ZYVOR · NETRA

Network observability and emergency control, without CNI lock-in.

A buyer brochure for enterprises evaluating Netra: how it sees inside a node with eBPF, what happens step by step when traffic starts dropping and how the node's packets are captured as evidence, and everything else it does.

OBSERVE
Flows, TCP health, DNS timing, kernel drops, workload identity. No CNI required.

DIAGNOSE
Which connection lost packets, why, and in which kernel function.

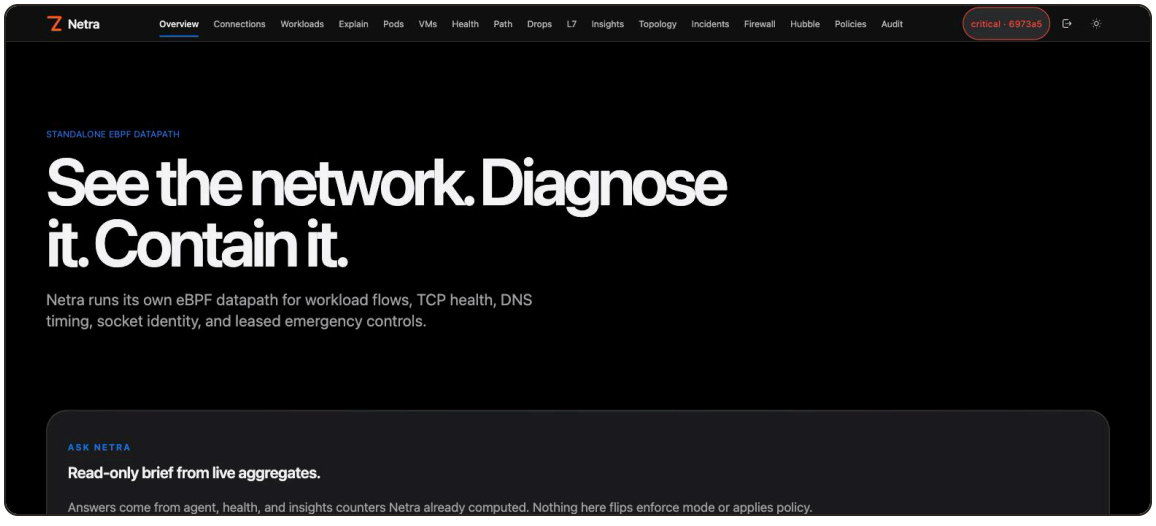
CONTAIN
Leased emergency deny, previewed against observed traffic first.

FAIL OPEN
Every enforcement rule reverts on its own.

NEW IN THIS EDITION

- Kernel drop attribution
- TCP retransmit and reset events
- Listen-queue pressure
- Sampled Redis, SQL, Kafka, gRPC counts
- TLS plaintext sampling (opt-in)
- OIDC roles
- Agent mutual TLS
- OTLP and Loki push
- Per-workload metrics and SLOs
- Batched map reads
- Tested upgrade, rollback and HA

Standalone eBPF for Linux and Kubernetes · Optional Cilium and Hubble enrichment · x86_64 and arm64



Illustrative lab UI: the Netra overview. UI captures in this brochure are taken from lab deployments; diagrams are conceptual.

When enterprises buy Netra

Modern clusters accumulate network tools faster than operators can trust them. CNI dashboards explain policy verdicts. Host stacks still drop packets in softnet, qdisc and the NIC. Emergency denies that never expire become the next outage. Netra is built for that gap.

Buyer problems Netra addresses

1. CNI-only visibility leaves node-level loss unexplained.
2. Changing or mixing CNIs should not erase observability and emergency control.
3. Incident kill-switches must auto-revert: a permanent silent deny is unacceptable.
4. Evidence (capture, alerts, SIEM, AI) must not quietly collect payloads, argv or secrets.
5. Security teams need to know exactly what a node sensor can see: what is on by default, what is opt-in, and what is never collected.

What Netra is, in one sentence

Netra is standalone eBPF network observability and leased emergency containment for Linux and Kubernetes: observe-first, CNI-independent by default, with optional Cilium and Hubble enrichment.

Two workloads

netra-agent (per node)

A privileged DaemonSet that loads the eBPF programs, owns its maps under `/sys/fs/bpf/netra`, reads them in batches and reports every few seconds. It never touches Cilium-owned maps.

netrad (controller)

The API, the dashboard, the alert poller, the durable state and the enforcement leases. Active/passive HA with a Kubernetes Lease. Serves the UI over HTTPS by default.

At a glance

28
dashboard pages: overview, flows, drops, path, capture, firewall, insights and more

8
eBPF objects, each loaded on its own so one cannot break another

0
kernel modules and application changes

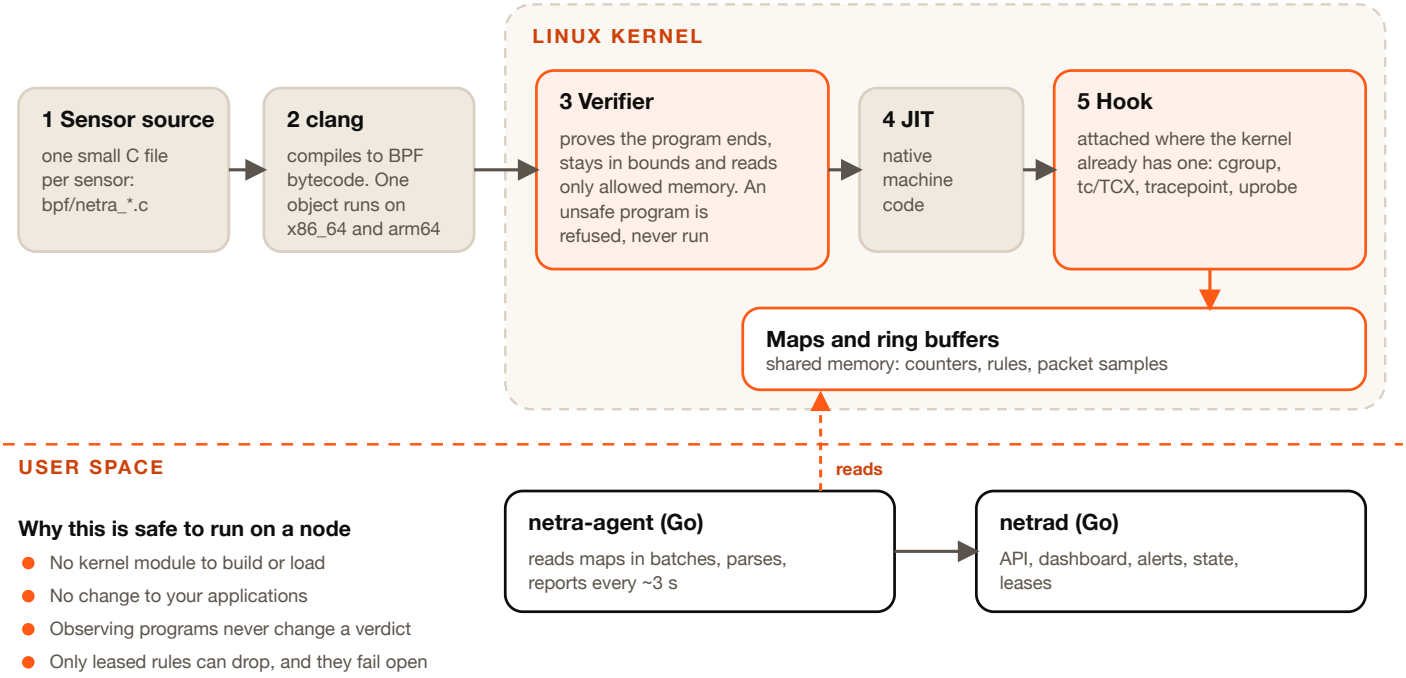
180
MCP tools for AI agents; the 60 that change state are off by default

2
CPU architectures verified on real kernels: x86_64 and arm64

30+
CI jobs on every change, plus a nightly run

How eBPF does it

eBPF lets Netra run small, verified programs inside the Linux kernel at the points where packets, sockets and drops already pass. That is how it sees a node without a kernel module, a sidecar in every pod, or a change to any application.



From source to a kernel hook and back to the agent. Illustrative.

What Netra attaches to

cgroup packet and socket hooks for every workload, TC/TCX on interfaces you choose, XDP if you opt in, tracepoints for drops and TCP events, and uprobes in OpenSSL if you opt in.

How the data comes out

Programs write counters and samples into maps and ring buffers. The agent reads big maps in batches and keeps only the top entries, so a busy node costs a few dozen syscalls per read.

What keeps it safe

The kernel verifier refuses anything unsafe. Each sensor is its own object. Observing programs never change a packet's verdict. Rules that can drop are leased and fail open.

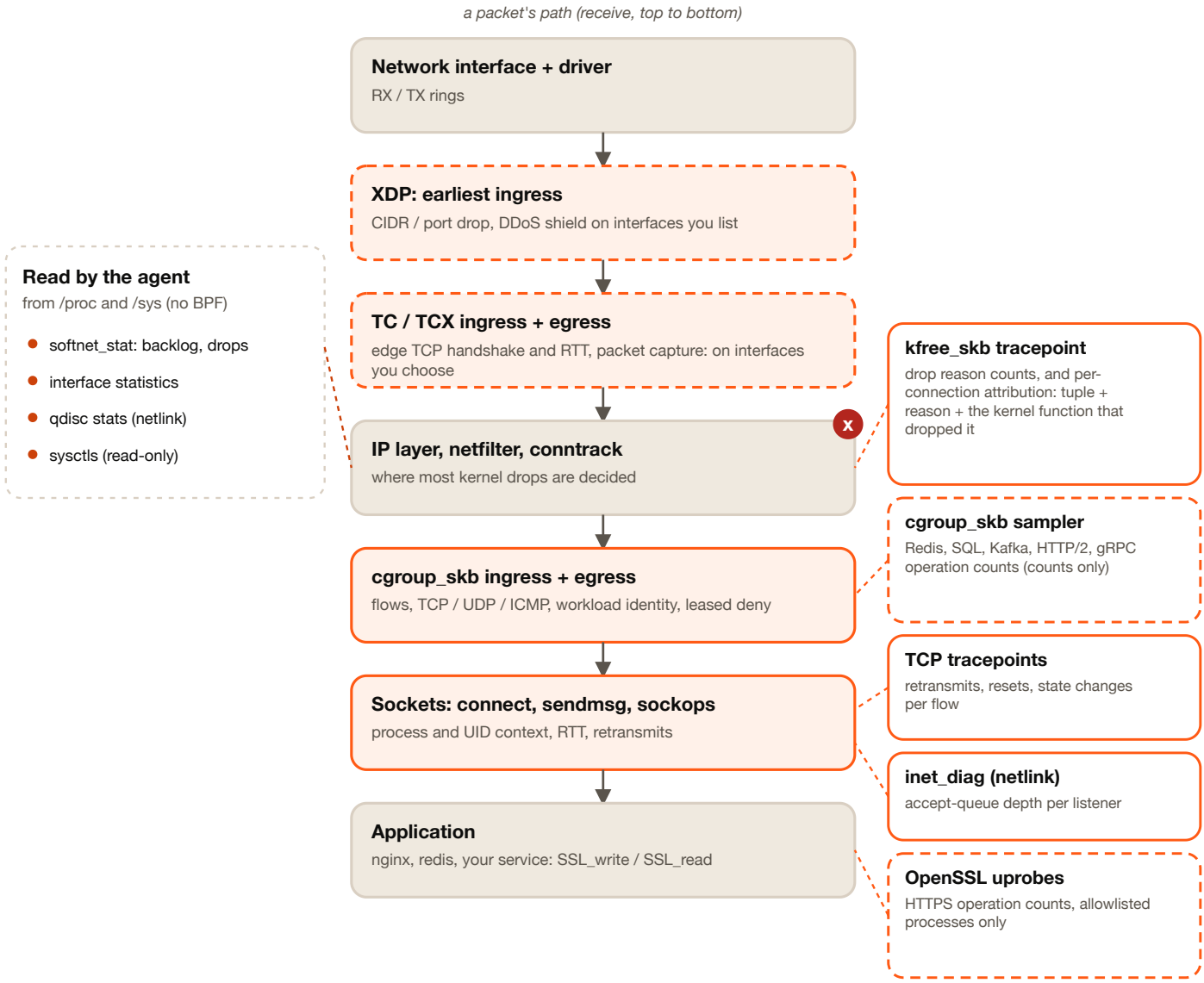
Proven against the real verifier. Netra's objects are loaded by real kernels in CI, on Linux 6.8 and 6.17, on x86_64 and arm64. When the first sampled-protocol program was rejected (the compiler had removed a bounds check), it was fixed and is now loaded by a test on every run.

Why eBPF, not the alternatives

	Typical kernel module	Typical sidecar or proxy	Netra with eBPF
Sees node-level drops	yes	no	yes: the drop tracepoint, softnet, qdisc
Changes to applications	none	one per pod	none
Risk to the kernel	a bug can crash the node	lives in user space	the verifier proves each program safe before it runs

Where Netra sits in the Linux network stack

A packet crosses the same layers on every node. Netra places a sensor at each layer that matters and reads the rest from what the kernel already exposes, so the picture of an incident is complete from the wire to the application.



Why cgroup hooks

They see every workload's traffic with its identity, on any CNI, without an interface to attach to.

Why a drop tracepoint

Most packet loss is decided inside the kernel. The tracepoint says which connection, why, and in which function.

Why TCX for capture

It observes packets on the interface you pick without replacing anything, and never changes a verdict.

The eight eBPF objects

Each sensor is a separate object with its own verifier budget and its own switch. A node that cannot run one loses only that sensor, and says why.

Object	Hook	What it reads	Default	Needs
netra_tc	cgroup_skb in/egress, connect, sendmsg, sockops, drop tracepoint; TCX and XDP optional	flows, policy drops, connection tracking, kernel drop reasons	on, observe mode	cgroup v2
edge_intel	TCX ingress and egress	handshake and RTT histograms, retransmit / RST / FIN counters, before NAT	auto	Linux 6.6+ (TCX)
capture	TCX ingress and egress	filtered packet copies into a ring buffer	auto, idle until a session starts	Linux 6.6+ (TCX)
tlsfp	cgroup_skb egress	first bytes of a TLS ClientHello, for JA3 / JA4 in user space	auto	cgroup v2
tcpevents	four TCP tracepoints	retransmit, reset and state change, with the flow tuple	auto	tracefs (no BTF)
dropinfo	drop tracepoint	tuple, reason and the kernel function that dropped the packet	auto	kernel BTF, tracefs
l7sample	cgroup_skb	first 128 bytes on the ports you configure, parsed in the agent	**off**	cgroup v2
ssl	uprobes on SSL_write / SSL_read	up to 128 bytes of plaintext, allowlisted processes only	**off**	host PID view, OpenSSL

Kernel baselines

Ring buffers need Linux 5.8+. Batched map reads need 5.6+. TCX needs 6.6+. Only drop attribution uses kernel type information (BTF); every other sensor works without it.

Never in the way of other tools

Netra owns only `/sys/fs/bpf/netra`. It does not pin over Cilium's maps, and its capture and edge programs return the kernel's default verdict, so they cannot change a packet's fate.

Layouts and names come from the running kernel

Tracepoint record layouts and drop-reason names are read from the kernel's own description at load time. Reason numbers change between kernels: `NETFILTER_DROP` is 8 on Linux 6.8 and 12 on 6.17. A table baked into the agent would mislabel every drop on one of them, so Netra never bakes one in. A layout it cannot read safely is counted and reported, never misread.

What a node reports when a sensor cannot run

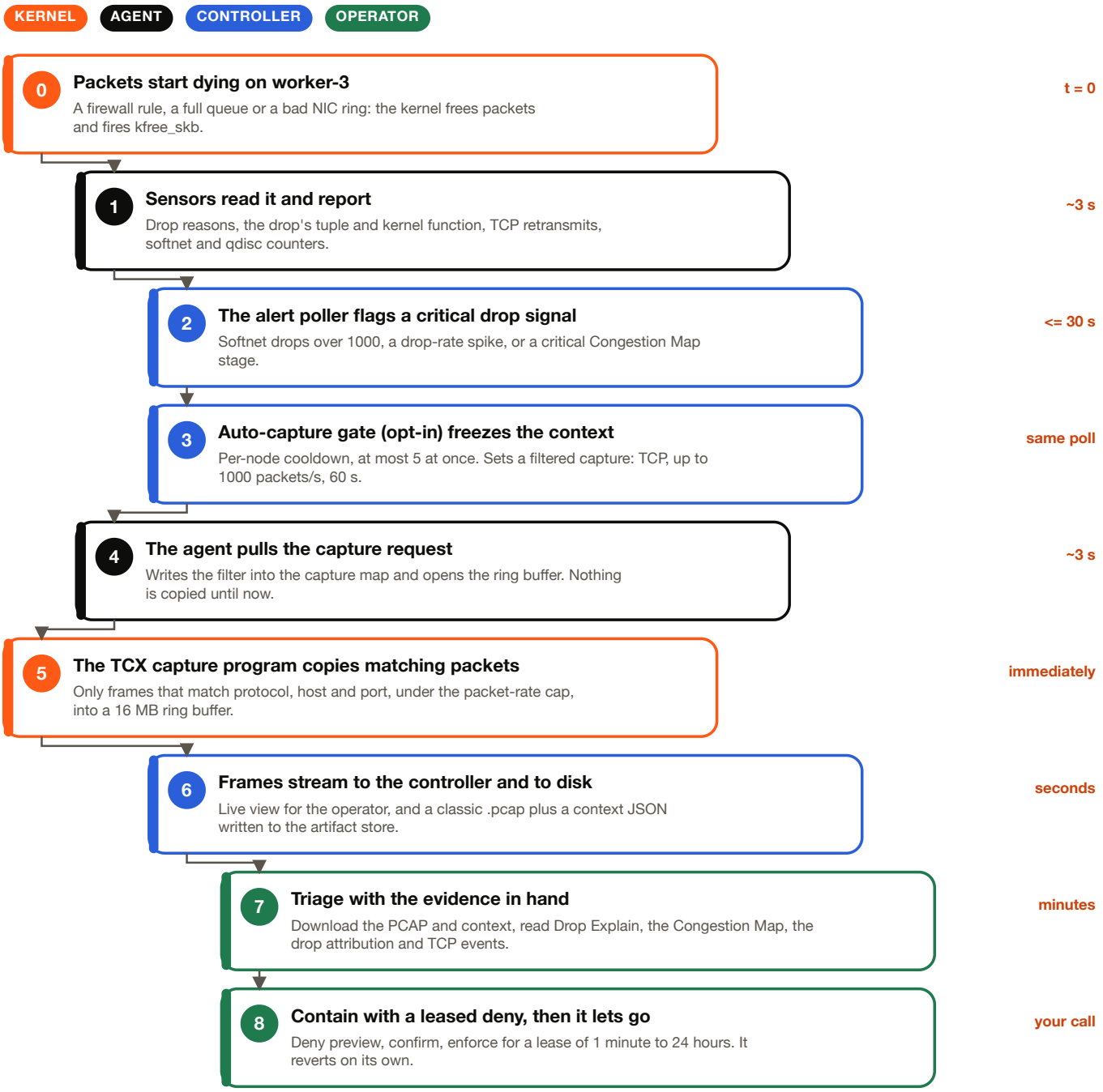
Node	Sensor	Reported
worker-1	drop attribution	running
worker-2	drop attribution	kernel BTF not available

Illustrative. Three modes on every sensor: `auto` runs it if the node can and reports why if not, `off` skips it, `required` stops the agent instead of running without it.

TRAFFIC-DROP SCENARIO · PART 1 OF 4

Packets start dying on a node

It is 02:10. Pods on **worker-3** report timeouts and retries. The CNI dashboard shows healthy policies. Nothing explains it. Here is what Netra does, in order, and what each step gives the operator.

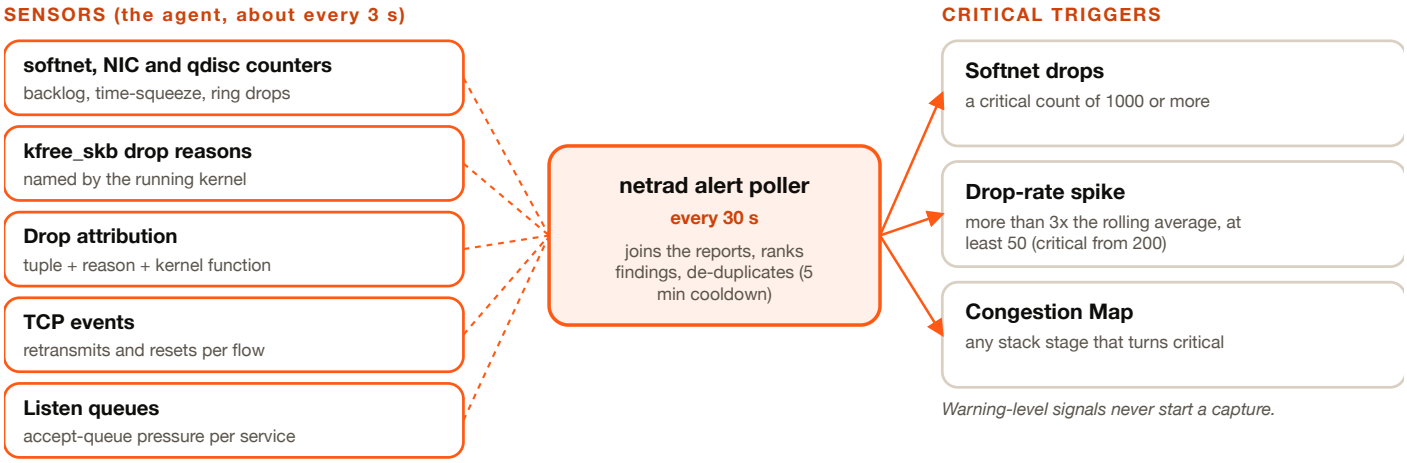


Timings are typical, not guarantees. Steps 3 to 6 are the opt-in auto-capture path (off by default); an operator can start the same capture by hand at any time. Diagram is illustrative.

TRAFFIC-DROP SCENARIO · PART 2 OF 4

Detect the drop, and explain it

The sensors that explain a drop are always on where the node supports them, so the evidence exists before anyone asks for it. The controller turns critical signals into an alert, and the dashboard shows where in the stack packets die.



Warning-level signals never start a capture.

DROP ATTRIBUTION — *illustrative*

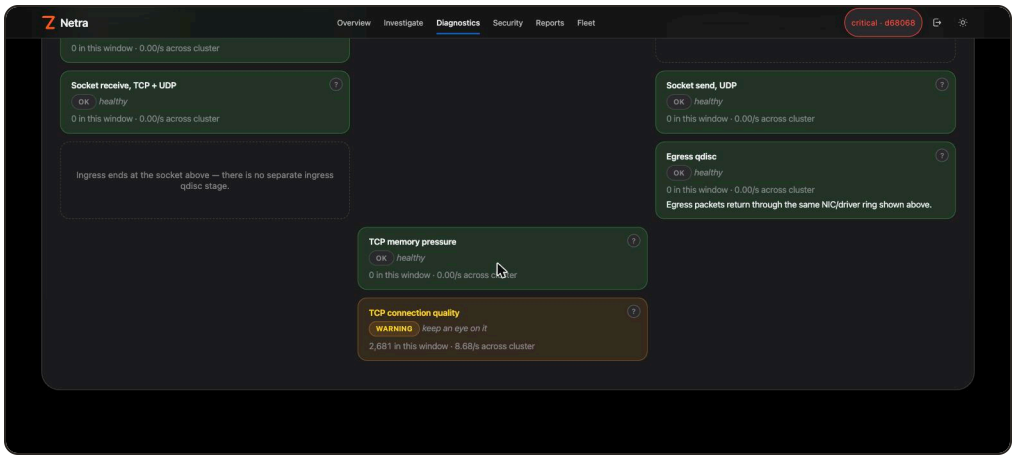
NETFILTER_DROP	nft_do_chain [nf_tables]	tcp 10.0.0.5:41722 -> 10.0.0.9:443	1,204
NO_SOCKET	__udp4_lib_rcv	udp 10.0.0.7:5353 -> 10.0.0.9:9	86

What you can read

- **Drops page:** kernel skb reasons, softnet pressure, interface counters and policy-drop findings.
- **Drop attribution:** which connection, which reason, which kernel function (`nft_do_chain` versus `__udp4_lib_rcv` is the difference between a firewall rule and a closed port).
- **TCP events:** the flows behind retransmits and resets.
- **Listen queues:** which service's accept queue is full.

Explain, without guessing

- **Drop Explain** gives an evidence-backed reading of one flow, passively, from data already collected.
- **Congestion Map** paints every stack stage (NIC, softnet, qdisc, socket) by severity across the cluster.
- Tuning advice is review-only with apply and rollback commands. Netra never writes a sysctl.

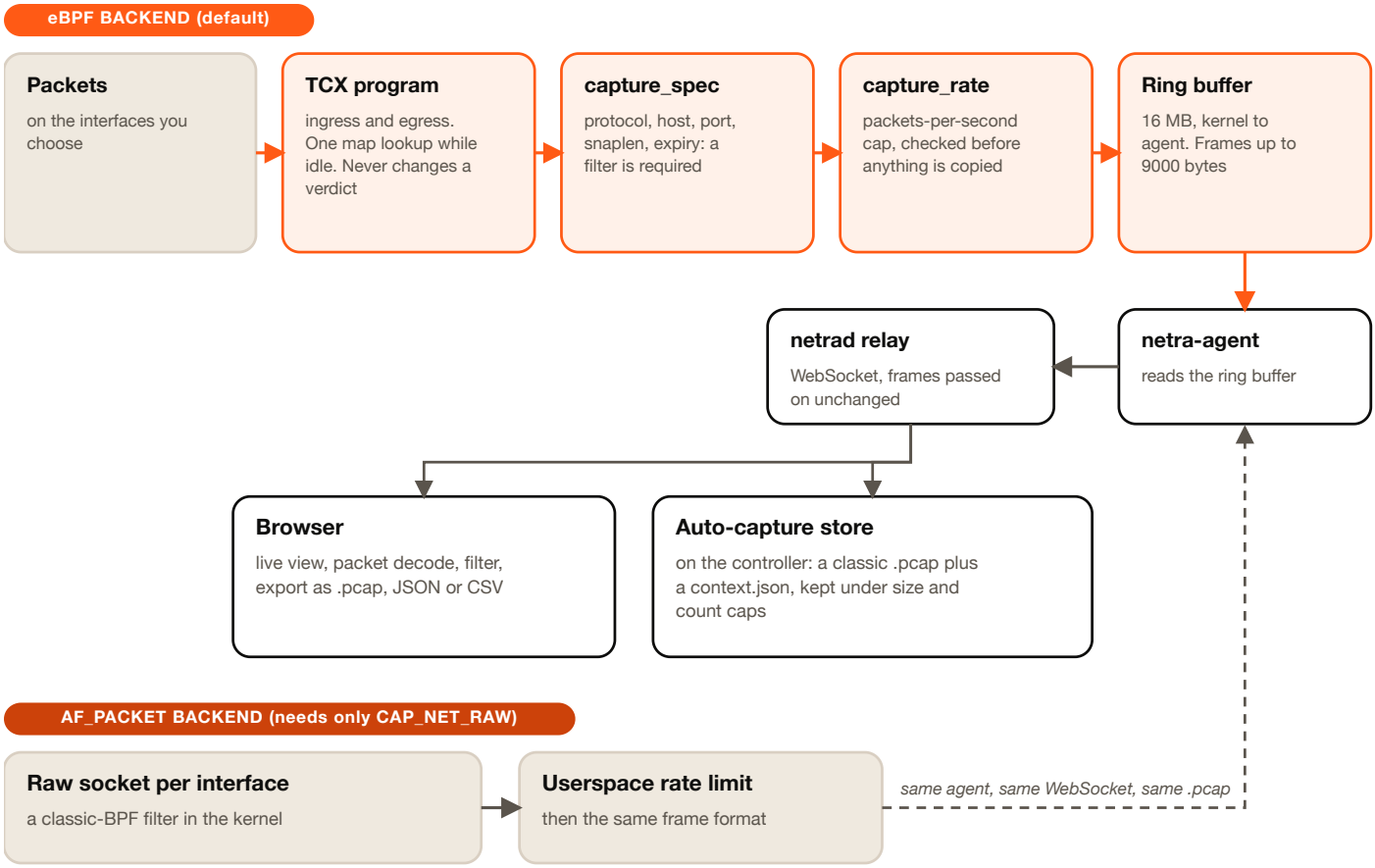


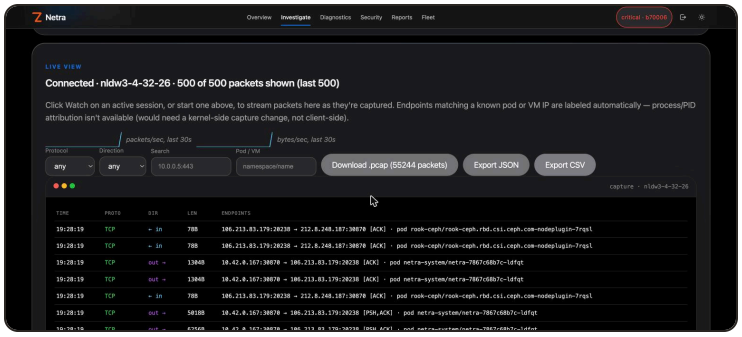
Illustrative lab UI: the Congestion Map, colored by severity

TRAFFIC-DROP SCENARIO · PART 3 OF 4

Capture the node's packets

When the signal is critical, Netra freezes the context and starts a filtered, time-boxed capture on the affected node. Two backends produce the same frames, so the operator's view and the saved PCAP do not depend on the kernel.





Illustrative lab UI: live capture, decoded and labeled by workload

Guard rails

- Auto-capture is **opt-in**, off by default.
- A filter is always required. No whole-interface capture.
- Auto: 60 s, up to 1000 packets/s. Manual: up to 5 minutes.
- Per-node cooldown 10 minutes; at most 5 auto captures at once.
- Store capped at 50 files and 1 GB. Downloading needs the operator role.

The evidence bundle

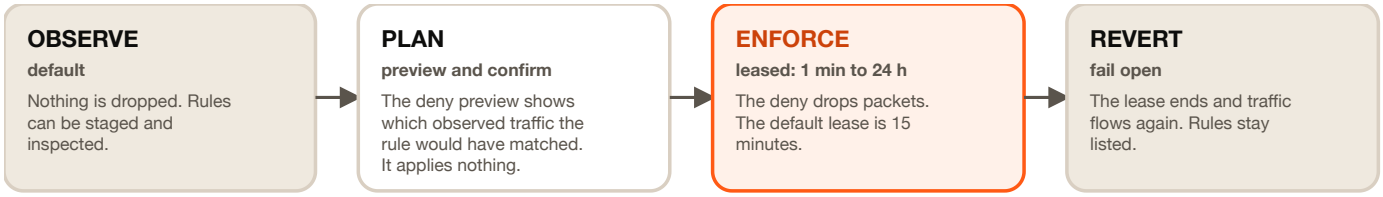
A classic **.pcap** that opens in Wireshark, and a **context.json** frozen at the moment of the trigger: node, host and kernel, whether CPU or memory was hot, the top workloads, a short comm-only process list, and the drop, softnet and qdisc counters.

What a capture contains. Full frames of the traffic that matches your filter, up to 9000 bytes each. It is a deliberate, filtered, time-boxed exception. Never collected: command lines, environments, executable paths, Secrets or logs.

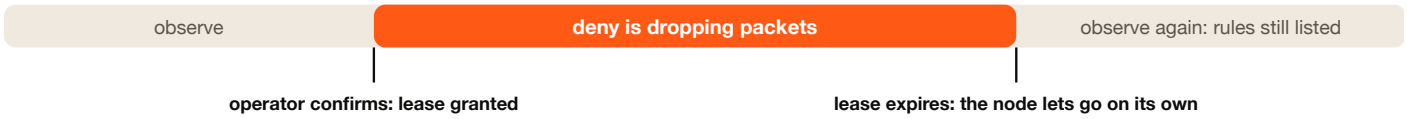
TRAFFIC-DROP SCENARIO · PART 4 OF 4

Triage, then contain, then let go

With the PCAP, the context and the drop attribution in hand, the operator knows what is hitting the node. If it must be stopped now, Netra offers a containment that expires on its own.



TIME



IT ALSO LETS GO WHEN

The lease expires
checked on the node itself, even if the controller is gone

The controller is lost
no refresh for NETRA_FAILSAFE_AFTER (60 s by default)

The controller restarts
an old lease is never resurrected from disk

HA leadership changes
the new leader starts in observe

Before you deny

The **deny preview** scores a proposed rule (an IP, CIDR, port, DNS name, SNI or process) against the counters Netra has already observed and applies nothing: it shows which flows the rule would have matched. The **blast-radius** view walks the observed dependency graph outward from a workload (1 to 6 hops, 3 by default) and shows what depends on it. Both read observed traffic only, and say so: they are a guide, not a policy verdict.

What you can lease

Exact IPv4 and IPv6, CIDR (ingress or egress, optional SYN-drop so established connections survive), port, DNS name, TLS SNI, UID, process name, capability-gated socket deny, packet and byte ceilings, and per-workload new-connection caps. Allow-exceptions always win.

Proven on a real kernel

In CI, on every change, a real agent on a veth pair shows that:

- a deny rule never blocks in observe mode;
- in enforce it drops exactly what it names, and a second port on the same host keeps working;
- egress rules guard connections the host starts, ingress rules guard connections the peer starts, and replies to the host's own connections are exempt;
- an allow rule beats a deny rule, and an IPv6 rule drops IPv6 and leaves IPv4 alone;
- a lease expires on its own with the rule still listed;
- the agent fails open when the controller dies.

What Netra collects, and what it never does

The old promise was "no payloads". The accurate one is stronger and more useful: nothing that could carry your data is collected by default, every exception is opt-in, limited and stated here.

By default

Metadata only: flow tuples and counters, workload and process identity (comm, pid, uid, cgroup), DNS query names on UDP/53, TLS SNI and JA3/JA4 fingerprints, the HTTP/1 method and Host of a single cleartext packet. No payload bytes are copied to user space.

Opt-in, and limited

Packet capture: full frames of a filtered, time-boxed session. **Sampled L7:** the first 128 bytes on ports you configure, parsed in the agent. **TLS plaintext:** up to 128 bytes per call, from processes you name. Each is off until you turn it on.

Never

Command lines, environments, executable paths, Secret contents, logs or dmesg. Request paths, query strings, headers, bodies, SQL text and cache keys from the sampling sensors. API keys or sink credentials in a log, an API response or a metric.

Question	Answer
Does Netra decrypt TLS?	Not by default. TLS plaintext sampling is a separate opt-in that reads OpenSSL's buffers in the agent, for processes you list. The list is enforced **in the kernel before any byte is copied**. The plaintext is parsed in memory and never exported; only counts (method, status, a validated host name) leave the agent.
What can a sampled count contain?	An allowlisted operation name (a Redis command, a SQL verb, a Kafka API, an HTTP method and status, a gRPC method), a coarse outcome, or a validated pattern. This is enforced by the shape of the output and tested by planting secrets in real requests and asserting they appear nowhere the agent or controller exposes.
What does the host-PID view widen?	Finding each process's OpenSSL library needs a view of the host's processes. The chart asks for it only when TLS sampling or process metadata is enabled.
Who can download a capture?	The operator role or above. With OIDC on, the audit trail names the person.
Can a busy node be made to spend unbounded resources?	No. Sampling is rate-limited per flow and the kernel counts what it skipped; capture has a packet-rate cap, a time box and a store cap; per-workload metrics have a hard cardinality cap.
Are credentials for exports safe?	Headers, tokens, secrets and passwords are never logged and never appear in an API response. CI plants them and scans every request body, log line and response.

Coverage limits, stated plainly. TLS sampling sees OpenSSL only: not Go's crypto/tls, statically linked BoringSSL, or applications with their own TLS. HTTP metadata by default is one cleartext HTTP/1 packet. DNS parsing is ordinary UDP/53; DoH, DoT and TCP DNS are not inspected. The agent is privileged and should run only on trusted nodes.

The newest kernel sensors

Five sensors answer the questions that used to need a lab: which connection lost packets, which flows retransmit, which service is refusing connections, and what the application protocols are doing.

Drop attribution — auto

The dropped packet's connection, the reason (named by the running kernel) and the kernel function that dropped it. No header yet? It is counted with no tuple, never a made-up one.

TCP events — auto

Retransmits, resets and state changes per flow, with the tuple. Names the flows behind the older aggregate counters. No BTF needed.

Listen queues — auto

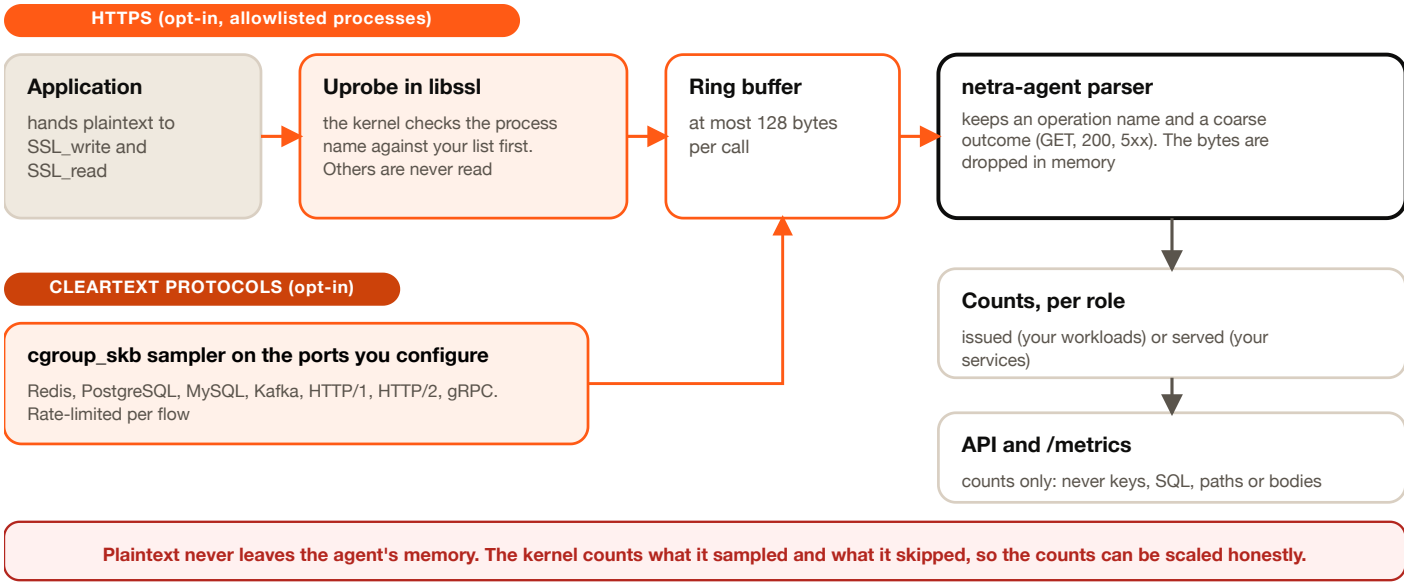
Accept-queue depth and overflow per listening socket through the kernel's socket-diagnostics interface: no BPF program, no payload.

Sampled L7 — opt-in

Redis commands, PostgreSQL and MySQL verbs, Kafka APIs, HTTP/1 and HTTP/2 status, gRPC methods and errors. Counts carry a role (issued or served) so a request seen twice is never counted twice.

TLS plaintext sampling — opt-in

HTTPS method, status and host from OpenSSL, on the processes you allowlist. The same allowlisted counts as cleartext protocols, read where the application hands bytes to the library.



Plaintext never leaves the agent's memory. The kernel counts what it sampled and what it skipped, so the counts can be scaled honestly.

What each sensor costs is measured and shown on the next page, with where and with what caveats.

Diagnostics an SRE already recognises, made cheaper and deeper

The dashboard keeps the views operators rely on and adds the kernel-level evidence behind them.

Path diagnostics

Active TCP connect latency and transport pressure (cwnd, packets out, retransmits, lost, delivered rate) per cgroup and workload, from a socket-level program.

Edge TCP intel

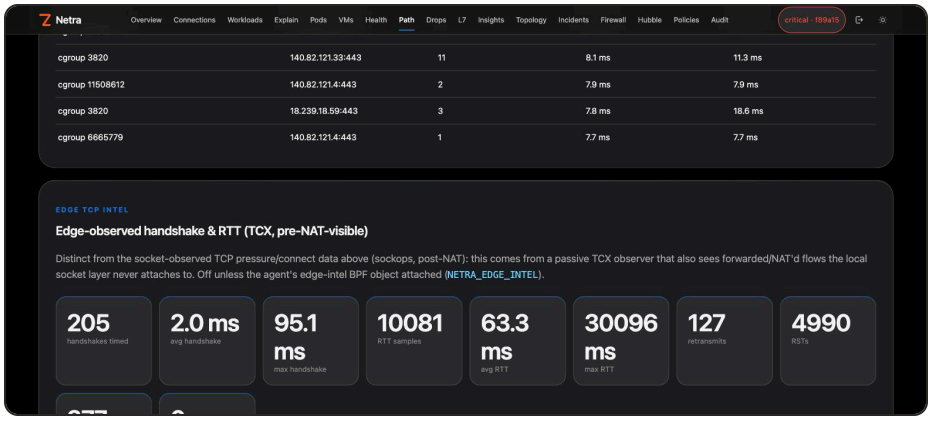
A passive TCX observer adds handshake and RTT histograms and retransmit / RST / FIN counters for forwarded flows that the local socket layer never attaches to.

Observability

L3/L4 flows with direction and hook, DNS timing, TLS SNI and JA3/JA4, cleartext HTTP method and Host, workload identity, a live topology, and a queryable flow history with RED metrics.

Kernel and host

Softnet, interface and qdisc pressure, a read-only sysctl audit against a baseline, per-node CPU, memory and load, and listen-queue depth.



Illustrative lab UI: Path Diagnostics with edge TCP intel

Cheap by design: what each sensor costs (measured)

Sensor	Measured cost	Where, and the caveat
Drop attribution	about 100 ns per drop; about 540 ns on a shared runner	4-vCPU arm64 VM on Linux 6.8, and a GitHub runner on 6.17. Loopback, not a NIC under load. It runs only for dropped packets; on a host dropping millions per second use `off`.
Sampled L7	about 29 ns on an unconfigured port, 65 ns on a configured one	Per packet, Linux 6.8 arm64 VM, loopback.
TLS plaintext	about 1.0 µs (x86_64) and 1.4 µs (arm64) per probe; about 3 to 4% on an HTTPS request that includes a handshake	Shared CI runners, loopback, one client library.
Agent map reads	194% of a core down to about 23%; memory 313 MiB down to about 114 MiB	A busy shared lab host with about 130,000 flows per table. Batched reads, and identical report output (checked against the old code).

Measured on lab hosts and CI runners; results depend on your kernel, hardware and traffic.

Emergency firewall and behavior insights

Emergency firewall: leased enforce

Netra's containment layer is deliberately an emergency tool, not a full CNI policy engine, QoS system, L7 proxy or IDS/IPS. It starts in observe mode, enforcement needs a lease, and everything reverts on its own.

- Exact IPv4 / IPv6 and CIDR deny, ingress or egress, with optional SYN-drop mode.
- Port, DNS name, TLS SNI, UID and process name, with allow-exceptions.
- Packet and byte ceilings, and per-workload new-connection caps.
- Optional XDP early ingress on interfaces you list; optional native NetPol allow-list and default-deny per workload.
- Workload scoping: enforce only the workloads you select; unresolved traffic fails open.
- Blast-radius and deny preview apply nothing.
- Every change is audited, with the person named when OIDC is on.
- Rules survive restarts and HA failover; enforcement never does.

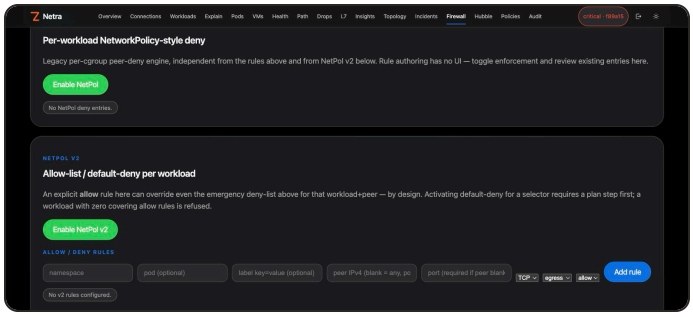
Behavior and rate insights

Controller-side insights turn eBPF metadata into a workload dependency graph, a known-good behavior baseline, drift detection for newly observed destinations, DNS names, SNI and HTTP hosts, time-window rate baselines with deterministic multipliers, exposure scoring, and review-only remediation proposals.

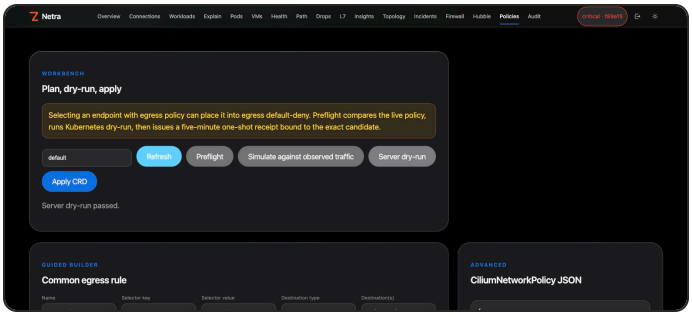
> **Netra never auto-applies learned policy.** A baseline is an explicit operator decision, and clearing it needs a dedicated confirmation. Recommendations come from observed traffic, which may be incomplete, and are labelled as such.

Policy authoring and GitOps

When Cilium is present, Netra plans and applies `CiliumNetworkPolicy` with a fresh single-use preflight receipt, risk confirmation for high-impact plans, rollback history, and optional policy-as-code reconciliation from a mounted directory.



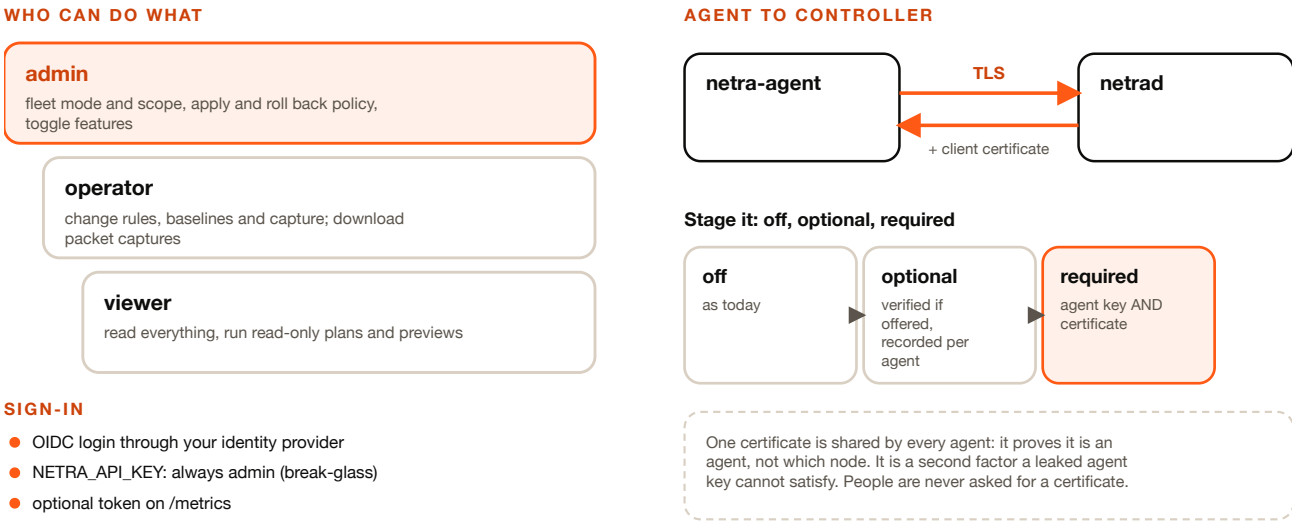
Illustrative lab UI: the Firewall page



Illustrative lab UI: plan, dry-run and apply with a one-shot receipt

Access, security and export

All of it is off by default and additive: nothing about an existing install changes until you turn a piece on, and the static keys keep working.



Sign-in and roles

People sign in through your identity provider (OIDC) and get viewer, operator or admin; the audit trail names the person. A new mutating route is gated at operator by default, and an identity-provider outage is a temporary error, not a logout.

Agent mutual TLS

A client certificate as a second factor for agent-only requests. Roll out with **optional**, watch which agents have moved, then switch to **required**. Certificates renew in place, and any bad setting fails closed.

Metrics and SLOs

Aggregate `/metrics` by default, optionally token-gated. Opt in to per-workload counters with a hard cardinality cap; a dying pod never makes a counter go backwards. Network SLOs with burn-rate alerting, a chart-rendered **ServiceMonitor** and **PrometheusRule** (checked with promtool in CI), and a Grafana dashboard.

Pushing telemetry out

Sink	Sends	Notes
OTLP / HTTP	metrics, audit logs, block and deny events as spans	OTLP/HTTP JSON to a collector's base URL
Loki	audit and block events under a small bounded label set	Multi-tenant, basic auth or headers
Syslog, Snowflake	audit events: RFC 5424 over UDP or TCP; a Snowflake table	Key-pair authentication for Snowflake
Alert channels	severity-filtered alerts	Webhook (HMAC-signed), e-mail, Slack, Teams, Twilio SMS and WhatsApp, HTTP bridge
SIEM pull export	events, audit, blocks and flows	JSON, JSONL, CEF, RFC 5424 syslog, OTLP

Push sinks are best-effort and leader-only in HA, so two replicas never double-ship. Sink credentials are never logged and never appear in an API response; CI plants them and scans every body, log line and response.

AI, MCP and ChatOps: read-only until you say otherwise

Ask Netra and digests

Briefs, digests, suggestions, explain and draft turn data the controller already computes into operator text. They never flip enforce mode or apply policy. The engine is heuristic by default with no external calls; an OpenAI-compatible rewrite is optional and receives bounded aggregates only (counts, a health score, top-N destinations, DNS names and processes), never packets or secrets.

netra-mcp for AI agents

180 stdio tools: **120** read and generator tools are always available; the **60** that change state exist only when you set `NETRA_MCP_ALLOW_MUTATIONS`. Mutations reuse the same bearer auth, self-reverting leases and audit log (policy applies also need a single-use preflight receipt), with the MCP actor named. CI runs the real server against a real controller.

Slack and Teams ChatOps

Slash commands and an interactive confirmation flow that mirrors the UI's confirmations, with a separate credential from the operator key.

netractl and netra-doctor

An operator CLI for the main API surfaces (install, status, features, maps, capture, export) and a read-only host-readiness preflight: cgroup v2, bpffs, BTF, tracefs, TCX and drop-reason support.

Alerting

A poller evaluates health, path, drop, kernel-network and drift findings on an interval (30 s by default) and publishes new or escalated ones through the configured channels, with severity-aware cooldown and de-duplication. Alerting is off until you configure a channel. Delivery is best-effort with bounded retries; a failed channel does not block the others.

Example: an AI agent works the drop scenario

`netra_ebpf_drops` → `netra_ebpf_diagnose` → `netra_drops_explain` → `netra_capture_status` → `netra_insights_blast_radius` → `netra_ebpf_deny_add`

The first five are read tools. The last exists only when you enable mutations, and a rule it adds drops packets only while an enforcement lease is running. Every step is audited with the MCP actor named.

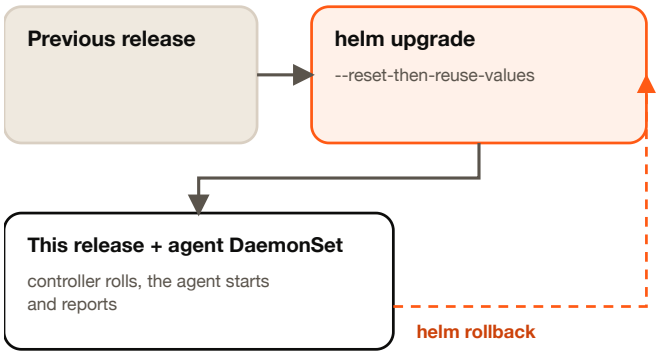
Where the AI and automation stop

- Nothing suggested by the heuristic engine, the LLM rewrite or a recommendation is ever applied on its own.
- Policy recommendations are generated from observed traffic, which may be incomplete; they are labelled as such.
- Mutating MCP tools, ChatOps actions and the UI go through the same authentication, lease and audit path; policy applies also need a fresh single-use preflight receipt.
- Prompts are advisory text; the real gate is the mutation switch.

Deploy, upgrade, and how it is tested

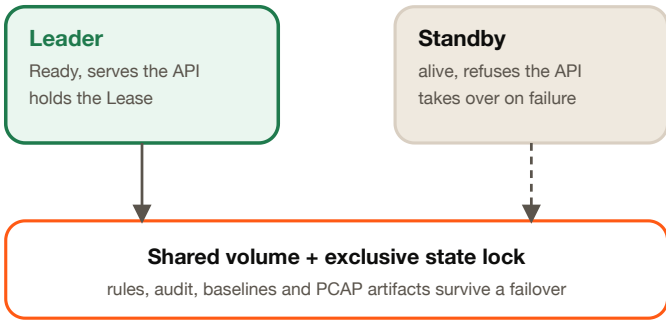
One Helm chart installs the controller and the agent DaemonSet. Plain manifests are available. `netra-doctor` checks a node before the privileged agent runs on it.

UPGRADE AND ROLLBACK



- rules, baseline and API key survive
- a pod restart keeps the state
- mode returns to observe after a restart

HIGH AVAILABILITY



On a crash or a graceful delete the standby is promoted. Enforcement restarts in observe: an old lease is never carried across.

Tested for real, not just unit-tested

EVERY PUSH

unit tests and the race detector, gofmt, shellcheck, lint, a vulnerability scan, Helm rendering

REAL CONTROLLER

MCP, persistence, alert and export sinks, OIDC, mutual TLS, every netractl command

REAL KERNEL

enforcement, packet capture and the sensors on a veth pair: x86_64 and arm64, Linux 6.8 and 6.17

REAL CLUSTER

install, upgrade from the previous release, rollback, the plain manifests, HA failover

REAL BROWSER

sign-in, all 28 dashboard pages against a live controller, firewall actions

EVERY NIGHT

the whole suite under the race detector, fuzzing, a kernel matrix, multi-arch images

Each check is proven by breaking what it tests: the job must then fail.

Enforcement and capture are checked with a real agent on a real kernel and real traffic. **Upgrades** install the previous release, write state, upgrade to the new one, restart a pod and roll back, and assert the rules, baseline and API key survive. **HA** checks that exactly one replica serves, the standby refuses the API, and both a crash and a graceful delete promote it with the shared state intact.

Honest limits. A real Cilium and Hubble install is not in CI (the chart's Cilium mode is checked by rendering only). HA is tested on one node, so it proves the election and the state lock, not your shared filesystem.

Zyvor suite placement: Netra and PacketWolf

Netra and PacketWolf are suite counterparts, not a wired pipeline. Neither product is a dependency or API consumer of the other today.

	Netra	PacketWolf
Default market	Mixed or non-Cilium clusters; an easy entry point	Cilium-standardized fleets
Core job	Observe, drops, congestion and leased emergency deny	Observe, AutoPolicy, healer and platform containment
CNI dependency	None (Cilium and Hubble optional)	Cilium-first
Policy posture	Review-only drafts; enforcement is time-leased and fails open	Learns and operates Cilium policy at platform depth
Maps	Owns `/sys/fs/bpf/netra` only	Works with the Cilium datapath and policy

Co-existence rules

1. The same cluster is allowed but not required; Netra's maps stay isolated.
2. Do not dual-own a long-lived deny: PacketWolf and Cilium for durable posture, Netra leases for an incident kill-switch that auto-reverts.
3. Enable Hubble and CiliumNetworkPolicy in Netra only for enrichment, not as PacketWolf's datapath.
4. There is no Netra to PacketWolf control-plane sync today.

Pick Netra when you need CNI-independent diagnostics and short-lease emergency deny. Pick PacketWolf when Cilium is already the CNI and you want full platform intelligence, AutoPolicy and healer depth.

Buying checklist

1. Run `netra-doctor` on candidate nodes: cgroup v2, bpffs, BTF, tracefs and TCX support.
2. Start observe-only: flows, Path, Drops, Congestion Map. No enforcement lease yet.
3. Turn on the optional sensors you want and read what each node reports it can and cannot run.
4. Exercise capture (and, if you want it, auto-capture) on a non-production signal, and decide who may download PCAPs.
5. Pilot a leased deny after its preview, and confirm it reverts on its own.
6. Wire alerting, SIEM, OTLP or Loki as needed; keep AI and MCP read-only until trusted; decide on OIDC and agent mutual TLS.
7. Rehearse an upgrade, a rollback and an HA failover in staging.
8. If Cilium is present, decide explicitly between Netra enrichment and PacketWolf platform ownership.

See **Explain** **Capture** **Contain** **Integrate** **Trust**

flows on any CNI · the connection, reason and kernel function · a filtered PCAP · a leased deny · OIDC, mTLS, OTLP, Loki, SIEM, MCP · tested for real

See the network. Contain the incident. Fail open by default.

Talk to sales → zyvor.dev/netra