PURETENSOR
INTELLIGENCE DISTILLED

WireGuard Inside VXLAN

How a mesh VPN on Kubernetes nodes loses 8x throughput while every latency check stays green

Paper: PT-R-2026-027 v1.0

Author: PureTensor Ltd

Date: 27 September 2026

Status: Published

Abstract

Every node in our Kubernetes cluster runs Tailscale, a mesh VPN built on WireGuard. The pod network is flannel in VXLAN mode. VXLAN tunnels layer-2 frames inside UDP; flannel is the component that builds the pod network between nodes. Flannel places pod-network addresses on the host itself: each node's VXLAN device carries a /32 from the pod range, and its pod bridge carries a /24 gateway address. Tailscale offers every local address to its peers as a possible WireGuard endpoint. A peer that tries a pod address reaches it, because every node has a route to every other node's pod slice. The resulting path is WireGuard inside VXLAN, a double encapsulation, and Tailscale selects it repeatedly.

On 26 September 2026 this cost about 8x in throughput on the 25G tier. Using iperf3 (a standard TCP throughput tester) from a two-GPU compute node, we measured 529 to 580 Mbit/s to a storage node on the pod path. The same pair after the fix gave 4.44 to 4.45 Gbit/s sustained, and a clean neighbouring storage node gave 4.65 Gbit/s. The fault had been seen once before, on 21 July 2026, between the two GPU nodes on the 200G fabric: 27 to 55 Mbit/s, against 6.9 Gbit/s after recovery. Throughout, every latency-shaped signal stayed green. Tailscale's own ping answered "in 0s", the LAN round trip was 0.395 ms, every node was Ready, Ceph reported HEALTH_OK and no pod was outside Running.

The fault was misread twice before it was understood. In July it was treated as a one-off that a daemon restart cured, and the detector written then could not see it. In September it was first read as a "latch" onto a bad path, and a restart bought about 20 minutes. The Tailscale daemon's journal showed something else: continuous re-selection, with 4 to 15 switches onto pod addresses per node per 10 minutes. Tailscale 1.102.4 offers no configuration lever to exclude these addresses, and the VXLAN /32 cannot be removed without breaking pod routing. We therefore installed one host firewall rule that drops inbound Tailscale discovery traffic whose source is the pod network. Pod-path switches then fell to zero on the four nodes we counted.

A read-only follow-up on 27 September found the fix holding for about 14.8 hours on the four nodes that had not rebooted. The two nodes that had rebooted since the fix were flapping again, 110 and 44 times respectively. At boot, our rule had been installed first and the Tailscale daemon had then inserted its own accept rule above it. The control is only as good as its position in the rule table. A check that the unit is enabled proves nothing. Later the same day we tested that diagnosis directly. We re-inserted the rule above the daemon's chain on the two rebooted nodes, and pod-path switches stopped: 0 on both over the next 37 minutes, against 7 and 29 in the preceding 2 hours. We also found why the rule's drop counter had read zero even

where it worked: the filter table is rewritten about once a minute, which resets every counter in it, and between rewrites the rule was dropping 2 to 4 pod-sourced discovery packets.

1. Setting

The fleet runs a Kubernetes cluster on k3s, mostly on-premises with a few remote members. The pod network is flannel in VXLAN mode on the k3s default pod CIDR, 10.42.0.0/16, and each node owns a /24 slice of the form 10.42.N.0/24.

On-premises, the tiers differ by physical link. The storage nodes are on 25G, two GPU nodes share a 200G fabric, and the monitoring tier is on 1G. The two-GPU compute node was the iperf3 client for every throughput measurement below.

Every node also runs Tailscale. At the time of the measurements the version was 1.102.4, which we confirmed live on 27 September. Two pieces of Tailscale's internals appear throughout this paper. Magicsock is its path-selection layer: for each peer it collects candidate endpoints and picks one to send WireGuard traffic to. Disco is its discovery protocol, the probe-and-answer exchange that magicsock uses to test candidates. Disco rides on the WireGuard UDP port, which is Tailscale's default port unless configured otherwise.

2. How the Host Comes to Own Pod Addresses

The fault starts with where flannel puts the pod network. Flannel does not keep it off to one side. On each node, the VXLAN device carries the address 10.42.N.0/32 and the pod bridge carries 10.42.N.1/24, and both sit alongside the node's ordinary LAN address. From the host's point of view these are local interface addresses like any other.

Tailscale enumerates every local address as a WireGuard endpoint candidate. Its own log on one storage node shows three local endpoints advertised on the WireGuard port: the VXLAN /32, the pod-bridge address and the node-LAN address. Peers receive all three and are free to try any of them.

A pod address, when tried, answers, because the pod route exists on every node. The route to a remote node's 10.42.N.0 goes via that address on the local VXLAN device, with the local 10.42.M.0 as source. A disco probe sent to a remote node's pod address therefore leaves through VXLAN, arrives, and is answered. Magicsock sees a working, low-latency path and may select it. Once selected, every WireGuard packet between the two nodes is wrapped in VXLAN as well, so the traffic is encapsulated twice. The VXLAN device MTU (maximum transmission unit, the largest packet the link will carry unfragmented) is 1450. Tailscale's disco log reports mtu=1360 on the pod path.

The obvious remedies are closed. The VXLAN /32 cannot be removed, because it is the next-hop identity every pod route depends on. Each remote /24 is routed "via 10.42.x.0 dev onlink". There is one permanent ARP entry per remote node, and there are matching entries in the bridge forwarding database. Removing the address breaks pod-to-pod routing across the whole cluster.

Tailscale 1.102.4 has no configuration lever for this either. The CLI help offers no option to exclude an interface from endpoint candidates; the only interface-related setting is the name of the TUN device. The full list of debug preference keys contains no endpoint or WireGuard preference. The advertise-routes option governs which subnets are routed through the tailnet, not which local addresses are offered as endpoints. The fix therefore had to sit outside the VPN.

3. Continuous Re-selection, Not a Latch

Our first reading on 26 September was that some peers had "latched" onto pod addresses: made one bad choice and stuck with it. The disco log does not support that reading. The excerpt below comes from a storage node's journal and concerns one peer, a GPU node.

Time (UTC)	Peer now using
20:20:07	peer's VXLAN /32 (pod path)
20:20:07	peer's node-LAN address (0 s later)
20:23:06	peer's VXLAN /32 (pod path)
20:23:06	peer's pod-bridge address (pod path)

In a single second the path moved onto the pod network and off it again. Three minutes later it moved onto one pod address and then to the other. The documentation written with the fix describes the flapping as happening every 30 to 60 seconds. That cadence is the fix author's description. We did not measure it separately.

To count the behaviour we used the daemon's journal. The detector counts lines of the form "now using 10.42." over a window. Each such line is a switch onto a pod address; lines naming LAN addresses are switches of the ordinary kind. The table gives, for one 10-minute window before the fix and one after, the pod-path switches over all path switches.

Node	Before fix	After fix
Monitoring node	15 / 28	0 / 6
GPU node (200G fabric)	6 / 19	0 / 0
Storage node A (25G)	5 / 13	0 / 2
Storage node B (25G)	4 / 14	0 / 3

Before the fix, between a quarter and a little over half of all path switches on these nodes were onto pod addresses. After the fix, switches onto LAN addresses continued on three of the four nodes (6, 0, 2 and 3), which is ordinary magicsock behaviour, and switches onto pod addresses went to zero. The fix changed which candidates could win. Path selection itself carried on.

Magicsock is not stuck. It keeps re-evaluating its candidates, and because the pod candidates work, they keep coming back into use. Any remedy that resets the selection, such as a daemon restart, will be undone by the next round of re-evaluation.

4. What the Pod Path Costs

All throughput figures were taken from the two-GPU compute node over Tailscale on 26 September. Each used iperf3 with 4 parallel streams for 6 seconds. Most are single measurements. Where a range is given it comes from a handful of runs.

Path	Endpoint in use	Throughput
Compute node to storage node A (25G)	pod path	529 to 580 Mbit/s
Compute node to storage node B (25G)	node LAN (clean)	4.65 Gbit/s
Compute node to storage node A, after a daemon restart	node LAN	4.71 Gbit/s
Compute node to storage node A, after the fix	node LAN	4.44 to 4.45 Gbit/s sustained; 5.13 Gbit/s in one verification
Compute node to a monitoring node (1G NIC)	pod path	856 Mbit/s (1G line rate)

We compared these figures in two ways. Taking the same pair before and after the fix, 529 Mbit/s against 4.44 Gbit/s is 8.4x. Taking the pod-path pair against a clean neighbour on the same tier, 4.65 Gbit/s against 529 Mbit/s is about 8.8x. Our sources summarise the cost as "about 8x" on the 25G and 200G tiers. For the 25G tier that is what we measured. For the 200G tier in September we have no recorded figure, a point we return to in Section 9.

The July occurrence was worse. On the 200G fabric, one GPU node had selected the other's pod-bridge address and measured 27 to 55 Mbit/s. Every other pair on the tailnet (the set of machines joined to the VPN) measured 6 to 9 Gbit/s. A daemon restart then re-selected the fabric address and gave 6.9 Gbit/s. We do not know why the July penalty was so much larger than September's, and we have not tried to explain it.

On the 1G monitoring tier the pod path still reached line rate, 856 Mbit/s. There the physical NIC is the ceiling, not the encapsulation, so the flapping costs no bandwidth. We deployed the fix there anyway. Constant path changes and the renegotiation that comes with them are a correctness problem even when they cost no throughput.

The clean Tailscale figure, 4.4 to 5.1 Gbit/s on 25G links, is itself far below line rate. That is the WireGuard tunnel's ceiling in this setup, and it is a separate limit. This paper does not explain it.

5. Why Every Check Stayed Green

The fault removes bandwidth without removing connectivity. Almost everything we monitor is built to detect loss of connectivity.

Signal	Reading while on the pod path
Tailscale's own ping to a pod-path peer	"pong ... in 0s"
ICMP round trip on the LAN	0.395 ms
Pod-to-pod ping across the VXLAN, after the fix	0.155 ms (0.16 ms in one source)

A double-encapsulated path across a local switch still has sub-millisecond latency. Tailscale's ping rounds it to zero seconds. The pod path is a working path by every definition a liveness probe uses (a liveness probe asks only whether something answers). The fleet sweep that found the problem also found every node Ready, Ceph at HEALTH_OK, and zero pods outside the Running state. Nothing in the health surface was red.

Only a bandwidth measurement exposed the loss. Only the daemon's journal showed its shape: a count of switches onto 10.42. addresses. Neither signal is part of an ordinary health check, and neither would have been consulted without a reason to look.

6. Three Wrong Conclusions

The fault was misdiagnosed twice before it was fixed, and a third wrong conclusion followed from the second.

The first came in July. Restarting the Tailscale daemon re-selected the fabric address, and the incident was recorded as a "trap" that "can recur after any daemon or CNI restart". CNI is the container network interface, the plugin layer to which flannel belongs. The detector written at the time grepped the daemon's status output for a direct endpoint in 10.42. That detector was not valid. The status output shows the endpoint in use at the moment it is read. Because the path flaps, a read can come back clean while pod paths are being selected and abandoned underneath it. A single read of state cannot see a fault that is a rate of change.

The second came on 26 September, and the chronology shows how it formed.

Time (UTC)	Event
about 20:00	Fleet sweep in progress. First reading: peers have "latched" onto pod addresses. Tailscale daemons restarted on the storage nodes, the GPU nodes and a monitoring node (restarts logged 20:00:30 to 20:00:33). Compute to storage node A recovers from 580 Mbit/s to 4.71 Gbit/s.
20:14	Sweep report opened.
20:14 to 20:26	The compute node logs 7 switches onto pod addresses, to 3 different peers.
20:16:52	Storage node A's daemon restarted again.
about 20:20	Storage node A is back on a pod path, about 20 minutes after the first restart.
20:25:06	Last pod-path switch on storage nodes A and B and the GPU node (20:25:37 on the monitoring node).
20:26:40 to 20:26:51	Fix applied on the nodes; boot-persistent unit activated.
20:28:14	Fix committed to our scripts repository.

The restart appeared to work. Throughput recovered from 580 Mbit/s to 4.71 Gbit/s, which is what a latch-and-reset account predicts. About 20 minutes later the node was back on a pod path, and the disco log at 20:20:07 and 20:23:06 showed why. A remedy that works and then regresses is evidence of a race between the reset and the process that caused the fault. It is not a fix. Both misdiagnoses read the symptom, one bad path at one moment, instead of the log, which showed constant re-selection.

The third wrong conclusion was specific to the monitoring tier. Those nodes also carry cluster control-plane duties, so the tier was at first "left alone" to avoid restarting daemons there. That reasoning applied only to the restart remedy. Once the remedy became a firewall rule, there was no restart to avoid, and the exception was dropped.

7. The Fix

The fix is one host firewall rule. It drops inbound UDP to Tailscale's WireGuard port when the source address is in the pod network, 10.42.0.0/16. The rule lives in its own chain, which is jumped to from the INPUT chain. Nothing else is matched. Pods, the network-policy controller and metrics exporters are untouched, and no Kubernetes workload uses that port from a pod source.

The intended effect is narrow. Pod addresses are still advertised as candidates, but a disco probe arriving from the pod network is never answered. The pod path therefore never completes discovery, and magicsock settles on the LAN or fabric endpoint. We did not try to stop Tailscale from offering the addresses, because no lever exists for that. We only made the offer fail.

The rule was deployed to every on-premises k3s node as a boot-time unit, ordered to start before the Tailscale daemon so that it would survive a reboot. Section 8 shows what that ordering did.

We verified the fix on 26 September. All five peers of storage node A converged on its LAN address, and the GPU peers converged on their fabric addresses. Compute to storage node A went from 529 Mbit/s to 4.44 Gbit/s sustained. The pod-path switch counts in Section 3 went to zero on all four nodes counted. Pod-to-pod ping across the VXLAN was unaffected at 0.155 ms, and storage node A's own pods stayed Running.

8. The Follow-up: A Control Only as Good as Its Position

On 27 September, between about 03:00 and 11:15 UTC, we made a read-only check of whether the fix had held. Nothing was changed. We read journals and rule tables on live nodes.

On the four nodes that had not rebooted since the fix (storage nodes A and B, the GPU node and a monitoring node), there were zero switches onto pod addresses from 20:26 UTC on 26 September to about 11:15 UTC on 27 September. That is about 14.8 hours. There were also zero in the last 10 minutes on each node. On these nodes the guard's jump is INPUT rule 2, ahead of Tailscale's own input chain at rule 3.

Two nodes had rebooted after the fix: a monitoring node at 03:08 UTC and the two-GPU compute node at 03:36 UTC. Both had resumed flapping, mostly against each other.

Node	Pod-path switches since reboot	Pattern
Compute node (to 11:12 UTC)	110 (113 of 117 in the 24 h window are to that one monitoring peer)	every hour, 4 to 22 per hour; 32 in the 2 hours before 11:12
Monitoring node (to about 11:15 UTC)	44	every hour, 1 to 10 per hour

The rule tables show the cause. At boot the guard unit applied first; on the monitoring node the guard was active at 03:09:02 and the Tailscale daemon at 03:09:03. The daemon then inserted its own input chain above the guard, and that chain accepts all UDP to the WireGuard port from any source. On the compute node the guard's jump sat at INPUT rule 9, behind Tailscale's chain at rule 7. On the monitoring node it sat at rule 8, behind rule 7. Tailscale's chain on the compute node had accepted 8,540 packets to the WireGuard port. The guard's DROP counter read 0.

Our ordering was therefore wrong. We had reasoned that a rule applied before the daemon would be in place when the daemon started, which is true. But the daemon installs its own rules at start-up and puts them ahead of whatever is already there. A rule meant to pre-empt a daemon's firewall rules has to be inserted after the daemon has installed them, or re-asserted afterwards. The unit was enabled, active and correct in content on both rebooted nodes, and it did nothing. The rule's position decided the outcome. Its presence did not.

The compute-to-monitoring pair is on the 1G tier. By the reasoning in Section 4 this regression is not expected to cost measurable bandwidth, and no iperf3 was run on 27 September to check. It is still the same path instability the fix was deployed to stop.

The follow-up also left one observation unexplained at the time. The guard's DROP counter read 0 on the four correctly ordered nodes too, where pod-path switches had stopped, so the counters did not confirm the mechanism we had stated.

The same-day test

Later on 27 September we ran the experiment the follow-up called for, on the two rebooted nodes only. At about 11:18 UTC we re-applied the guard, which re-inserts its jump at the top of INPUT. On both nodes it then sat at rule 2, above Tailscale's chain at rule 9. We changed nothing else.

Node	Pod-path switches, 09:18 to 11:18 UTC (guard behind Tailscale's chain)	Pod-path switches, 11:18 to 11:55 UTC (guard ahead)
Compute node	29	0
Monitoring node	7	0

At the preceding rate we would have expected about 9 and about 2 switches in a 37-minute window. The window is short, and a count of 2 expected against 0 observed on the monitoring

node is weak on its own. The compute node's drop from a rate of about one switch every four minutes to none in 37 minutes is the clearer signal. Both agree with the rule-order diagnosis.

The same session explained the zero counter. We sampled the guard's counter and the counters of Tailscale's own chain every 15 s for 4 minutes on the monitoring node. Both climbed and then fell back to zero together, about once a minute: Tailscale's chain went from 185 accepted packets to 6 between two samples, and the guard from 2 to 0. Something on these nodes rewrites the filter table periodically, and every rewrite resets every counter in it. The guard's position also moved from rule 2 to rule 3 during the test, while staying above Tailscale's chain, which fits the same rewrite. We believe the rewriter is one of the cluster's own network components, but we have not identified it. Between rewrites the guard's DROP counter read 2 to 4 packets, all UDP to the WireGuard port from the pod network. The rule is doing what we said it does. A counter that is reset every minute can only ever show the last minute, and our earlier single reads happened to land just after a reset.

9. What These Numbers Will Not Carry

The throughput figures are thin. Each is a single iperf3 run of 4 streams for 6 seconds, or a small range from a handful of runs on one evening. No raw iperf3 output was retained. What remains are the figures recorded in the session report, the lesson note and the header of the fix. These sources disagree in the third digit. The after-fix figure is 4.44 Gbit/s in the report, 4.45 Gbit/s in the lesson note and 5.13 Gbit/s in the fix header (one verification). The post-restart figure is "4.7 Gbit/s" in the fix header and 4.71 Gbit/s in the report. The pod ping is 0.155 ms in one source and 0.16 ms in another. The "about 8x" claim survives any choice among them: the ratios range from 8.1x to 9.7x.

The 8x figure comes from one pair, compute node to storage node A, with one clean neighbour for comparison. We have not shown that every 25G pair on a pod path loses the same fraction. For the 200G tier the September cost is asserted by our sources, which say "about 8x on the 25G and 200G tiers", but no September 200G iperf3 figure was recorded. The only fabric figures we have are July's, 27 to 55 Mbit/s against 6.9 Gbit/s. Those show a large penalty on the fabric, but they come from a different day and do not match the September ratio.

The switch counts are one 10-minute window per node before the fix and one after, on four nodes. They show a clear change from non-zero to zero. They do not characterise the rate over longer periods or across the other on-premises nodes. The "every 30 to 60 seconds" cadence is a description, not a measurement.

The 27 September regression was established from live reads of two nodes' journals and rule tables. Its root cause, rule order, was then tested by moving the rule and watching the count fall to zero, but over a single 37-minute window. The explanation of the zero DROP counter rests on 4 minutes of sampling on one node, and we have not identified the component that rewrites the table.

The mechanism may be specific to Tailscale 1.102.4 and to flannel in VXLAN mode. A CNI that does not put pod-network addresses on the host would not present these candidates. Another Tailscale version might enumerate candidates differently. We tested neither.

The clean-path ceiling of 4.4 to 5.1 Gbit/s on 25G links is a separate limit and is not explained here.

Some claims survive all of this. The host owns pod-network addresses under flannel VXLAN, and Tailscale 1.102.4 advertises them, as its own log shows. Peers select them repeatedly, as the journal shows. The pod path cost roughly 8x on one measured 25G pair. Latency and health checks did not show the fault. A restart was reversed within about 20 minutes. A firewall rule placed ahead of Tailscale's chain was followed by zero pod-path switches for about 14.8 hours on four nodes. The same rule placed behind Tailscale's chain was followed by 110 and 44

switches on two nodes. Several claims do not survive. We have no September measurement of the 200G cost. The measured penalty rests on one pair. The rule-order test covers only 37 minutes on two nodes. The findings may not generalise beyond this Tailscale version and this CNI.

10. What We Changed, and What Comes Next

We replaced the July detector, which read the daemon's status output, with the journal count of switches onto 10.42. addresses. That count is the only signal that saw the fault reliably. The deployed rule is in place on every on-premises node. The follow-up shows that the check around it has to assert two things: that the pod-path switch count is zero, and that the guard's jump sits above Tailscale's input chain in the rule table. Knowing that the boot unit is enabled is not enough.

The two rebooted nodes now have the guard above Tailscale's chain, and the boot unit is being changed so that it inserts its rule after the daemon has installed its own, and re-asserts it whenever the daemon restarts. The next experiments follow from the remaining gaps in Section 9. We will run iperf3 across the 200G fabric while a pod path is in use and keep the raw output, and we will identify the component that rewrites the filter table, so that the guard's counter can be read as a rate rather than a snapshot.

PureTensor operates a sovereign AI infrastructure fleet (on-premises GPU compute, storage, and serving) run day-to-day by autonomous agents under human direction. This paper is PT-R-2026-027. The measurements were taken on 26 September 2026, with the fix deployed at 20:26 UTC. They are set against an earlier occurrence on 21 July 2026, a read-only follow-up on 27 September 2026 between about 03:00 and 11:15 UTC, and a rule-order test and counter sampling later that day between about 11:18 and 11:55 UTC. Daemon journal extracts, rule tables and the session record are retained. Raw iperf3 output was not retained, and only the recorded figures remain.