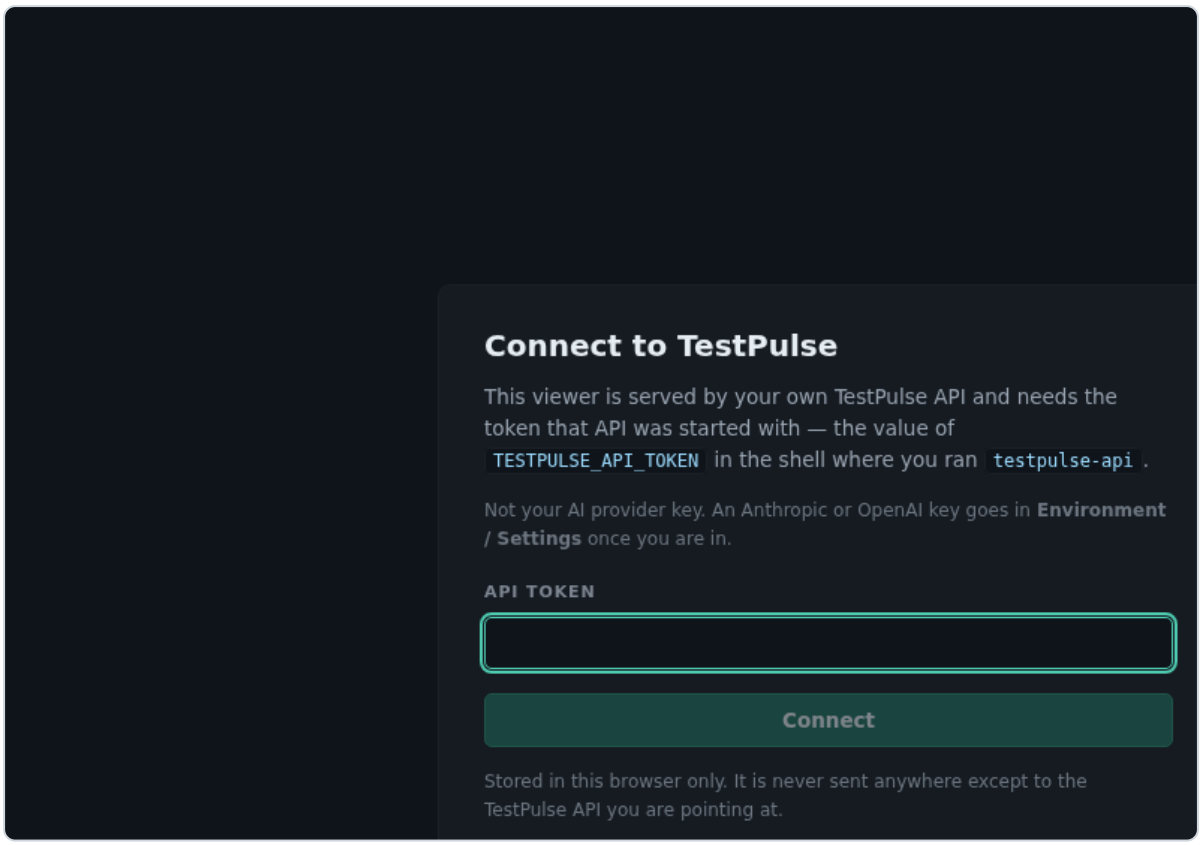
Any value works — the command above simply generates a strong one. **Keep it.** You need the same string again in the next step, and again on any other browser you open the viewer from.

Open **http://127.0.0.1:8000**. The viewer is served by the same process — the compiled UI ships inside the package, so there is no second container to run and no separate front-end port.

The browser asks for the same token

The viewer opens on a **Connect to TestPulse** screen with one **API token** field. Paste the value you just exported — the same string, not a new one.

- The token is checked against `/v1/health` **before** it is stored, so a typo is rejected on the spot. Storing first and discovering the mistake later means every panel fails separately with no explanation.
- Once accepted it is saved in that browser's `localStorage` (key `testpulse.apiToken`) for that origin, and you are not asked again there. It is never written into the downloaded package and never leaves your machine.
- A different browser, a private window, or a different `host:port` is a different origin — you will be asked again. Enter the same token; it is not a new one.
- If a request is ever rejected for authentication, the viewer returns to this screen rather than failing silently.



The first screen, in every browser you open the viewer from. It wants `TESTPULSE_API_TOKEN` *— the value the server was started with — not an AI provider key.*

If you paste an AI provider key here, it will say so. A value starting `sk-ant-`, `sk-proj-`, `sk-`, `AIza`, or `gsk_` is recognised on sight and named, because at this point in setup that is the credential most people have in hand. That key is a different thing entirely: it goes in **Environment / Settings** once you are inside the viewer — see §7.

Variable	Default	Meaning
`TESTPULSE_API_TOKEN`	*(none)*	Bearer token for every `/v1/*` route
`TESTPULSE_PORT`	`8000`	Listen port
`TESTPULSE_CONFIG`	*(none)*	Path to your testbed YAML (§4)
`TESTPULSE_ARTIFACTS`	`./artifacts`	Where run evidence is written

TestPulse writes `testpulse.db` (SQLite) into the working directory on first start. Run it from a directory you intend to keep, or set `TESTPULSE_DB_URL` to a Postgres URL.

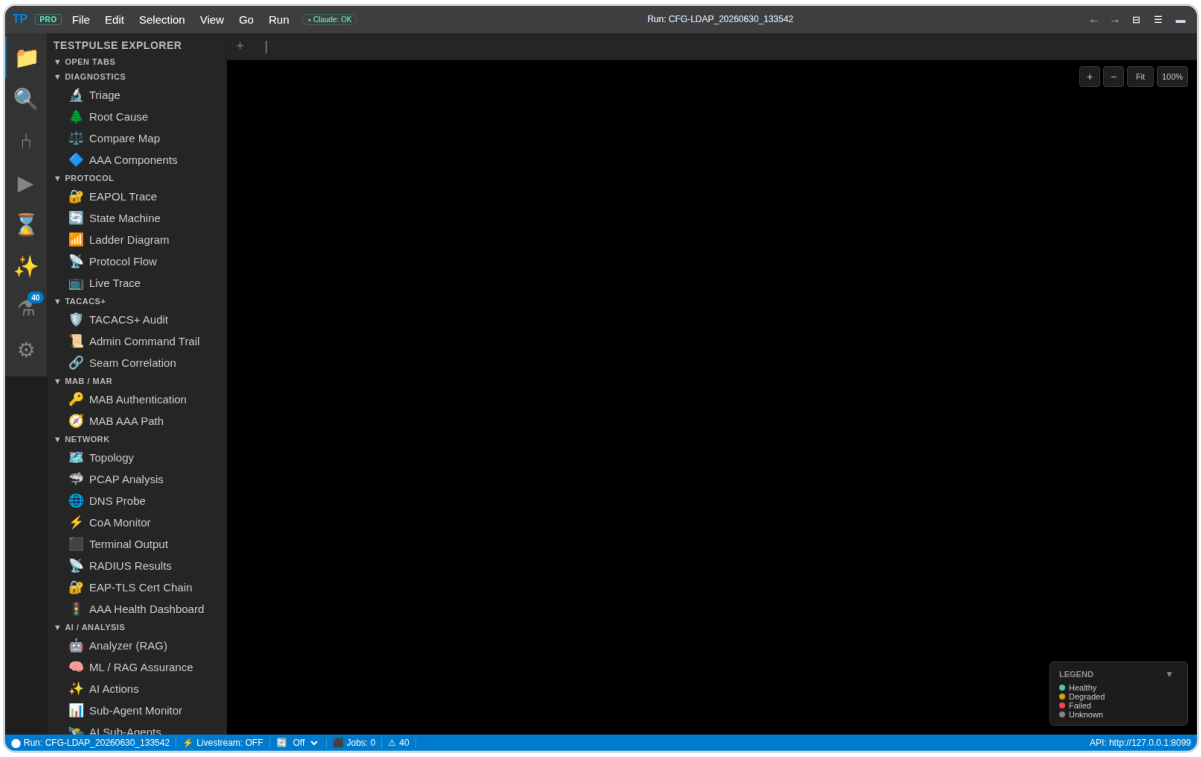
Quick check that it is alive:

```
curl -s -o /dev/null -w '%{http_code}\n' http://127.0.0.1:8000/
healthz          # 200
curl -s -H "Authorization: Bearer $TESTPULSE_API_TOKEN" \
```

```
      http://127.0.0.1:8000/v1/
runs                                                  # your runs
```

`/healthz` is unauthenticated on purpose so a load balancer can probe it. Everything under `/v1/` requires the bearer token.

The Explorer starts collapsed. After you connect, the editor area is empty and there is no tree until you open one: press `Ctrl+B`, or click the icon at the top of the Activity Bar. Nothing auto-opens — that is deliberate (a panel that opens itself covers the Explorer), but it does mean the first screen is blank until you ask for something.



Every panel this build ships, grouped by what it reads: Diagnostics, Protocol, TACACS+, MAB, Network, AI/Analysis, Test Execution, Observability, Evidence. The Activity Bar down the left is eight icons — Explorer, Search Runs, Compare Runs, Test Runner, Jobs, Copilot, Anomaly, Settings. Your runs are listed under RUNS at the bottom of the tree.

A brand-new install is not empty. Five sample runs are seeded into the artifacts root on first start, so every evidence panel has something real to draw before you have run anything yourself. Delete them from the artifacts directory when you no longer want them.

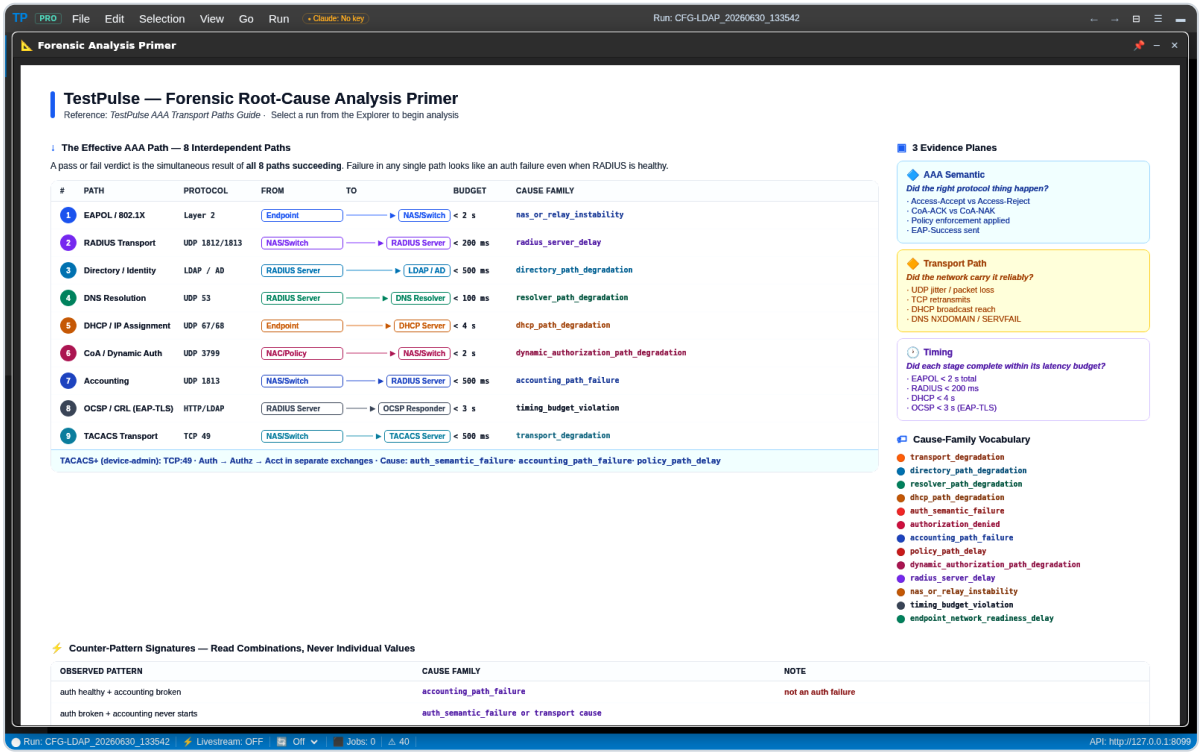
---

3. Diagnose a log you already have

First — how a classification is reached

Before you read your first verdict, it is worth seeing the model that produces it, because the output is deliberately not a single answer. TestPulse ships that model

as a panel rather than leaving it to the documentation: **Explorer → REFERENCE → Forensic Analysis Primer**.



*The effective AAA path as nine numbered hops. Each row is one hop — its protocol and port, the two devices it runs between, the latency budget it is measured against, and **the cause family that hop failing produces**. Down the right: the three evidence planes with the question each answers, and the complete cause-family vocabulary. Underneath, the counter-pattern table — the combinations that are routinely misread.*

Read the table left to right and the classification stops being a verdict handed down and becomes an address. A `directory_path_degradation` is hop 3 — `RADIUS Server → LDAP/AD` over LDAP, budget 500 ms. A `dhcp_path_degradation` is hop 5, `Endpoint → DHCP Server` on UDP/67-68, budget 4 s. `transport_degradation` on TCP/49 is hop 9, the TACACS+ leg. **The cause family names the hop, and the hop names what to go and look at.**

Three things follow from that table, and they are why a diagnosis here refuses to collapse into one word:

- **A pass is all nine hops succeeding at once.** Any one of them failing presents as "authentication failed", including the six that are not authentication. That is the whole reason "RADIUS is flaky" is never an acceptable answer — RADIUS is one hop of nine, and it is frequently the healthy one.
- **Each hop is measured against its own budget**, not a global timeout. EAPOL gets 2 s, RADIUS 200 ms, the directory 500 ms, DHCP 4 s. A stage inside its budget is not a finding no matter how slow it feels.
- **The vocabulary is closed.** The thirteen names in that right-hand column are the only classifications a diagnosis can return. It names one of them or it returns `null` — it never invents a label, so the same failure carries the same name across every run, every panel, and every export.

The panel reads nothing from your run and is identical on every install — it is the key, not the result. Open it beside a verdict when you want to know which hop a cause family is accusing, or what budget a stage was held to. §5 is the same material in prose, for reading away from the machine.

One rough edge in this build: the panel opens at 902×622 and clips its right-hand column at that size — drag the window corner out, and the evidence planes and the vocabulary become readable.

Diagnosing the log

The fastest path to value needs no configuration and no server at all. If you have a `tac_plus.log` or `radiusd -X` capture, diagnose it directly:

```
testpulse-diagnose diagnose --protocol tacacs --file /var/log/tac_plus/
tac_plus.log
```

```
{
  "decision": "reject",
  "confidence": 0.85,
  "events_parsed": 1,
  "evidence": { "TACACS_AUTHEN_REPLY_FAIL": 1 },
  "cause_family": "auth_semantic_failure"
}
```

`--protocol` accepts `tacacs`, `radius`, `dot1x`, `eap_tls`, `peap`, `mab`, and `coa`. The 802.1X variants all ride the same RADIUS Access-Request/Accept/Reject exchange, so they share a parser; `tacacs` and `coa` have their own decision logic.

Give it the raw log, not a summary

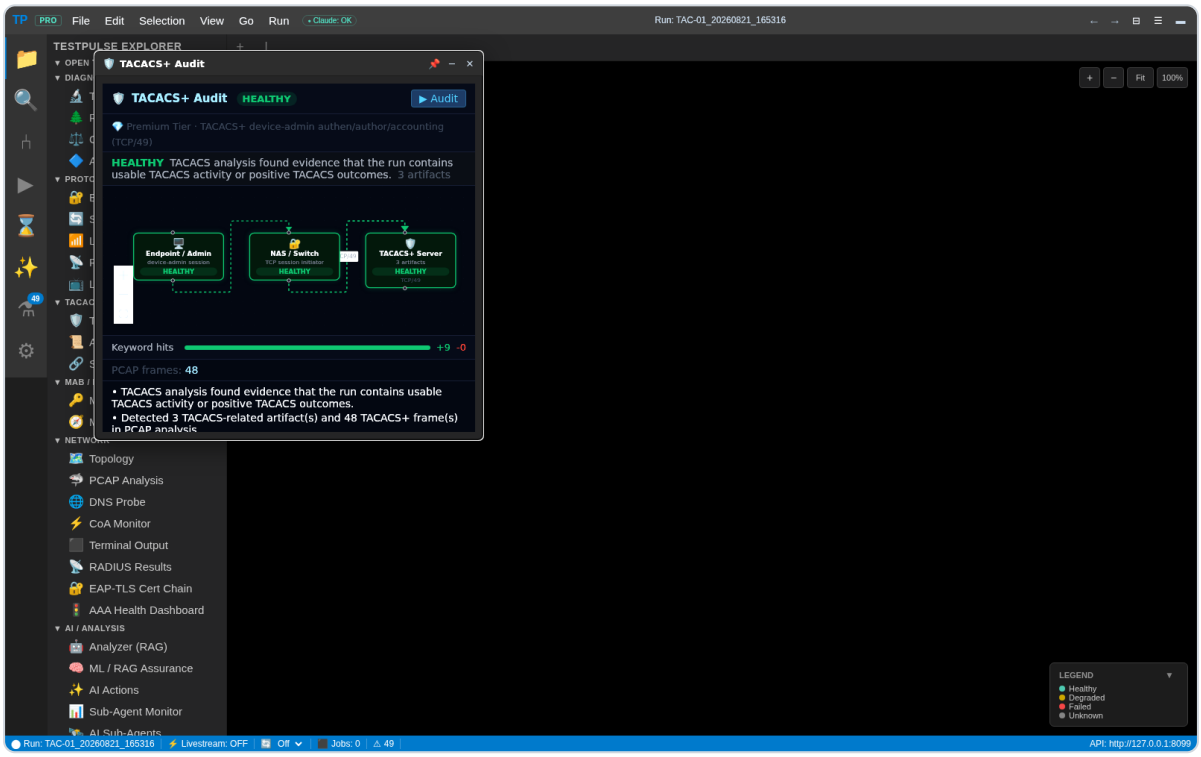
This is the single most common reason a first run returns nothing:

```
{
  "error": "no_events_parsed",
  "message": "No recognizable AAA log lines found for this protocol.
Paste the raw
              tac_plus.log or radiusd -X output, not a summary or
excerpt."
}
```

Protocol	What the parser reads
`radius` and the 802.1X variants	**`radiusd -X` debug output** — the lines containing `Received Access-Request Id …` / `Sent Access-Accept|Reject Id …`, with the `User-Name`, `Calling-Station-Id`, `NAS-IP-Address` attribute lines that follow
`tacacs`	the **tac_plus server log** — `login query …`, `login failure …`, `authorization …` lines
`coa`	

Protocol	What the parser reads
	`radiusd -X` output covering the CoA/Disconnect exchange on port 3799

FreeRADIUS's one-line `Auth: Login incorrect …` summary is **not** enough — it records the verdict but none of the evidence, which is precisely what TestPulse exists to read. Restart with `radiusd -X` (or `freeradius -X`) and capture that.



TACACS+ is a first-class protocol here, not a RADIUS special case: its own audit panel, its own device role on the map, and its own probe on TCP/49.

---

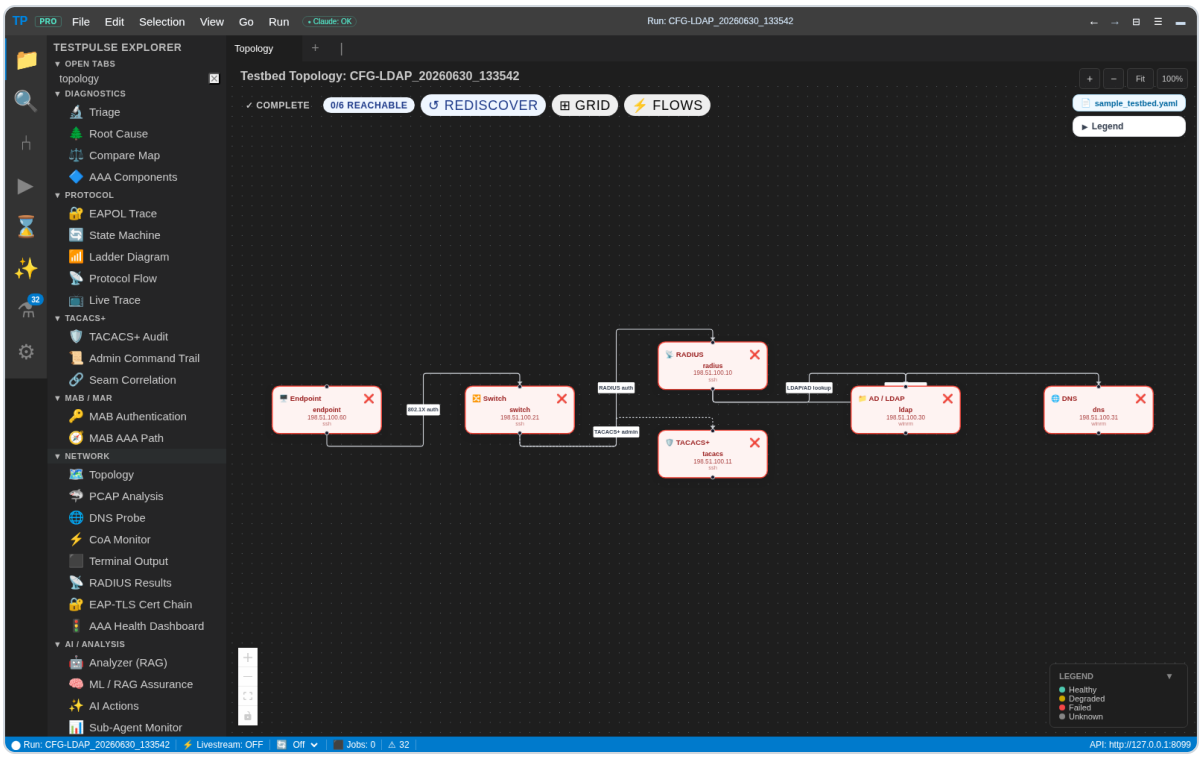
4. Point it at your testbed

TestPulse ships with a **sample testbed** and loads it when you have not configured anything, so the Topology panel draws a network on first run instead of an empty box. Every address in it is from the RFC 5737 documentation range, so every device reads as unreachable — that is the truthful result, not a failure, and it shows you the shape of the map before you own one.

To map **your** network, copy `env.example.yaml` from the package, edit the addresses, and point at it either way:

```
export TESTPULSE_CONFIG=/path/to/my-testbed.yaml
# explicit, wins over everything
# —— or just save it as ./testbed.yaml where you run testpulse-api
```

Topology then draws your devices, pinging each one for reachability and latency. The file drives everything: which devices exist, where their logs are, and where to capture. It is the source of truth — TestPulse has no hidden device list.



First run, before you have configured anything: the packaged sample, drawn from `sample_testbed.yaml` *(named in the chip, top right). Every address is from the RFC 5737 documentation range, so the count reads* ***0/6 reachable*** *— the truthful result on a machine that owns none of them. Point* `TESTPULSE_CONFIG` *at your own file and the same panel draws your network with real ping and latency.*

Resolution order, first hit wins: `TESTPULSE_CONFIG` → `./testbed.yaml` → `./env1.yaml` → the packaged sample.

Credentials: named, never guessed

Prefer `*_env` and keep the secret out of the file:

```
radius:
  ip: 198.51.100.10
  secret_env: TESTPULSE_RADIUS_SECRET      # export
TESTPULSE_RADIUS_SECRET=…
```

TestPulse ships no default shared secrets. If one is missing it tells you which variable to set and stops. That is the honest behaviour: sending a guessed shared secret to your RADIUS server produces an authentication failure on *your* box, which then comes back as a TestPulse finding — a confidently wrong answer is worse than no answer.

A literal value in the file works too. Then the file holds a secret: `chmod 600` it and never commit it.

Declare only what you have

Sections you leave out are **skipped, not assumed**. If you do not declare a DHCP pool, TestPulse does not check lease placement and does not guess a range your network never agreed to — the result is "unknown", never "wrong". The same is true of `ldap:`, `pcap:`, and `switch:`.

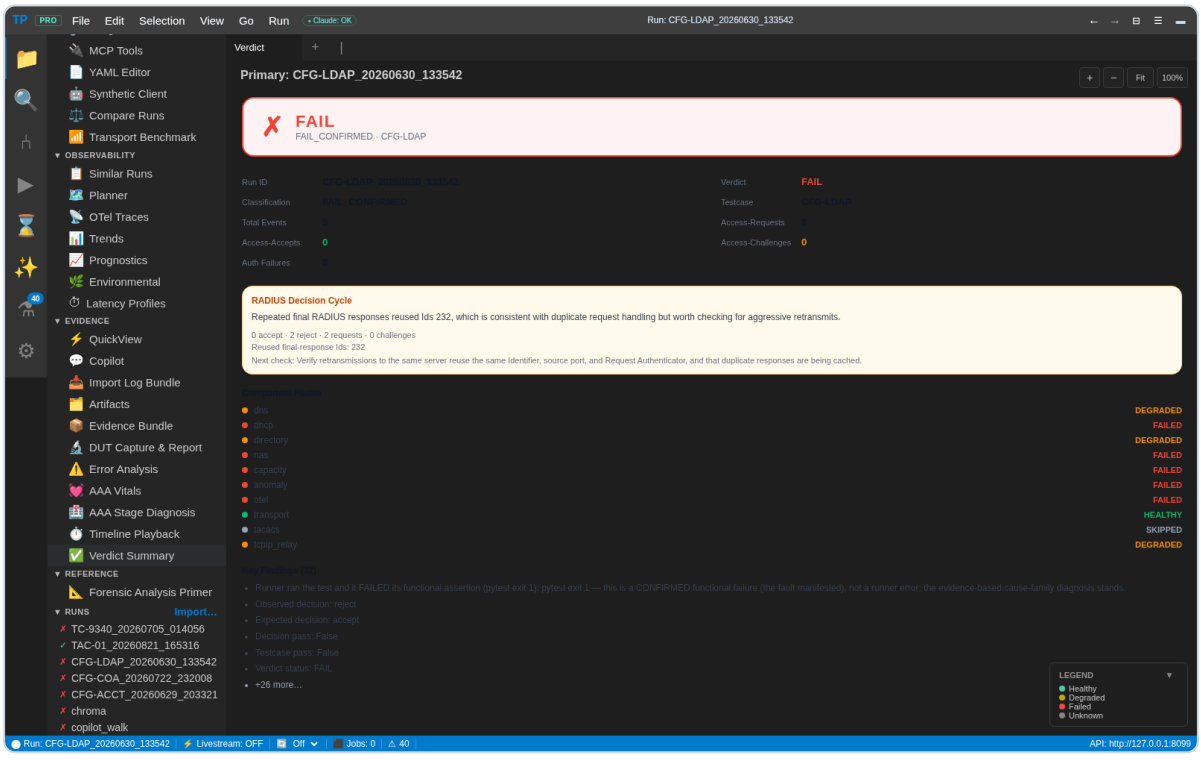
---

5. Reading a diagnosis

The three evidence planes

A verdict is only trustworthy when all three agree, so all three are always evaluated:

Plane	Question it answers
AAA semantic	did the right protocol thing happen? (Accept/Reject, CoA, policy)
Transport path	did the network carry it reliably? (loss, retransmit, DNS/DHCP/LDAP health)
Timing	did each stage finish inside its latency budget?



One run, all three planes. The verdict and its classification at the top, the RADIUS decision cycle in prose with the counters behind it, then per-component health — `dns` degraded, `dhcp` failed, `directory` degraded, `transport` healthy, `tacacs` skipped. "Skipped" and "failed" are different answers, and the panel keeps them apart.

The effective AAA path

Identity and policy lookups make DHCP, DNS, and the directory part of the authentication path, not adjacent to it:

```
endpoint → DHCP/DNS → NAS/switch → RADIUS/TACACS+ → policy → LDAP/AD → enforcement
```

Cause families

The vocabulary is fixed and machine-readable. A diagnosis names one of these or returns `null` — it never invents a new label:

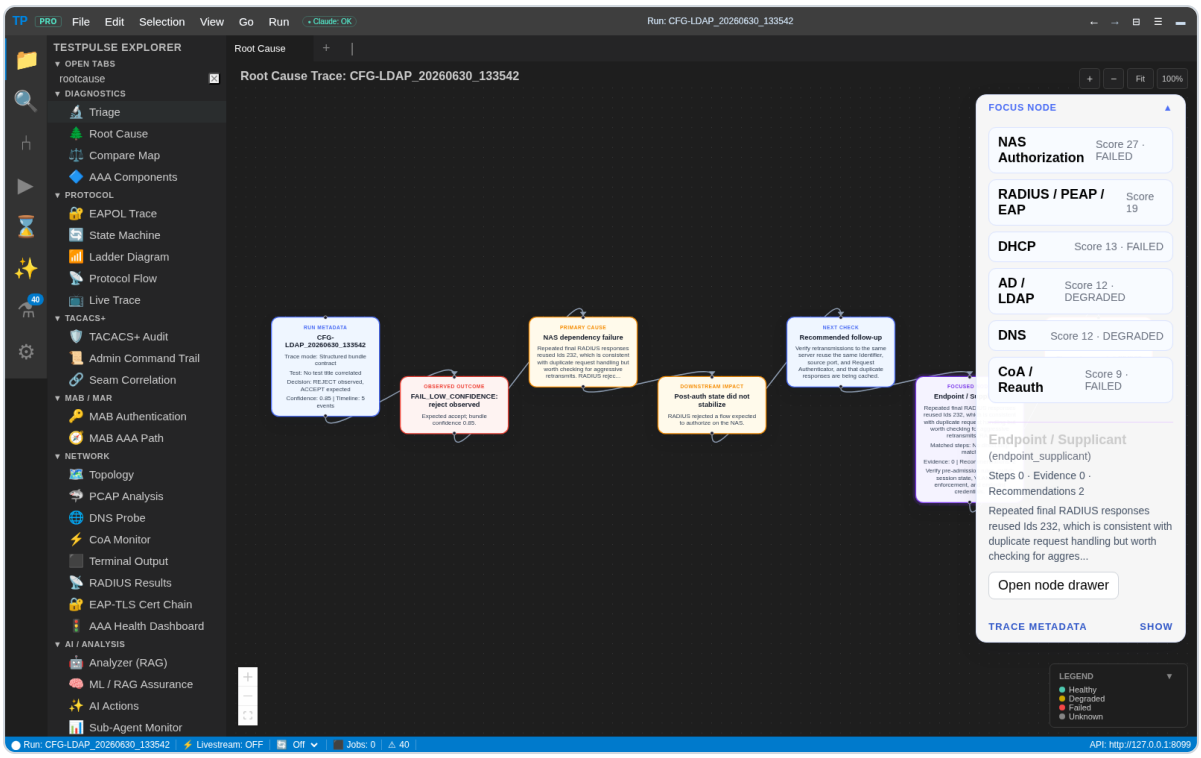
`transport_degradation` · `directory_path_degradation` · `resolver_path_degradation` · `dhcp_path_degradation` · `policy_path_delay` · `radius_server_delay` · `nas_or_relay_instability` · `dynamic_authorization_path_degradation` · `authorization_denied` · `auth_semantic_failure` · `accounting_path_failure` · `timing_budget_violation` · `endpoint_network_readiness_delay`

Read combinations, not single counters

The pattern across counter groups is the diagnosis. A few that are routinely misread:

Pattern	Cause family	Not this
auth healthy, accounting broken	`accounting_path_failure`	an auth failure
auth healthy, enforcement never applied	`policy_path_delay` or `nas_or_relay_instability`	an auth failure
EAP-Success, then endpoint at `169.254.x.x`	`endpoint_network_readiness_delay`	an auth failure — the port never moved to the target VLAN
Access-Request sent, no reply of any kind	`radius_server_delay`	a reject
CoA sent, NAK or no reply	`dynamic_authorization_path_degradation`	a policy error

`dhcp_path_degradation` means **"this MAC has no confirmed lease"** — established from the lease file, the switch binding table, ARP, or a RADIUS Accounting `Framed-IP-Address`. Never from a ping: an endpoint can hold a valid lease and still not answer ICMP.



The same run as a causal chain: run metadata → observed outcome → primary cause → downstream impact → the next check to run. The panel on the right scores each component in the effective AAA path, so you can see which one the verdict actually rests on.

---

6. Packet capture

Capture points are declared in your testbed YAML, so TestPulse captures what you tell it to and nothing else:

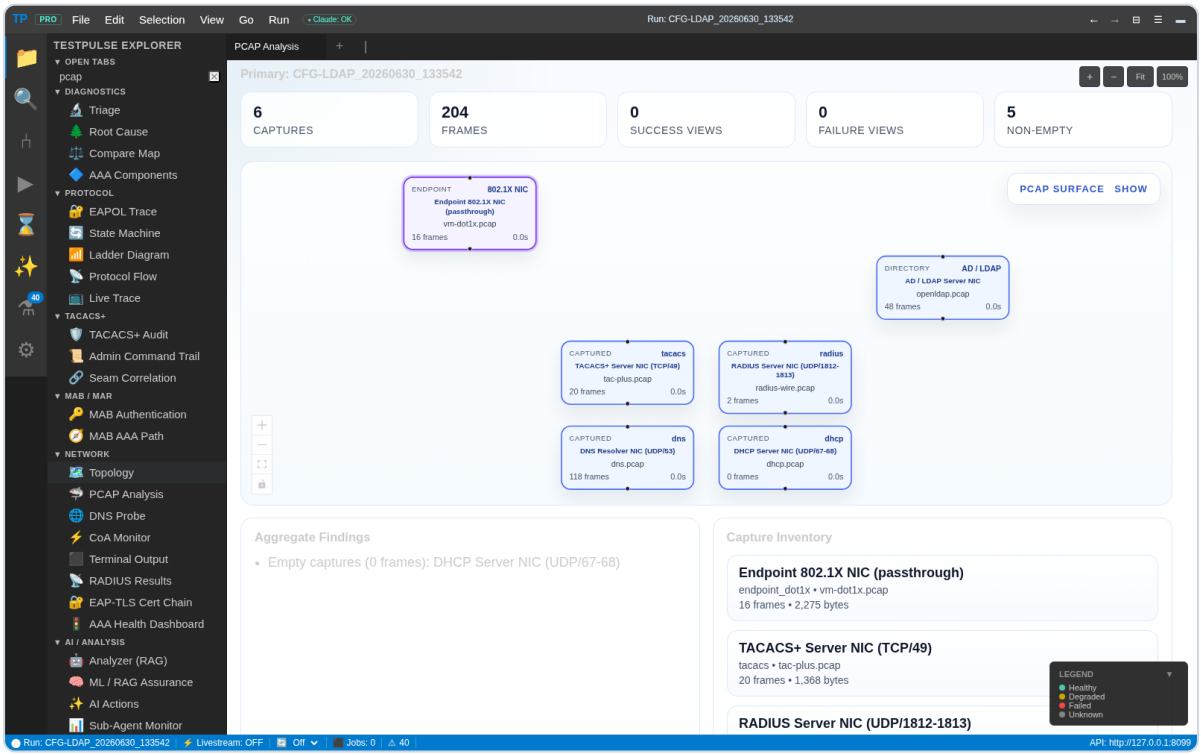
```
pcap:
  capture_points:
    - name: radius-wire
      bpf: "udp port 1812 or udp port 1813"
    - name: tacacs-wire
      bpf: "tcp port 49"
```

```
testpulse-pcap-start  <run-id> --duration 30      # run-id is positional
testpulse-pcap-status <run-id>
testpulse-pcap-stop   <run-id>
```

The run id is a directory name under the artifacts root (`TESTPULSE_ARTIFACTS`, `./artifacts` by default) — the same root the API writes runs to, so a run created in the viewer is addressable from the CLI by its own name.

These three commands drive the **running API**, they do not capture locally. The artifacts root that matters is therefore the one the *server* was started with: if

`testpulse-pcap-start` reports a path you did not expect, check the environment of the `testpulse-api` process, not your shell.



Six captures, 204 frames, one card per capture point — and the DHCP NIC called out as empty rather than quietly averaged into the total. An empty capture is a finding, not a gap to paper over.

Bringing your own capture

Where TestPulse cannot capture for you — no SSH to the device, a span port you mirror by hand, a capture a colleague sent you — drop the file into the run directory under the device's name (`radius.pcap`, `switch.pcap`, `dhcp.pcap`, `ad_ldap.pcap`, `wireless.pcap`) and it is picked up as evidence.

A missing capture is recorded as missing. The evidence bundle is marked partial and no forensic claim is made about transport-path health, rather than quietly inferring one from the logs. Deeper frame analysis needs the `[pcap]` extra; without it, frame and byte counts are still produced.

---

7. AI narration is optional, and it is your key

TestPulse ships with **no AI key and no AI subscription**. Parsing, correlation, the deterministic diagnosis, the cause family, and the evidence bundle all run locally with no network call. That is the product; AI is narration on top of it.

Install the provider SDK first — it is not in the base package, and which one you need depends on whose model you are using:

```
pip install 'testpulse[ai]'     # Anthropic or OpenAI, with your own key
pip install 'testpulse[aws]'    # Bedrock / SageMaker in your own AWS
account
```

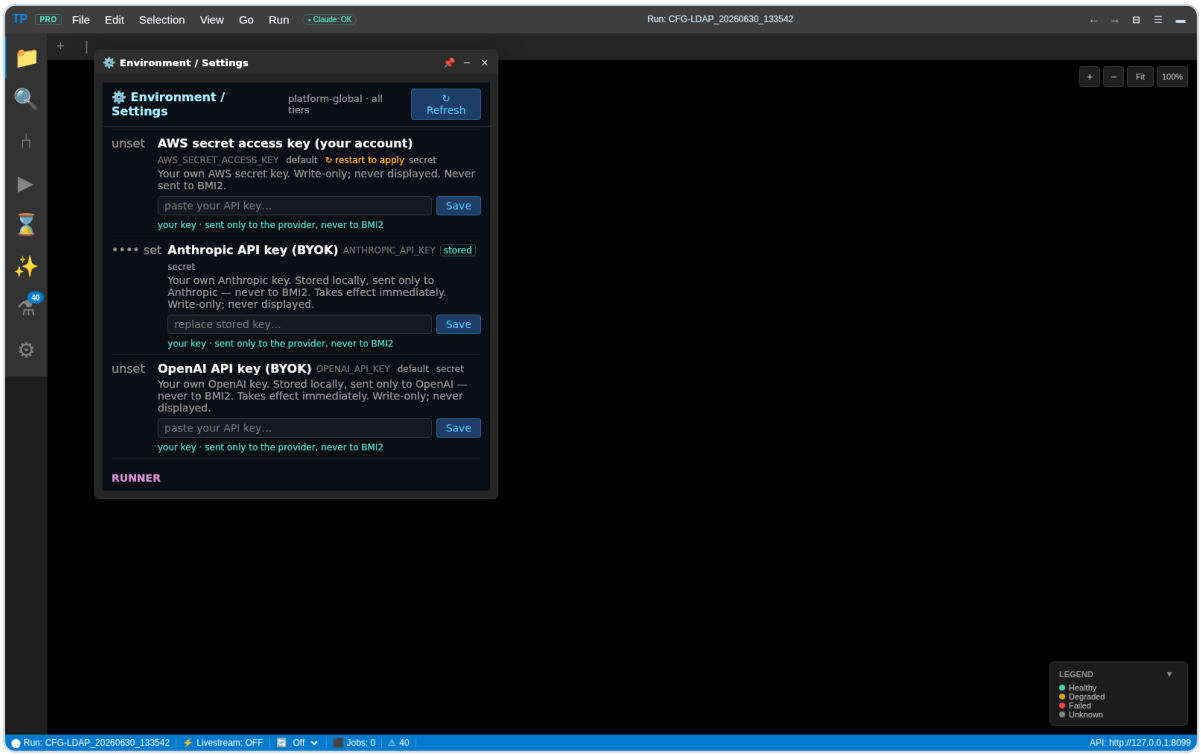
Getting this wrong is the single most common way a correctly-configured install still refuses to reason: the credential saves, the panel shows it stored, and nothing says the SDK behind it is absent.

Then supply your key, either way:

```
testpulse-diagnose configure --provider anthropic --api-key sk-ant-…
testpulse-diagnose status
```

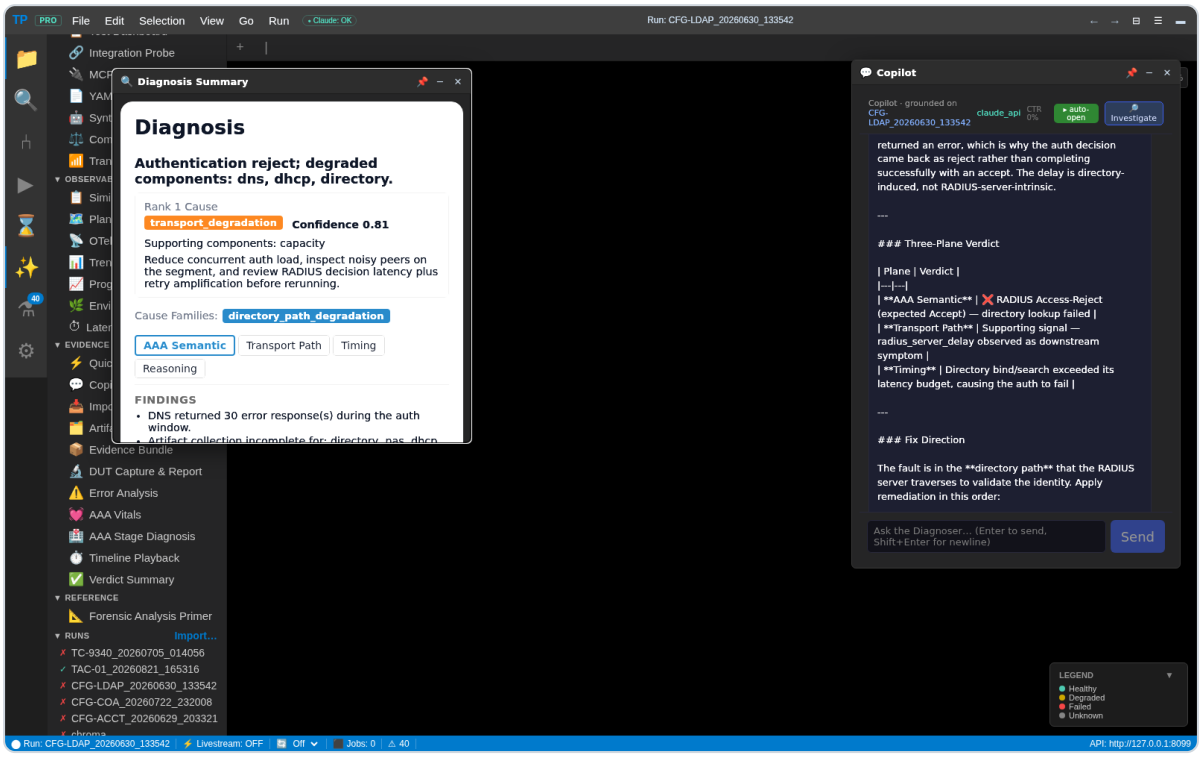
`status` shows the active plan, entitlements, whether a BYOK key is configured, and — worth checking once — **where that key actually landed**. The CLI prefers your OS keychain and falls back to a `0600` file when no keychain backend is reachable, which on a headless box is the common case.

The same key can be entered in the viewer under **Environment / Settings**, where it applies immediately with no restart. That path is a **different store**: the viewer writes it to `<artifacts>/_settings/platform_settings.json`, mode `0600`, readable by your user. Either way it stays on your machine and is never sent to BMI2 — but if your threat model requires a key never to sit in a file, use the CLI on a host with a working keychain, or pass `ANTHROPIC_API_KEY` / `OPENAI_API_KEY` in the environment and save nothing.



`Edit → Environment / Settings…` → *the AI/RAG group. Keys are write-only: the field shows* `set` *or* `unset` *and never the value, and each row states where the key goes — to that provider, never to BMI2.*

A key **without** the SDK is the one combination that looks configured and is not: the key saves, and reasoning still refuses. TestPulse tells you so when you save the key, and again if you ask a question anyway — but installing the extra first avoids the round trip.



With a key configured, the Copilot answers from the run's own evidence bundle — here it names the cause family, gives a verdict per evidence plane, and ends on a fix direction. It also opens the panel its answer rests on, so the claim and the evidence are on screen together. Every number in that answer came from the local diagnosis; the model is narrating it, not producing it.

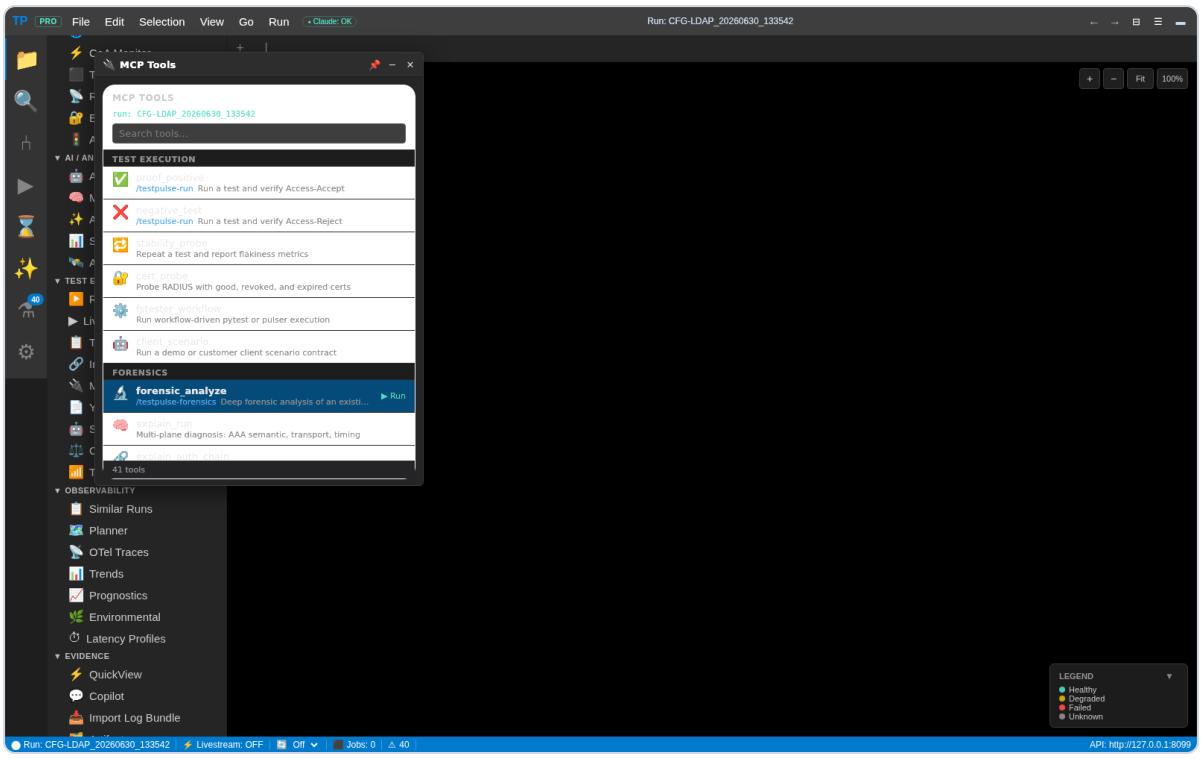
---

8. Driving TestPulse from an AI client (MCP)

With the `[mcp]` extra, TestPulse exposes its tools over MCP so an AI client can run tests, pull evidence, and read diagnoses directly:

```
pip install 'testpulse[mcp]'
testpulse-mcp          # stdio transport
testpulse-mcp-sse      # SSE transport
```

Point your MCP client at the command. `testpulse-menu` lists the available tools grouped by category.



The same tools are browsable in the viewer — searchable, grouped, and runnable against the selected run without leaving the page.

---

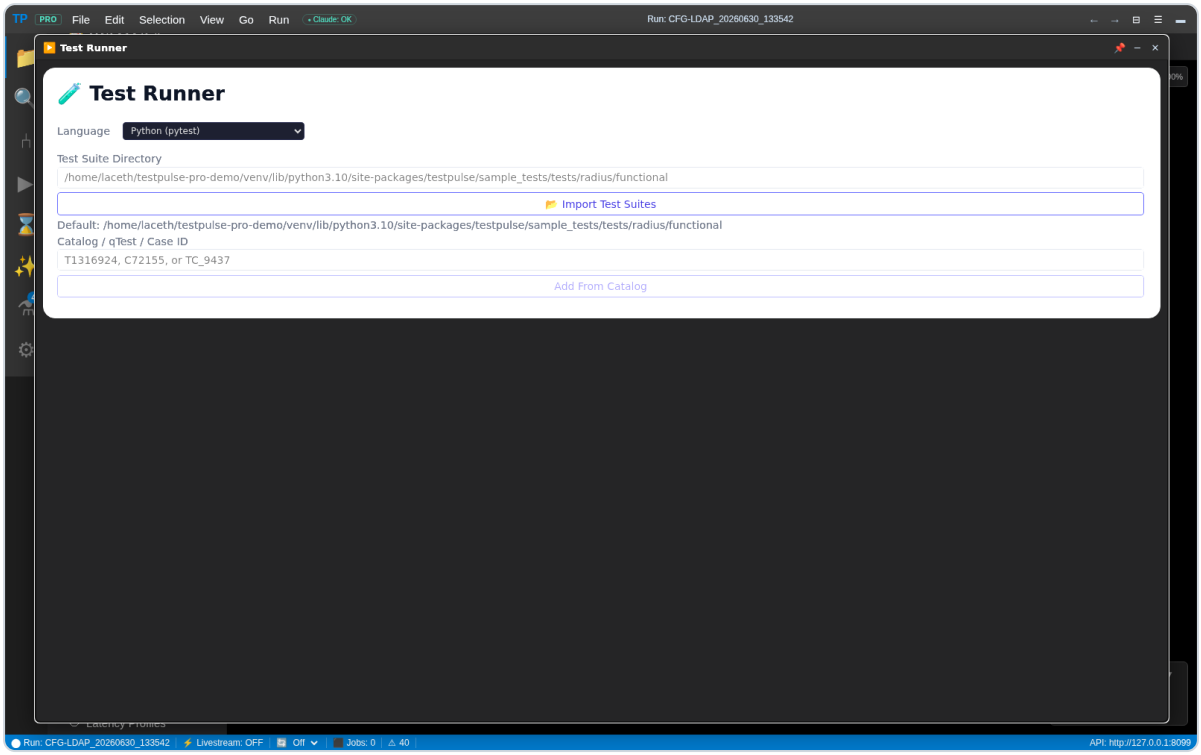
9. The CLI

The package installs 31 commands. The ones worth knowing first:

Command	Does
`testpulse-api`	the API + viewer, one process
`testpulse-diagnose`	BYOK config, plan status, one-shot log diagnosis, feedback
`testpulse`	run diagnostics against the configured testbed
`testpulse-menu`	list every available tool by category
`testpulse-config`	`list` / `show` / `validate` / `select` / `edit` your testbed YAMLs
`testpulse-tacacs-audit`	TACACS+ command-authorization audit
`testpulse-seam-correlate`	correlate evidence across component seams
`testpulse-save-baseline` / `-list-baselines` / `-delete-baseline`	baseline management
`testpulse-index-run` / `-delete-run`	run index maintenance
`testpulse-pcap-start` / `-stop` / `-status`	capture control

Command	Does
`testpulse-migrate`	database schema migration

Every command supports `--help`.



The shipped suite from the viewer rather than the shell. The Test Suite Directory defaults to the sample tests inside the installed package, so there is something to run before you have written anything.

---

10. What this build does not include

Deliberately, and stated rather than hidden:

- **The Trainer** — the curriculum, grading, and human sign-off loop is a separate product.
- **Virtual-lab surfaces** — live attack generation and the cluster panels need a lab this download does not ship. They are **absent, not disabled**: a greyed-out panel advertising something it can never run is worse than no panel.
- **Vendor plugins** — vendor-specific appliance packs ship separately. The core is RADIUS and TACACS+, which is what this build does well.

---

11. Troubleshooting

Symptom	Cause	Fix
	no credential configured	set `TESTPULSE_API_TOKEN` (§2). `TESTPULSE_ENV=dev`

Symptom	Cause	Fix
`refusing to start in unauthenticated open mode`		allows open mode in development only
`401` from a `/v1/*` route	missing or wrong bearer token	send `Authorization: Bearer $TESTPULSE_API_TOKEN`. `/healthz` is the unauthenticated probe
The viewer opens on a **Connect to TestPulse** screen	expected on first use in each browser — the build ships with no credential in it	paste the value of `TESTPULSE_API_TOKEN` (§2). The same string the server started with, not a new one
"That looks like an AI provider key" on that screen	pasted an Anthropic / OpenAI / Google / Groq key	that key goes in the AI panel (§7). This screen wants `TESTPULSE_API_TOKEN`
Asked for the token again on another browser or port	the token is stored per browser origin, not on the server	enter the same token there. Nothing is wrong
"this browser is blocking local storage" on that screen	`localStorage` is blocked — private window, embedded webview, or site data disabled	allow site data for that address, or open the viewer in a normal (non-private) window
`{"error": "no_events_parsed"}`	a summary log, not raw output	use `radiusd -X` output or the tac_plus server log (§3)
"which variable do I set?" on start	a shared secret is declared `*_env` but unset	export the named variable. TestPulse names it rather than guessing a secret
Address already in use	something already holds `:8000`	set `TESTPULSE_PORT`
Frame counts appear but no protocol breakdown	`[pcap]` extra not installed	`pip install 'testpulse[pcap]'`
Open in Wireshark does nothing, or reports it could not launch	the API has no desktop to draw on — it was started without `DISPLAY` (a service manager, a bare `ssh`, a scrubbed launcher). Wireshark itself is usually fine	TestPulse now detects the display; when it cannot, start the API from your graphical session or name it: `DISPLAY=:10 testpulse-api`. `who` shows which display is yours. See A.13
Wireshark opened but you cannot see it	it went to a different desktop — console `:0` while you are on xrdp `:10`, or the reverse	check `display` in the response (A.13), then restart the API with that `DISPLAY`
Copilot answers *no reasoning provider available*	a key is set but the AI SDK is missing	`pip install 'testpulse[ai]'` — or `'testpulse[aws]'` for

Symptom	Cause	Fix
		Bedrock/SageMaker. A key alone is not enough
AWS keys saved, Bedrock still not used	`TESTPULSE_AI_PROVIDER` still defaults to `openai`	set it to `bedrock` in ⚙ AI/RAG. Credentials do not select a provider
AWS settings saved but nothing changed	they are marked **↻ restart to apply** — boto3 clients are built at start-up	restart `testpulse-api`; the log prints `applied N stored setting(s)`
Badge says **Claude: No key** while the Copilot answers	the badge reports whichever provider the router can see	fixed — the router now enumerates Bedrock, so the badge reads `DeepSeek (Bedrock): OK`. If it persists, check `/v1/llm/status` shows `bedrock.available: true`
A panel you expected is missing	it is out of scope for this build	see §10 — absent is intentional
Any other error code or message	—	**Addendum B** is the full reference: HTTP codes, machine-readable `error` slugs, runner console messages, and the two warnings that are the product disagreeing with its own AI
Topology shows devices you do not own	no config yet, so the shipped sample is loaded	point `TESTPULSE_CONFIG` at your own YAML, or save one as `./testbed.yaml`

---

12. Feedback

Beta feedback goes through the product:

```
testpulse-diagnose feedback
```

or the **Feedback** tab in the viewer. **Feedback never includes your raw logs or captures** — it carries your message and the build identifier, nothing else.

If you are blocked and need a person rather than a form, mail **support@bmi2.com**. That address is a forwarder to the maintainer, so it is a best-effort beta channel and not a contracted SLA. Attach the output of:

```
testpulse-diagnose status
```

It prints the plan, the entitlements in force, and whether a BYOK key is configured — which is most of what a first reply would otherwise have to ask you for. Say what you ran alongside it.

Addendum A — Panels removed in the 2026-08-27 scope pass, and how a panel gets removed

Added 2026-08-27. Read this if a panel you saw in an earlier build is gone, or if you maintain the build and need to take one out.

A.1 Why this pass happened

§10 says what this build leaves out. That list was accurate about the *categories* — Trainer, virtual lab, vendor plugins — but the per-item decisions behind it were not all made. The scope matrix that drives the build records a verdict and a reason for every panel, and while every excluded panel had a specific, dated, argued reason, all 72 shipped panels carried the same one sentence:

> *"Works from the tester's own evidence — no lab dependency."*

Exclusion was decided item by item. **Shipping was the default nothing had to earn.** So panels that do need a lab stayed in, because no one had asked them individually. This pass asked.

A.2 What was removed

Nine panels, each because the thing it reads does not exist on a machine that is not BMI2's lab:

Panel	What it needed that you do not have
RFC 2544 Flow	the in-cluster Ostinato drone. The flow snapshot returned 404 on every run
PCAP Device Control	capture aimed at lab pods — every run recorded `pcaps=0`
Burst Alerts	a **Grafana webhook**, and therefore a Grafana/Prometheus stack
Windows Audit	a Windows endpoint reachable over WinRM
MAB MAC Repository	the FreeRADIUS `authorize` repository — a lab-server file, not your evidence
Relay Path	`tcp-relay`, a virtual-lab construct. 404 on every run
Parallel Analysis	no data path in this build. 404 on every run
Test Case Ref	qTest — BMI2's test-management integration
fstool Analysis	Forescout `fstester` tooling — internal

RFC 2544 itself did not go away. The loopback probe still ships and still works, with no drone and no lab:

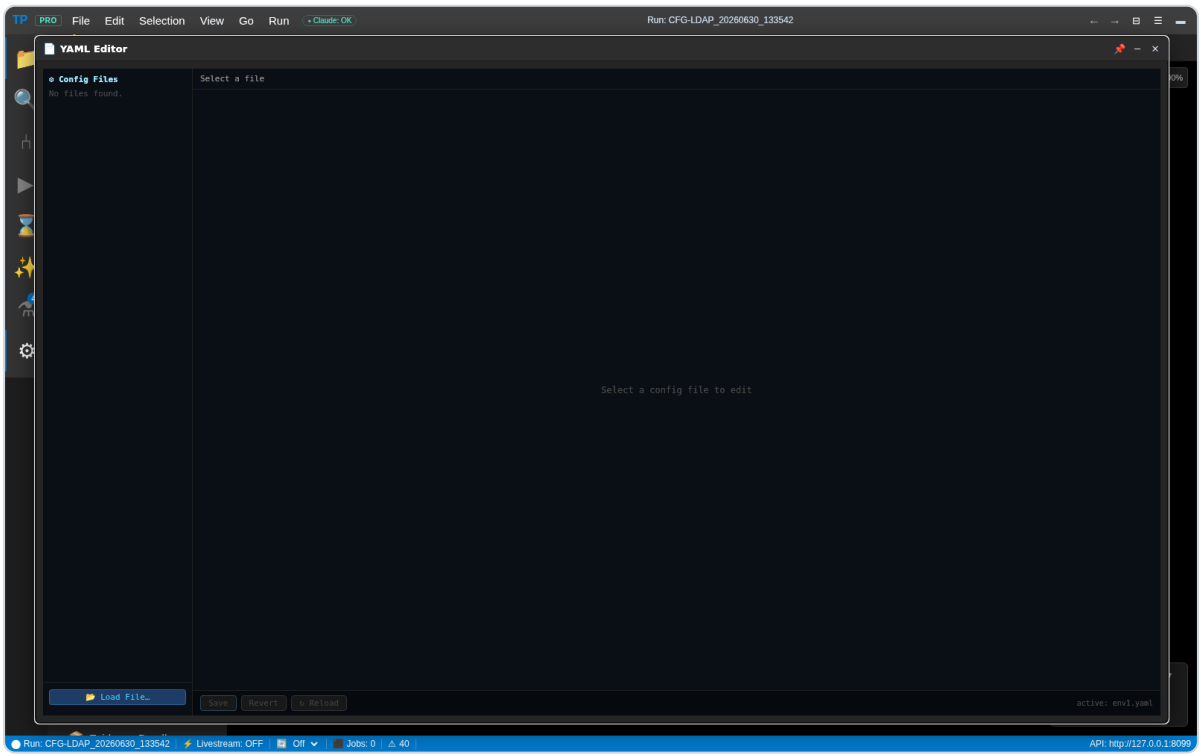
```
curl -X POST -H "Authorization: Bearer $TESTPULSE_API_TOKEN" \
     -H 'Content-Type: application/json' \
     -d '{"loopback_mode": true, "frame_sizes": [64, 512, 1518]}' \
     http://127.0.0.1:8000/v1/transport/rfc2544
```

It reports `[LOOPBACK MODE — NON_COMPLIANT]` in its own summary, because loopback latency is not an RFC 2544 compliant measurement. That label is the point: a number you can trust to be honest about what it is beats a compliant-looking number produced without the hardware the standard requires.

Two more surfaces were removed from the **server**, not the UI: `/v1/testlab/*`, which served BMI2's internal CI suite catalog, and `/v1/faults`, the Source-of-Truth fault injector. Neither is a product feature and neither should have been reachable from a download.

A.3 Where to edit your testbed

Explorer → Test Execution → YAML Editor, or **Edit → Edit Testbed YAML**. Both open the same panel: it lists the config files it can find, edits them, and sets which one is active.



The editor as it opens on a fresh install. ***Known limitation in this build:*** *the file list only discovers* `env*.yaml` / `env*.yml` *(and the legacy* `radius.yaml` *names), so a testbed saved as* `./testbed.yaml` *— the filename §4 recommends — is not listed, and the footer falls back to reporting* `active: env1.yaml`*, a file that does not exist in your install. Topology and the Run button resolve your file correctly; it is the picker that does not see it. Until this is fixed, either name your file* `env1.yaml`*, point* `TESTPULSE_CONFIG` *at it, or open it here with* ***Load File…****.*

Corrects an earlier draft of this addendum, which said the gear — *Edit → Environment / Settings…* — was the one place that edits your testbed YAML and

loads a device file. It is not, and never was. The gear holds feature flags, platform settings, and your AI key. It touches no config file. That mistaken belief is why the YAML Editor was removed from an interim build, and why *Edit → Edit Testbed YAML* opened an empty window in it. Both are restored.

What *did* stay removed is genuine duplication:

- **Testbed Config** — a second Explorer entry opening the identical panel.
- **The Test Runner's own Config File dropdown** — a third place to answer "which testbed am I driving?", which let you run against one testbed while every other panel showed another. The Test Runner follows the active config now, like every other surface.

Two of these — the guide's own claim, and the removal made on it — are recorded here rather than quietly corrected, because the removal shipped in a build.

A.4 How a panel is removed (for maintainers)

Never by editing the TypeScript. `web/src/proBuildScope.ts` is generated. The decision lives in one place and the build is derived from it, so that a panel decided out cannot quietly ship anyway:

1. Set the row's `Verdict` and `Reason` in `docs/reference/matrices/pro_build_explorer_scope.csv`. Date the reason and say what the panel needed that a downloader does not have.
2. `python scripts/gen_pro_build_scope.py`
3. `cd web && VITE_TESTPULSE_BUILD=pro npm run build`
4. `python scripts/pro_panel_regression.py --base http://127.0.0.1:8088`

Step 4 is not optional. It opens every remaining panel and every title menu on a fresh page and reports what broke — measuring ambient traffic first, so a panel is blamed only for what it newly caused.

Verdicts:

Verdict	Effect
`SHIP`	in the Explorer, in the bundle
`EXCLUDE-LAB` / `-TRAINER` / `-COMMERCIAL` / `-INTERNAL` / `-PLUGIN` / `-PRODUCT`	out of the Explorer, out of the bundle, and any title-menu entry that opens it is hidden too
`EXCLUDE-EXPLORER`	**out of the Explorer only.** The code still ships and every other route into it still works

That last verdict exists because of a real failure. *Environment / Settings* was marked `EXCLUDE-PRODUCT` on the reasoning that it is reached from the title bar and does not belong in the Explorer tree — correct on its own terms. But every `EXCLUDE` verdict also stubs the component out of the bundle, so the gear opened an empty shell, and the BYOK key entry that lives inside it was unreachable in the one build that needs BYOK. **Removing a panel and moving its entry point are different decisions, and the matrix could previously only express the first.**

Three routes reach a panel — the Explorer tree, the Quick Action bar, and the title menu — and the generator now derives all three from the same CSV rather than from three hand-kept lists. Menu entries that *call an API* instead of opening a panel cannot be derived from the window map, so they are listed explicitly in `NON_FLOAT_MENU_ACTIONS` with the route each one needs; that is how `Run → EPC + PCAP Capture…` was still being offered in a build where `/v1/epc/*` is 404.

A.5 Also fixed in this pass

- **The gear opened an empty window.** Cause and fix in A.4.
- **The Copilot appeared before you asked for it.** It was auto-opened at startup, which covered the Explorer. Nothing auto-opens now.
- **`GET /v1/pcap/devices` named devices your config does not contain.** It read capture points under the key `device:` only, while a generated testbed writes `name:` — so your capture points were invisible and it fell back to a legacy list. Both spellings are accepted now; the config is the source of truth for what exists.
- **DUT Capture & Report was never signed in.** The panel called the API through a fetch helper of its own that sent no `Authorization` header, so every request came back 401 while the same URL answered 200 to `curl` with the same token. The panel read as broken and the API as inconsistent; neither was true — the request simply arrived anonymous.
- **Two panels reported 405 on open.** PCAP Analysis and AI Actions posted telemetry to an unversioned path the API does not route, and the single-page fallback — which answers `GET` only — replied 405.
- **Two panels polled routes this build removes** — the Sub-Agent Monitor asked for the AI-subscription surface every three seconds, and the Test Runner asked for a Trainer route once per install. Both now know before asking.
- **The Run button could not find a testbed at all** when installed from the wheel, reporting a path inside your own virtualenv. It now resolves the same testbed the Topology map draws.

A.6 If a panel you need is missing

Removal is a judgement about what a download can support, not about what matters. If one of these is something you can actually drive — you *do* run Grafana, you *do* have a Windows endpoint on WinRM — say so through **Feedback** (§12) and name the panel. The matrix is a record, not a verdict on your environment.

A.7 The gate run that closed this pass

The removal pass ended with the gate green: **62 of 62 panels, 6 of 6 menus.** Getting there turned up three more defects and one fault in the gate itself.

Copy Run ID had never worked anywhere but localhost. *Edit → Copy Run ID* called the browser Clipboard API, which is gated on a **secure context**. The viewer is normally reached over plain `http` at a LAN address, where `navigator.clipboard` does not exist — and the failure was discarded, so the menu item did nothing at all: no copy, no error, no explanation. It worked in development on `localhost`, which *is* a secure origin, which is how it survived this long. It now falls back to a copy path that works on an insecure origin.

"Edit Test YAML" was the same panel under a second name. It opened the same editor as *Edit Testbed YAML* — same window, same file list — under a label that implied a second kind of file. There is one entry now.

The Latency Profiles panel asked for a profile that only exists in our lab. It shipped with `aaa_core_freeradius` as its starting selection and requested it the moment it opened, so on any install without that profile it reported a 404 for a name you never chose. It now starts on whatever profiles your server reports, and says plainly when there are none. Same shape as the `env1.yaml` default that made the Run button unusable from a wheel: our lab's identifier shipped as everyone's starting point.

And the gate accused a panel that had done nothing wrong. RADIUS Results was reported as reloading the app. It does not: probed on its own, five times over, it opened cleanly every time. The gate checks for a reload by reading a sentinel it set before the click, and treated *any* failure to read it as proof of navigation — but that read also fails when the page is merely BUSY, which a panel that builds a graph on open can easily be. It was reporting **"I could not ask"** as **"the app navigated"**: the more alarming of the two, and the untrue one. It now asks twice with a settle between, and separately recognises the app's own stale-chunk recovery — a deliberate reload after a new deploy — as its own outcome rather than as a panel misbehaving.

The last one is worth stating plainly because a test that cries wolf costs more than no test: the first version of this same gate reported "0 of 77 panels clean", and a later one reported 71 broken, both false. Every one of those numbers had to be thrown away. A gate is only useful while its failures are believed.

A.8 Editor panels open full, and two Activity Bar icons are gone

Panels you edit in now fill the window. The Synthetic Client editor opened at 422×402 on a 1600×1000 screen — a form you scroll to use — and the YAML Editor at 600×500. Both now open filling the viewport, as the Test Runner already did. The old sizes remain as the floor a smaller screen falls back to.

Two Activity Bar icons are removed, both because the build already answered what they offered:

- **Account** — there are no accounts here. This is a single-operator install with a bearer token: no sign-in, no profile, nothing the icon led to. It was also a duplicate, opening the very same sidebar view as the ⚙ beside it.
- **Quick Actions** — a shortcut bar whose verbs the **Run** menu already carries, in the same order: Stability Probe, Proof Positive, Negative Test, Run Tests, then Windows Audit and Cert Probe. Two routes to one set of actions earns its place in the internal build, where that bar sits beside the lab work it drives; in a single-operator install it is a second place to look for what the menu already offers.

Jobs got an icon that means something. It was a filled black square, which depicts nothing. The view is the queue of jobs TestPulse has started and their progress, and it already carries a badge counting the ones still running, so it is now an hourglass — work in flight, which is the one thing you look at that icon to learn.

The Activity Bar is eight icons now: Explorer, Search Runs, Compare Runs, Test Runner, Jobs, Copilot / AI, Anomaly, Settings.

Two menu items were checked at the same time and both work as intended: *Edit → Edit Synthetic Client Config* opens the client editor, and *Edit → Find in Run…* opens the Search view in the sidebar rather than a panel, which is why it looks like nothing floats when you pick it.

A.9 The gear opens the editor, and TACACS+ is drawn as itself

⚙ **opens your config editor, full size.** It used to switch the primary sidebar to the same editor, squeezed into a column capped at 480px — a document editor rendered in a gutter. It now opens the editor filling the window, and the same click closes it.

TACACS+ is its own device on the map. It was drawn as a RADIUS server, because the model had no role for it: `DeviceRole` listed `radius_server` and nothing else that fit, so TACACS+ borrowed it and the code that draws the map split the two apart again by checking whether `"tacacs"` appeared in the device name. Three things followed from that, all now fixed:

- A TACACS+ server **named something else** — `tac_plus`, say — failed that name check and was handed the RADIUS flow: an LDAP edge and a DNS edge that exchange never makes.
- The `switch → TACACS+` link was **typed** as RADIUS. The caption read "TACACS+ admin" while the data said `radius`, so anything reading the link rather than the label counted it as a RADIUS edge.
- Reachability took the RADIUS path, which asks Kubernetes about a deployment defaulting to `freeradius` — **the wrong server answering for this one**. TACACS+ is now probed on **TCP/49**, the port the exchange actually uses, the way the directory is probed on 389 and DNS on 53.

`DeviceRole.TACACS_SERVER` and `LinkType.TACACS` are real members now, and the map labels the device TACACS+. For a product whose core is RADIUS *and* TACACS+, one of the two should not have been a special case of the other.

A.10 Where the default topology comes from

On first run — before you have configured anything — the Topology panel draws **`testpulse/sample_testbed.yaml`**, which ships *inside* the installed package. It declares six devices — RADIUS (1812/1813, CoA 3799), TACACS+ (49), the switch, the directory, DNS, and the endpoint — plus a capture point for each wire. It also declares a DHCP **pool** (`scope`, `pool_start`, `pool_end`) so lease placement can be checked; that section carries no `ip`, so there is no DHCP server to draw and the map shows six nodes, not seven.

Every address in it is from the RFC 5737 documentation range, so every device reads **unreachable**. That is the truthful result on a machine that has none of them — not a failure, and not a mock pretending to be your network.

To draw YOUR devices, first hit wins:

```
1. export TESTPULSE_CONFIG=/path/to/my-testbed.yaml   an explicit choice
always wins
2. ./testbed.yaml                                     next to where you
run testpulse-api
3. ./env1.yaml  ./env-k8s-virtual-lab.yaml            existing checkouts
keep working
4. testpulse/sample_testbed.yaml                      the packaged
sample
```

Copy the sample, edit the addresses, and the same panel draws your network with real ping and latency per device. The ⚙ icon opens the editor for it.

A.11 Your runs now record the wire

This is the change that makes the evidence panels work. Until now a run made by the shipped suite carried no packet capture at all: the capture path needed Kubernetes and a script from our repo, neither of which exists in an install. Six panels were empty as a result — EAPOL Trace, Ladder Diagram, Live Trace, PCAP Analysis, Timeline Playback, RADIUS Results — and the PCAP sub-agent reported a *transport fault* on a perfectly healthy run, inferring a problem from its own missing evidence.

TestPulse now captures on **your** machine, filtered by the `pcap.capture_points` in **your** testbed YAML, for the duration of each test. Nothing asks Kubernetes anything.

Capture needs permission, so TestPulse asks before the run rather than failing during it. If it cannot capture, it says exactly why and what to do:

```
[capture] SKIPPED — dumpcap cannot capture as this user — grant it with
'sudo setcap cap_net_raw,cap_net_admin+eip $(which dumpcap)' or run the
API
as a user in a capture group
```

That reason is written into the run's evidence bundle, and the PCAP sub-agent now reports **`not_captured` with no cause family** instead of `transport_degradation`. *"You have not granted capture"* and *"your network is degraded"* are different statements, and only one of them was ever true here. Absence of evidence is not evidence of a fault.

Grant it and the same run records real frames:

```
[capture] radius-wire: dumpcap -i any -w …/radius-wire.pcap -P -f udp
port 1812 or udp port 1813
[capture] radius-wire.pcap — 308 bytes
```

With the wire recorded, the MAB observation goes from *not applicable* to a real result sourced from `radius-wire-pcap`, and the PCAP sub-agent reports healthy.

You can still bring your own: drop a `.pcap` into the run directory and the analysis reads it, whether TestPulse captured it or you did.

A.12 The Protocol panels now draw your run

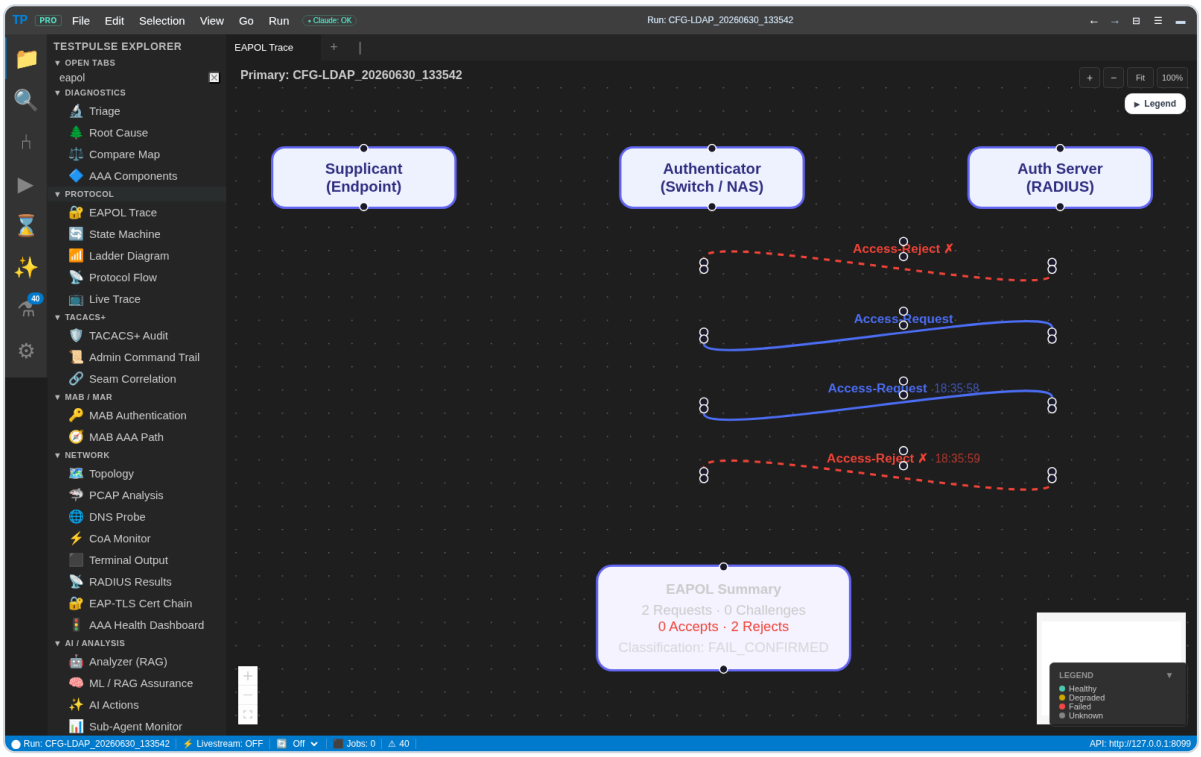
EAPOL Trace, the Ladder Diagram and the State Machine read the run's **timeline** — and the timeline is assembled from SERVER LOGS: `radiusd.log`, `dot1x.log`, `tac_plus`. If you have not given TestPulse SSH access to your RADIUS server there are no logs, so there was no timeline, so all three showed either nothing or a generic textbook diagram, for a run that plainly exchanged RADIUS.

TestPulse now builds a timeline **from your capture** when there are no logs to build one from. The wire carries the same exchange; it just needed reading.

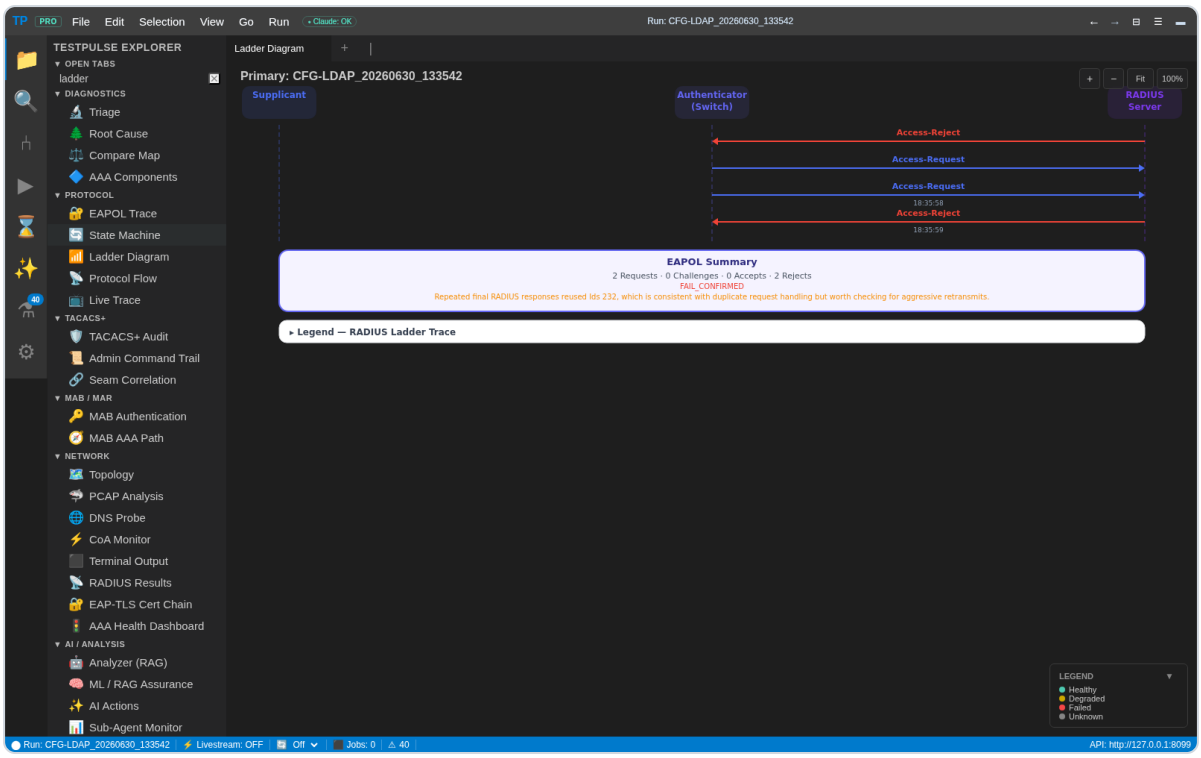
```
[evidence] timeline built from the wire — 4 RADIUS events
```

What that changes, on the same run:

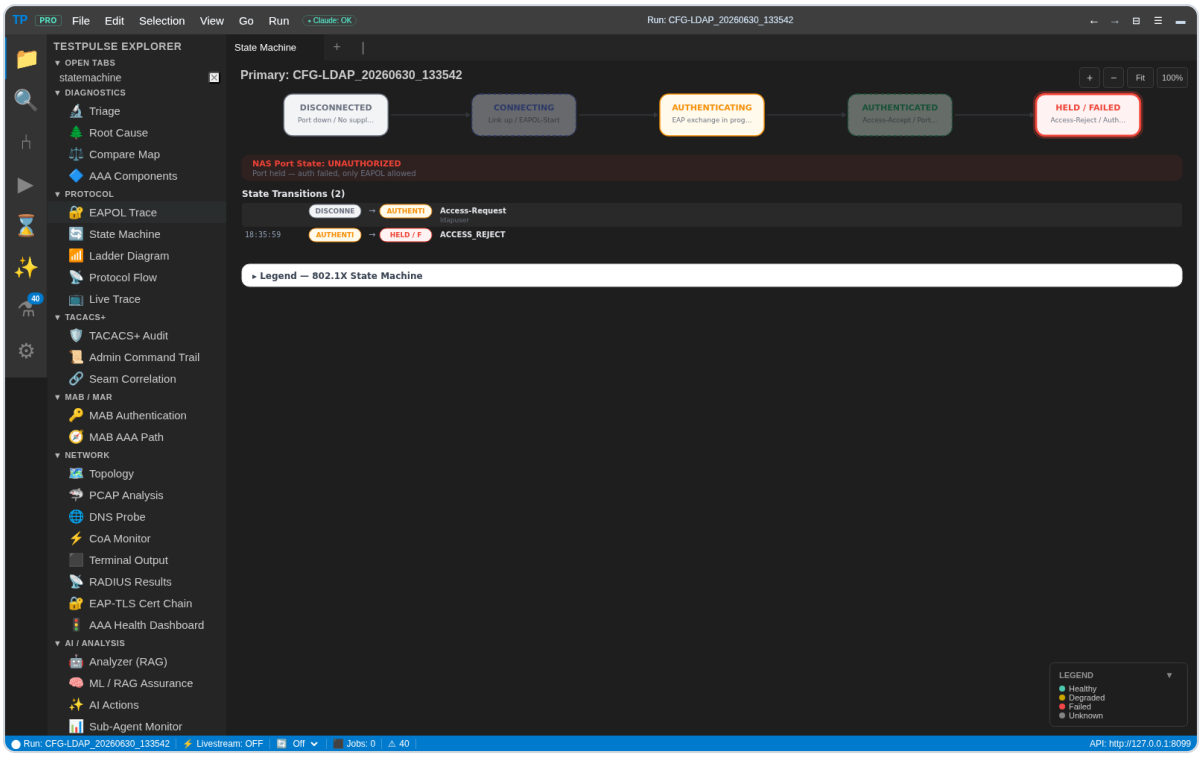
Panel	Before	After
EAPOL Trace	"No RADIUS/ EAPOL wire-trace events"	Supplicant → Authenticator → Auth Server, with each message
Ladder Diagram	same message	`Access-Request 17:52:29`, `Access-Reject 17:52:30`
State Machine	the five state boxes, no transitions	`State Transitions (2)` · `DISCONNECTED → AUTHENTICATING` · `AUTHENTICATING → HELD` · `NAS Port State: UNAUTHORIZED`



EAPOL Trace — Supplicant → Authenticator → Auth Server, every message drawn with its own timestamp, and the exchange summarised underneath.



The Ladder Diagram reads the same timeline as a sequence: two Access-Requests, two Access-Rejects, with the duplicate-Id finding stated beneath rather than left for you to spot.



The State Machine draws the five 802.1X states whether or not it has data — the transitions underneath are the part that comes from your run. `DISCONNECTED → AUTHENTICATING → HELD/FAILED`*, and the port left UNAUTHORIZED.*

A timeline built from logs is richer — the server explains its own reasoning — so it always wins. The wire-built one fills a gap; it never overwrites.

The State Machine was never broken, only starved. It looked like it worked, because it draws the five 802.1X states whether or not it has anything, and only the transitions between them come from your run. A panel that renders the same picture with data and without it is the hardest kind to catch.

Timestamps on these diagrams also read as **`2026`** on every message — the code took the last word of "Thu Aug 27 17:52:29 2026", which is the year. They now show the time.

A.13 Recording the wire, and opening it in Wireshark

Two things TestPulse cannot install for you. `pip install 'testpulse[pcap]'` brings the Python side. Recording the wire and breaking it down per protocol also need **Wireshark** — specifically its `dumpcap` and `tshark` binaries:

```
sudo apt install wireshark tshark          # Debian / Ubuntu
sudo dnf install wireshark-cli             # RHEL / Fedora
brew install --cask wireshark              # macOS
#  Windows: Wireshark + Npcap — https://npcap.com
```

Frame and byte counts work without any of it. The per-protocol breakdown, and capture itself, do not.

Capture needs privilege. TestPulse asks before each run rather than failing during it, and when the answer is no it tells you the exact command:

```
[capture] SKIPPED — dumpcap cannot capture as this user — grant it with
'sudo setcap cap_net_raw,cap_net_admin+eip $(which dumpcap)' or run the
API
as a user in a capture group
```

Opening a capture. The PCAP Analysis panel's *Open in Wireshark* launches it on your desktop with a display filter already applied — for a RADIUS device, `-Y radius`. Useful filters once you are in:

Filter	Shows
`radius`	the whole RADIUS conversation
`radius.code == 1` / `== 2` / `== 3` / `== 11`	Access-Request / Accept / Reject / Challenge
`radius.code == 4 || radius.code == 5`	Accounting-Request / Response
`radius.code >= 40`	CoA and Disconnect
`eap`	the EAP conversation inside RADIUS
`tcp.port == 49`	TACACS+
`eapol`	802.1X on the LAN — only present if you captured at the switch

That last row is worth understanding. **EAPOL is layer 2, between the supplicant and the switch.** Capturing on the machine running TestPulse sees the RADIUS exchange but never EAPOL, because EAPOL never reaches it. To see EAPOL you must capture at the switch — a mirror/SPAN port — and drop the resulting `.pcap` into the run directory, where the analysis picks it up like any other.

If Wireshark does not appear. A GUI program needs a desktop to draw on, and the API does not always have one — started from a service manager, a bare `ssh` session, or any launcher with a scrubbed environment, it has no `DISPLAY` at all, and Wireshark exits the instant it starts. That used to report as *"Wireshark could not be launched"*, which sends you to check an install that is perfectly fine.

TestPulse now finds the desktop itself. An explicit `DISPLAY` always wins — a deliberate choice is never overridden. Otherwise it asks `who` which display you are logged in on (this is what finds an xrdp session, which lands on `:10` and up), and failing that reads the X sockets actually present on the host. The answer tells you where the window was sent and how that was decided:

```
"launched": true, "display": ":10",
"display_source": "the display this user is logged in on (who)"
```

Read that line when a capture "does not open". **A window opened on the console while you are sitting at a remote session is indistinguishable from one that never opened** — and `display` is what tells the two apart.

Two cases it cannot solve for you, both with a one-line fix:

What you see	Why	Fix
`display_source: "no desktop found"`	the host has no graphical session at all — a headless server	run TestPulse where you are sitting, or copy the pcap and open it locally
It opened, but on the wrong screen	more than one desktop, and the detected one is not yours	start the API with the display you want: `DISPLAY=:10 testpulse-api`

`who` lists the displays in use, and you can always bypass the button entirely:

```
who                                                    # which
display is mine?
DISPLAY=:10 wireshark -r <run-dir>/radius-wire.pcap -Y radius
```

A.14 The testbed YAML — declaring your nodes

One file describes your network. Point at it with `TESTPULSE_CONFIG`, or save it as `./testbed.yaml` where you run `testpulse-api`; the ⚙ icon edits it.

Each top-level section is one device. **The KEY names the role** — that is how the Topology map knows what it is drawing and how the diagnosis knows which plane a device belongs to:

Section key	Drawn as	In the AAA path as
`radius`, `freeradius`	RADIUS	the authentication server
`tacacs`, `tac_plus`	TACACS+	device administration, TCP/49
`switch`	Switch	the NAS / authenticator
`nas`, `hostapd`, `hostapd_nas`	NAS	an explicit authenticator, when it is not the switch
`ldap`, `ad`, `openldap`, `directory`	AD / LDAP	the identity lookup
`dns`	DNS	name resolution in the AAA path
`dhcp`, `isc_dhcp`	DHCP	address acquisition after authorisation
`endpoint`	Endpoint	where the path starts
`firewall`	transport	in the path between endpoint and server
`wireless_controller`, `access_point`	wireless	for 802.11 deployments
`ntp`, `syslog`, `ca`, `policy_engine`	supporting	timing, logging, certificates, policy

A section TestPulse does not recognise still draws. Give it an `ip` and it appears with role `unknown` rather than being dropped — your network is the source of truth for what exists, not our list of names.

A minimal file with two servers and the switch between them:

```
env:
  domain_pack: aaa_core
  vendor: cisco_ios_xe        # or aruba, juniper, generic …

radius:
  ip: 10.0.0.10
  port: 1812
  accounting_port: 1813
  coa_port: 3799
  secret_env: TESTPULSE_RADIUS_SECRET     # NAME a variable, do not
inline the secret
  test_user: alice
  test_password_env: TESTPULSE_TEST_PASSWORD

tacacs:
  ip: 10.0.0.11
  port: 49
  secret_env: TESTPULSE_TACACS_SECRET

switch:
  ip: 10.0.0.21
  vendor: cisco_ios_xe
```

```
  user_name: admin
  password_env: TESTPULSE_SWITCH_PASSWORD

pcap:
  capture_points:
    - device: radius-wire
      bpf: "udp port 1812 or udp port 1813"
    - device: tacacs-wire
      bpf: "tcp port 49"
```

On `vendor`. It selects the command vocabulary for a device class — how a Cisco IOS-XE switch is asked for its port state differs from an Aruba or a Juniper. `generic` is the safe default and simply asks for less.

On secrets. Prefer `*_env`: name an environment variable and keep the value out of the file. TestPulse has no built-in defaults for shared secrets — if one is missing it names the variable to set rather than sending a guessed secret to your server, because a wrong secret produces a confusing auth failure on YOUR box.

On `pcap.capture_points`. Each entry is one thing worth recording: a name and a BPF filter, optionally an `iface`. This is what `GET /v1/pcap/devices` reports and what the capture records during a run. Both `device:` and `name:` are accepted for the key.

A.15 Two ways to bring your own AI — a subscription key, or your own AWS

TestPulse never ships an AI subscription. You bring one, and there are **two different kinds of credential** to bring. They are not interchangeable, and which you choose decides where your evidence goes and what the product can do.

	Subscription key	**Your own AWS (Bedrock)**
Credential	a bearer key pasted into ⚙	AWS creds — env, `~/.aws`, or an **IAM role**
Install	`pip install 'testpulse[ai]'`	`pip install 'testpulse[aws]'`
Providers	Anthropic, OpenAI	anything Bedrock offers
Reasoning		
Embeddings for the run corpus	OpenAI only (`text-embedding-3-small`)	Titan
Where run text goes	the vendor's API	your own AWS account
On EC2 in your VPC	n/a	**no key to store — the instance role is the credential**

The row that surprises people is embeddings. Reasoning and embeddings are different services, and not every provider offers both:

- **Anthropic publishes no embedding model at all.** An Anthropic key gives excellent per-run diagnosis and cannot build the corpus.

- **OpenAI does** — `text-embedding-3-small`, which takes a `dimensions` argument and returns TestPulse's 1536 natively. Set `TESTPULSE_EMBED_PROVIDER=openai` and the key you already saved is used for both.
- **Bedrock does NOT host OpenAI's embedding models.** "OpenAI via AWS" is not a route: Bedrock offers Amazon Titan and Cohere Embed. Titan returns 1024 dimensions, which TestPulse pads to 1536.

What "the corpus" is, and when it matters

Every run you keep is embedded and stored **locally, on your machine**. That is what powers Analyzer (RAG), Similar Runs, and cross-run comparison: asking *"have I seen this failure before?"* rather than *"what happened in this one run?"*. Per-run diagnosis does not use it — that is grounded in the run's own evidence bundle and works with any provider.

Check what you are actually running:

```
grep 'embed: provider' <your API log> | tail -1
#  provider=sha256_fallback (non-semantic)  ← similarity is meaningless
#  provider=sentence_transformers            ← real, local, free
#  provider=bedrock_titan                    ← real, in your AWS account
```

The fallback is deliberate: TestPulse keeps working without an embedding provider rather than refusing to start. It is honest about it in the log, and `last_embed_provider()` reports it so the RAG loop can be audited — but a hash is not a meaning, and two runs about the same failure will not be near each other.

Install `[rag]` whichever provider you pick. It carries `chromadb`, the vector store that does the actual cosine search. Without it TestPulse falls back to matching on cause family and recency and returns a FIXED similarity of 0.50 for every result — a ranked-looking list that ranks nothing. The embedding provider and the vector store are two separate requirements and you need both.

Mind the footprint. Both numbers below are measured, whole-virtualenv, on a clean install of this wheel:

Install	Virtualenv	Why
`testpulse[aws,rag]`	**539 MB**	the store, no local model
`testpulse[rag-local]`	**5.3 GB**	adds sentence-transformers, which pulls torch

Ten times the size for embeddings with no key and no cloud. Worth it if that is what you want; not worth it if OpenAI or Bedrock is doing the embedding — which is why they are two extras and not one, and why `[all]` deliberately leaves `rag-local` out.

Option 1 — the key you already have (OpenAI)

```
pip install 'testpulse[ai,rag]'
export TESTPULSE_EMBED_PROVIDER=openai      # the BYOK key in ⚙ is
reused
```

Option 2 — local model, no key, no cloud

```
pip install 'testpulse[rag-local]'          # the store AND a local
model (~5.3 GB)
export TESTPULSE_EMBED_PROVIDER=local
```

Free, private, nothing leaves the machine. Downloads a model on first use. The size is torch, which sentence-transformers requires.

Option 3 — your own AWS account

```
pip install 'testpulse[aws,rag]'          # boto3 — the keys do NOTHING
without it
export TESTPULSE_AI_PROVIDER=bedrock      # reasoning goes to Bedrock,
not OpenAI
export TESTPULSE_RAG_MODEL=us.deepseek.r1-v1:0
export TESTPULSE_EMBED_PROVIDER=bedrock   # only if you also want cross-
run search
export AWS_DEFAULT_REGION=us-east-1
# credentials: env vars, ~/.aws/credentials, or — best — an IAM role
```

Three things catch people here, in this order.

`[aws]` is not optional, and nothing warns you. boto3 is not in the base wheel. Save the keys without it and every field reads `set`, the panel says `stored`, and reasoning still refuses — the same "looks configured and is not" trap §7 names for Anthropic, on a path that used to name it nowhere.

`TESTPULSE_AI_PROVIDER` defaults to `openai`. Keys, region and model can all be correct and reasoning still goes to OpenAI, because the provider selector is a separate setting from the credentials. Setting the keys is not choosing to use them.

The AWS settings need a restart. The panel marks them **↻ restart to apply** and means it: boto3 clients are built at start-up. The Anthropic and OpenAI keys take effect on the next call; these do not. Restart `testpulse-api` and the log says what it picked up:

```
INFO:testpulse.settings:applied 7 stored setting(s): AWS_DEFAULT_REGION,
AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, TESTPULSE_ML_ROUTING_MODE,
TESTPULSE_AI_PROVIDER, TESTPULSE_EMBED_PROVIDER, TESTPULSE_RAG_MODEL
```

Confirm it end to end by asking the Copilot anything about a run — the answer names the provider and model that produced it:

```
"provider": "deepseek_r1_bedrock", "model": "us.deepseek.r1-v1:0"
```

How to tell a real similarity score from a fake one

A number between 0 and 1 always appears. What makes it worth anything is SEPARATION — the same failure described differently should score HIGH, an unrelated failure LOW. Measured on a clean install with the local model:

```
"TACACS authentication failed, TCP/49 retransmits"
    vs "TACACS auth failed with packet loss on port 49"   →  0.96   same
meaning
    vs "DHCP lease was never offered to the endpoint"     →  0.29
different
```

Those two sentences share almost no words, so a keyword search would rank them apart; the model ranks them together because it reads meaning. That gap is what makes "have I seen this before?" answerable.

Compare the two broken states, where the gap disappears:

State	What every score looks like
no embedding provider	random — a hash gives one changed character a completely different vector
no vector store	exactly `0.50` for everything — no cosine search ever runs
both present	0.96 vs 0.29 — a usable threshold

All three are configured in Environment / Settings

You do not have to use environment variables. Open **Edit → Environment / Settings…** from the menu bar — *not* the ⚙ in the Activity Bar, which opens the config editor (A.9). The **AI/RAG** group holds every one of them, and all are writable at any tier because they are YOUR credentials:

Setting	For
`ANTHROPIC_API_KEY`	Anthropic reasoning (no embedding model exists)
`OPENAI_API_KEY`	OpenAI reasoning **and** embeddings
`AWS_ACCESS_KEY_ID` / `AWS_SECRET_ACCESS_KEY` / `AWS_DEFAULT_REGION`	Bedrock in your own account — leave the keys blank to use an IAM role
`TESTPULSE_EMBED_PROVIDER`	which of `local` / `openai` / `bedrock` builds the corpus

Keys are write-only — stored locally, sent only to that provider, never to BMI2. The AWS ones need a restart (boto3 clients are built at startup); the vendor keys take effect on the next call.

The IAM policy needs `bedrock:InvokeModel` on the models you use (`amazon.titan-embed-text-v2:0` for embeddings). On an EC2 instance in your VPC, attach the role and there is **no key anywhere** — not in TestPulse, not in a file. Add a Bedrock **VPC endpoint** and the traffic never crosses the public internet, which is usually the point of choosing this route for AAA evidence.

Two things to know before you switch

Switching provider invalidates the corpus, and TestPulse now says so. Every indexed run records which model embedded it. Query with a different one and the response tells you rather than returning confident-looking numbers:

```
"embedding_provider": "bedrock_titan", "semantic": false,
"warning": "Query embedded with bedrock_titan, but indexed runs were
embedded
            with sentence_transformers — vectors from different models
are not
            comparable, so these scores carry no meaning. Re-index the
corpus
            after changing provider."
```

Pick a provider before you accumulate history, or re-index when you change.

Titan v2 returns 1024 dimensions; TestPulse stores 1536 and pads the difference. Harmless within one provider, and another reason a corpus should not straddle two.

A.16 Collecting from your own devices

The evidence bundle is built from device logs. `radiusd.log` from your RADIUS server, `tac_plus.log` from TACACS+, `dot1x.log` from the switch. The run timeline is assembled from those, and the timeline is what EAPOL Trace, the Ladder Diagram and the State Machine draw. Without them the diagnosis is working from a fraction of the evidence it was designed to read.

TestPulse can fetch them over SSH. **It does nothing until you ask**, using credentials you put in your own testbed YAML — nothing is collected implicitly, and no credential is stored anywhere but your file.

1. Declare the device and how to reach it:

```
tacacs:
  ip: 10.0.0.11
  port: 49
  user_name: admin                          # the SSH user on the device
  password_env: TESTPULSE_TACACS_SSH_PASS   # name a variable, never
inline it
  log_path: /var/log/tac_plus/tac_plus.log  # where the log lives on
that device
```

2. Check what is collectable before you run:

```
curl -H "Authorization: Bearer $TESTPULSE_API_TOKEN" \
     http://127.0.0.1:8000/v1/collect/targets
```

It answers per device, and says what is missing rather than failing quietly:

```
tacacs   10.0.0.11   collectable=True
radius   10.0.0.10   collectable=False  declare user_name and
password_env …
```

3. Collect into a run:

```
curl -X POST -H "Authorization: Bearer $TESTPULSE_API_TOKEN" \
     http://127.0.0.1:8000/v1/collect/<run-id>
```

A partial collection is a normal result — you get what was collected and what was not, each with its reason, rather than one verdict covering both.

The device must run an SSH server that this host can reach. That is the whole prerequisite, and it is worth stating plainly because it is where collection most often stops. A RADIUS or TACACS+ daemon running on a normal server meets it. A containerised lab usually does not — containers rarely run `sshd`, so there is nothing to connect to no matter how the credentials are written. If `collectable` says `True` but every command comes back failed, check that first:

```
nc -vz <device-ip> 22        # is anything listening?
```

Where SSH is not available, import the logs instead — the diagnosis reads them identically, and nothing downstream can tell the difference:

```
kubectl -n <ns> logs <pod> > <run-dir>/tac_plus.log     # containerised
scp user@device:/var/log/radiusd.log <run-dir>/          # anything else
```

Per-device paths and switches are in A.19. This section covers the TACACS+ and switch collection path; A.19 covers all seven device classes, where each log is fetched from, and how to turn them on individually.

Name the vendor if the device is not a Cisco. The command set is selected from `switch.vendor`, and a MikroTik answering Cisco IOS commands rejects every one of them:

```
switch:
  vendor: mikrotik_chr    # RouterOS command set; omit for Cisco IOS
```

This is your decision to make. TestPulse reaching into AAA infrastructure over SSH is a real security question, and you are the only party who can answer it: you own the devices and hold the credentials. Earlier builds simply left the capability out, which did not make anyone safer — it just left the bundle thin. If you would rather not, feed logs you already have:

```
testpulse-diagnose diagnose --protocol radius --file /path/to/
radiusd.log
```

A.17 Making packet capture work — one command

Capture needs `CAP_NET_RAW`, a kernel capability no package manager grants. Rather than print a command and leave you to reason about Linux capabilities, TestPulse ships one:

```
testpulse-capture-setup           # report what works, and what does
not
testpulse-capture-setup --grant   # grant it (prompts for sudo)
```

It reports the tool it found, its capability bits, your group membership, and then **actually tries to capture** — because capability bits and group membership can both look correct and still not work, so it asks the kernel rather than inferring:

```
  tool          : /usr/bin/dumpcap
  capabilities  : /usr/bin/dumpcap cap_net_admin,cap_net_raw=eip
  user          : laceth
  capture group : no
  capture test  : OK — capture works
```

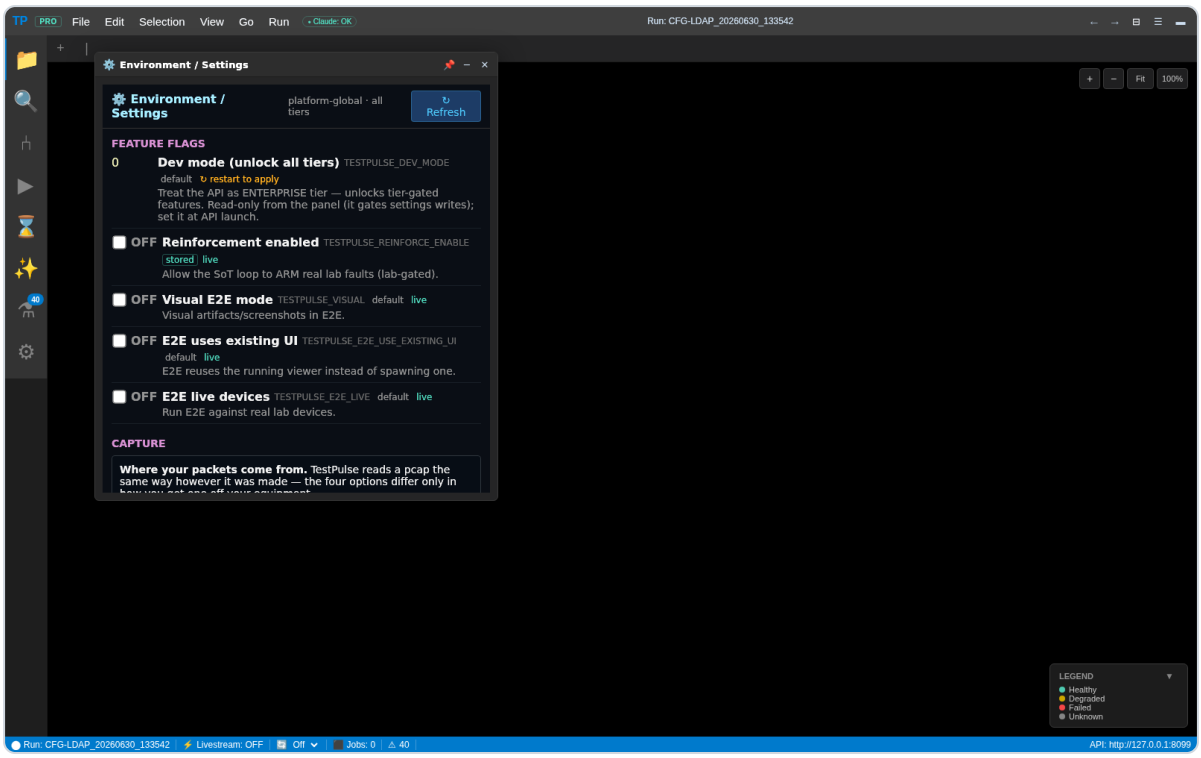
It privileges **dumpcap** rather than tcpdump — that is what Wireshark's own packaging does, and dumpcap is small enough that granting it a capability is a much narrower thing to do. `sudo dpkg-reconfigure wireshark-common` is the same mechanism through Debian's own prompt, if you prefer that route.

Or skip capture entirely. Drop a `.pcap` into the run directory — one you took at a mirror port, or with Wireshark on another host — and TestPulse analyses it exactly the same. That is also the only way to see EAPOL, which is layer 2 and never reaches the machine running TestPulse (see A.13).

A.18 Capturing on a router or a switch — the four ways

A.17 covers capture on the machine running TestPulse. The harder question is how to get packets off equipment TestPulse is not running on, and the answer differs per device. **Environment / Settings → Capture** now names the four options, says which of them work on your host right now, and hands you the exact commands for the ones you have to run yourself.

Open it from the menu bar: **Edit → Environment / Settings…**



Environment / Settings, scrolled to the top. Feature flags first — each one naming the variable behind it and whether it takes effect live or needs a restart — then CAPTURE, which is where the four sources below are chosen.

Source	How it works	Reach for it when
`host_nic`	Your switch mirrors the AAA port to a NIC on this host; TestPulse records from that NIC	**Preferred.** Vendor-neutral, and TestPulse does the recording — nothing to export
`cisco_epc`	TestPulse SSHes to the switch, runs `monitor capture`, pulls the file back	You cannot run a mirror cable, and the switch is IOS-XE
`routeros_sniffer`	The router streams packets to this host as TZSP on UDP/37008	MikroTik / RouterOS
`manual`	You capture however you like and drop the `.pcap` in the run directory	Always available; the fallback for any device

Each source reports its own readiness. A source is marked ● `ready` or ○ `blocked`, and a blocked one says specifically why — "the kernel refused: run testpulse-capture-setup --grant", or "declare user_name and password_env under switch: to let TestPulse drive EPC". A generic failure would leave you guessing; these are the actual missing piece.

Show steps gives you commands, not prose. For the two sources that configure your own equipment, the panel prints the exact CLI with your own addresses already substituted:

```
monitor capture TESTPULSE_UP interface <uplink> both
monitor capture TESTPULSE_UP match ipv4 host 10.0.0.10 any
monitor capture TESTPULSE_UP buffer size 10
monitor capture TESTPULSE_UP start
```

That is deliberate. Mirroring a port or starting an on-box capture is a change to **your** network, on your maintenance window — TestPulse hands you the commands rather than reaching in and reconfiguring your switch. Where it can capture without touching your equipment (a host NIC), it just does it.

Settings in this group

Key	What it is
`TESTPULSE_CAPTURE_SOURCE`	Which of the four you are using
`TESTPULSE_CAPTURE_INTERFACE`	Host NIC receiving the mirror, e.g. `eth1`. Blank = any
`TESTPULSE_CAPTURE_STREAM_TARGET`	Address RouterOS streams TZSP to
`TESTPULSE_CAPTURE_EPC_UPLINK`	Switch interface facing RADIUS, e.g. `Te1/1`
`TESTPULSE_CAPTURE_EPC_ACCESS`	Switch interface facing the endpoint, e.g. `Gi1/0/2`

A caveat worth stating before you rely on EPC. Its limits are platform-specific — buffer size, which interface types are supported, and on some platforms only CPU-punted traffic is captured rather than everything the hardware forwards. Verify it against your own platform before making it your only capture. A mirror port has none of those caveats, which is why it is the recommendation.

A.19 Telling TestPulse where your logs live

The evidence bundle is built from device logs, so where those live is the one thing TestPulse cannot work out for itself. Earlier builds tried: RADIUS was looked for at `/var/log/freeradius/radius.log`, then `/var/log/radius/radius.log`, then a third path — a guess dressed as a default, and wrong the moment you package your AAA stack any other way. Only TACACS had a configurable path at all.

Every device class now declares its own, in **Environment / Settings → CAPTURE → Where each device's log is fetched from** (menu bar: **Edit → Environment / Settings…**), or directly in your testbed YAML:

```
radius:
  ip: 10.0.0.10
  user_name: admin
  password_env: TESTPULSE_RADIUS_SSH_PASS
  log_path: /var/log/freeradius/radius.log   # where YOUR server writes
it
  collect: true                              # off until you turn it on
```

Seven device classes, each with its own path and its own switch:

Device	Section	Lands as	What it proves
TACACS+ server	`tacacs`	`tac_plus.log`	authen / author / acct decisions
RADIUS server	`radius`	`radiusd.log`	Access-Accept/Reject, accounting, CoA
Router / switch (NAS)	`switch`	`dot1x.log`	802.1X port state, EAPOL, AAA client
LDAP / AD	`ldap`	`ldap.log`	the identity lookup behind a policy decision
DNS resolver	`dns`	`dns.log`	name resolution in the effective AAA path
NAC / policy engine	`nac`	`nac.log`	the enforcement decision after auth
Supplicant endpoint	`endpoint`	`eapol.log`	the client side of the EAP exchange

The file is renamed on arrival, deliberately. Whatever your daemon calls it, it lands under the name the parsers read. They must not care how you packaged your syslog.

`collect` is separate from `log_path` on purpose. Switching a noisy or slow device off should not mean deleting the path you spent an afternoon finding.

Every failure names itself. A device that cannot be collected says which thing is missing, rather than leaving you with a thin bundle and no explanation:

```
tacacs     collected  113B   custom nested path honoured
router     collected    0B   the file exists on the device but is empty
ldap       error        0B   /opt/logs/slapd.log does not exist on
10.0.0.12
dns        error        0B   admin cannot read /var/log/named.log on
10.0.0.13
nac        skipped      0B   collection is off for this device
endpoint   skipped      0B   no SSH on 10.0.0.60:22
```

Note the distinction between the third row and the sixth: **a failed fetch leaves nothing behind**. An empty file in the run directory would read as "collected, and the device logged nothing" — the reassuring interpretation, and the wrong one. Only a file that genuinely exists and is genuinely empty appears as 0 bytes.

Where it ends up. A collected log is parsed into the evidence bundle, and the bundle is what gets embedded into the RAG corpus — logs are not embedded raw. So a log fetched from a path you declared reaches cross-run search by the same route as everything else.

```
GET  /v1/log-sources?probe=true
PUT  /v1/log-sources/<device>       {"log_path": "...", "enabled": true}
POST /v1/log-sources/collect/<run>
```

This needs an SSH server TestPulse can reach (see A.16). A containerised lab usually has none — import the logs instead.

A.20 What is in the box — dependencies and the SBOM

Every release ships **`SBOM.md`** and **`sbom.cdx.json`** (CycloneDX 1.5), both covered by `SHA256SUMS`. You are installing software that will hold your AAA credentials and SSH into your infrastructure; `SHA256SUMS` proves the bytes match what was built, and the SBOM says what those bytes are made of.

Required — 14 packages. The wheel does not run without them: `fastapi`, `starlette`, `uvicorn`, `pydantic`, `PyYAML`, `PyJWT`, `httpx`, `requests`, `cryptography`, `paramiko`, `python-dotenv`, `python-multipart`, `pytest`, and `tomli` on Python 3.10. `paramiko` is there because device collection is core, not an add-on; `pytest` because the Test Runner executes the shipped suite with it.

Optional extras, each unlocking one feature:

Install	Adds	For
`testpulse[ai]`	anthropic, openai	AI narration with your own key
`testpulse[aws]`	boto3	Bedrock / SageMaker in your own account
`testpulse[rag]`	chromadb	the vector store for cross-run search (~539 MB)
`testpulse[rag-local]`	chromadb, sentence-transformers	embeddings with no key and no cloud (~5.3 GB)
`testpulse[pcap]`	scapy, dpkt	richer pcap decode
`testpulse[mcp]`	mcp	the MCP server surface
`testpulse[postgres]`	psycopg2-binary	Postgres instead of SQLite
`testpulse[windows]`	pywinrm, python-evtx	Windows endpoint collection

External command-line tools. TestPulse shells out to these, and **no dependency resolver will fetch them** — which is exactly why the SBOM lists them. Each is optional; the row says what stops working without it.

Tool	Install with	Needed for
`dumpcap`	`wireshark`	packet capture on a host interface (preferred)
`tshark`	`tshark`	pcap decode when scapy is absent
`wireshark`	`wireshark`	opening a captured pcap in the GUI

Tool	Install with	Needed for
`tcpdump`	`tcpdump`	capture fallback where dumpcap is unavailable
`radtest` / `radclient`	`freeradius-utils`	the RADIUS probes in the shipped suite
`eapol_test`	`wpa_supplicant`	EAP negotiation without a real supplicant
`ssh`	`openssh-client`	device log collection
`openssl`	`openssl`	certificate chain and EKU inspection
`setcap`	`libcap2-bin`	`testpulse-capture-setup --grant`

What is deliberately absent, and worth saying plainly:

- **No telemetry or analytics package.** TestPulse reports nothing to BMI2.
- **No bundled AI account.** The AI extras talk to *your* provider with *your* key.
- **No vector store by default** — `chromadb` arrives only with the `rag` extra.
- **The viewer is bundled, not fetched.** The React app is compiled into the wheel, so rendering the UI makes no request to a CDN.

Addendum B — Error reference

Every code below is a string this build actually emits. It is not a catalogue of what an API of this shape might return: each was read out of the shipped package, so if you see something here it exists, and if you see something in the wild that is not here, that is worth reporting.

Errors are grouped by **what you do about them**, not by where they come from. A 403 you cannot fix and a 400 you can are different situations regardless of their numbers.

B.1 The API refuses to start

What you see	Why	Fix
`FATAL: no TESTPULSE_API_TOKEN and no TESTPULSE_JWT_SECRET configured — refusing to start in unauthenticated open mode`	no credential	Set `TESTPULSE_API_TOKEN` (§2). `TESTPULSE_ENV=dev` allows open mode in development only
`Address already in use`	something holds `:8000`	Set `TESTPULSE_PORT`

Both are deliberate. An unauthenticated API bound to `0.0.0.0` is a mistake you get to make exactly once, so this build will not start rather than let you.

B.2 HTTP status codes

Code	Means	Where it comes from
401	missing or wrong bearer token	any `/v1/*` route. `/healthz` is the unauthenticated probe
403	your plan does not include this	see the entitlement slugs in B.3
404	run, artifact or endpoint absent	also *"This endpoint is not part of this build"* — a surface this download does not carry (§10), not a fault
400	the input was not usable	`no_events_parsed`, `Invalid scan directory`
409	conflict with something already running	a capture is already active
413	payload too large	`log_too_large`, `pcap_too_large`
422	request body failed schema validation	usually a malformed field, not a missing one
429	daily quota reached	**free tier only** — see B.4
500	internal error	**please report it.** A 500 is a defect in this build, never a statement about your network
502 / 503	an AI provider was unreachable	`reasoning_unreachable`, and your evidence pipeline is unaffected

A 404 is not always absence. Three different things answer 404: a run that does not exist, an artifact a run never produced, and an endpoint this build deliberately omits. The third carries the hint *"This endpoint is not part of this build"*, which is the one to read before assuming something broke.

B.3 Machine-readable `error` slugs

Every JSON error carries a stable `error` key. Match on that, never on the prose — the sentence may be improved, the slug will not change.

Entitlement — your plan, not your setup

`protocol_not_entitled` · `pcap_intelligence_not_entitled` · `ai_reasoning_not_entitled` · `enterprise_required` · `premium_required` · `device_limit_exceeded`

Input — something about what you sent

Slug	Means
`no_events_parsed`	a summary log, not raw output. Use `radiusd -X` or the tac_plus server log (§3)
`unsupported_protocol`	`--protocol` was not one of the seven

Slug	Means
`log_too_large` / `pcap_too_large`	over the size cap; the response names the limit
`pcap_file_not_found`	the path does not resolve inside the run directory
`pcap_empty_capture`	the file exists and holds zero frames — a finding, not a gap
`invalid_evidence_payload`	the imported bundle did not match the contract
`unreadable`	the file exists but could not be parsed

Config — your testbed YAML

Slug	Means
`config_file_not_found`	`TESTPULSE_CONFIG` points at nothing
`config_parse_error`	the YAML is malformed; the message names the line
`secret_env_var_not_set`	a `*_env` key names a variable you have not exported. **TestPulse names the variable rather than guessing a secret** — a wrong shared secret produces a confusing auth failure on YOUR server

AI / BYOK — narration only; the diagnosis is unaffected

Slug	Means
`byok_key_not_configured`	no key. Add one in ⚙ Environment / Settings (§7)
`unsupported_provider`	the provider name is not one this build routes to
`narration_provider_error`	the provider answered with an error — usually the key or its credit
`reasoning_unreachable`	the provider could not be reached at all
`reason_request_failed`	the request was made and rejected

Capture and approval

`capture_consent_required` · `no_active_capture` · `not_approved` · `irreversible` · `summary_required` · `not_found`

`not_approved` and `irreversible` are the human-in-control gates: an action that changes your equipment waits for you, and one that cannot be undone says so before it runs.

B.4 429 `quota_exceeded` — you should never see this on Pro

The free tier caps `/v1/diagnose` at ten runs a day. **Pro has no run limit, no device limit, and full AI reasoning**, and an unlicensed Pro install now defaults to Pro entitlements rather than free.

If a Pro install ever returns 429, that is a bug and worth reporting with the output of `testpulse-diagnose status`, which prints the plan actually in force.

B.5 Console messages from the runner and the capture path

These are not HTTP errors. They appear in the Runner terminal and in the run's `runner_output.json`, and several of them are the product telling you something important about the *quality* of a result rather than reporting a failure.

Message	What it means
`[runner] SKIPPED … the test declined to run, which is not a pass`	The case skipped — usually a missing credential or tool. Recorded **INCONCLUSIVE**, never PASS. A skipped test is not a passing test
`[runner] NO TESTS RAN … nothing was selected`	The target matched no test
`Invalid scan directory`	The scan path is outside the directories this build will read
`[capture] SKIPPED — dumpcap cannot capture as this user`	Run `testpulse-capture-setup --grant` (A.17). Recorded as `not_captured` with **no cause family** — you have not granted capture, which is not the same as your network being degraded
`[capture] surface capture skipped: …`	An evidence surface could not be written; the run itself is unaffected
`[evidence] timeline built from the wire — N RADIUS events`	No server logs, so the timeline came from the capture (A.12). Informational

B.6 AI reasoning messages

Message	What to do
`no reasoning provider available in mode=…`	A provider is selected but unusable. The message names which half is missing
`boto3 is not installed — pip install 'testpulse[aws]'`	AWS credentials with no SDK — the combination that looks configured and is not
`no AWS credentials found (keys, AWS_PROFILE, or ~/.aws/credentials)`	The SDK is present and no credential resolves. An instance role counts
`A key is configured but the provider SDK is not installed: pip install 'testpulse[ai]'`	Same trap on the Anthropic/ OpenAI path
`Wireshark is installed, but the TestPulse API has no desktop to open it on`	No `DISPLAY`. TestPulse now detects one; when it cannot, see A.13

B.7 The two warnings that are not errors

These appear **inside a Copilot answer**, appended after it. They are TestPulse disagreeing with its own AI, and they are the most important messages in this appendix.

Warning	What happened
`⚠ Evidence check — this claim is not supported by the run.`	The answer asserted a cause family the run's own evidence does not carry — including on a run recorded PASS with no ranked causes
`⚠ Citation check — a cited panel does not show this.`	The answer cited a panel for a fault that panel's data does not contain. A citation names a source; it does not make the source agree

Neither rewrites the answer. You are shown the model's words and the contradiction, and you decide. Silently correcting a model would teach you to trust it, which is the opposite of what an evidence product should do — and it would hide exactly the behaviour a human reviewer needs to see.

Both were built after a real incident: asked what happened to a run that had **passed**, the Diagnoser reported a TACACS+ transport failure, named `transport_degradation`, and attributed it to an OTel trace that showed 8 spans, 0 errors, 0 timeouts and 31 ms p95 — while the viewer helpfully opened that panel beside the claim. If you see one of these warnings, the answer above it is suspect and the run's own panels are the truth.

© 2026 BMI2 — Tarik Antar Riley. All rights reserved. Proprietary & confidential.