

Replacing per-store AWS Site-to-Site VPN with Tailscale subnet routers

How a specialty food and beverage retailer cut its recurring monitoring connectivity cost by 89%.

CASE STUDY · NETWORKING · COST OPTIMIZATION · PCI DSS v4.0.1 · RETAIL

General educational guidance, not affiliated with or endorsed by any vendor named. Costs use published list prices as of 2026-09-09, except hardware, which is at the price paid at the time. Client details are anonymized. Published 2026-09-09.

89%

lower recurring connectivity cost

\$4,380 a year down to \$480

3.7

months to pay back the hardware

\$1,200 across 10 sites

A specialty food and beverage retailer was paying for 10 AWS Site-to-Site VPN connections that carried monitoring traffic and nothing else, and it had already reached AWS's default limit on those connections just as the business prepared to double its store count. Replacing them with a Tailscale tailnet, one Raspberry Pi per store acting as a subnet router, cut the recurring connectivity cost by 89%, and a new store adds a one-time \$120 device and nothing to the monthly bill until the tailnet passes 50 tagged resources. It also puts a Linux host in every store to patch and keep, which the cost table below does not price.

Scale	10 live retail sites, 20 planned
Devices monitored	20 to 30 per store
Retired	10 AWS Site-to-Site VPN connections and their customer gateways
Retained	SonicWall firewalls, the Nagios server, store internet circuits

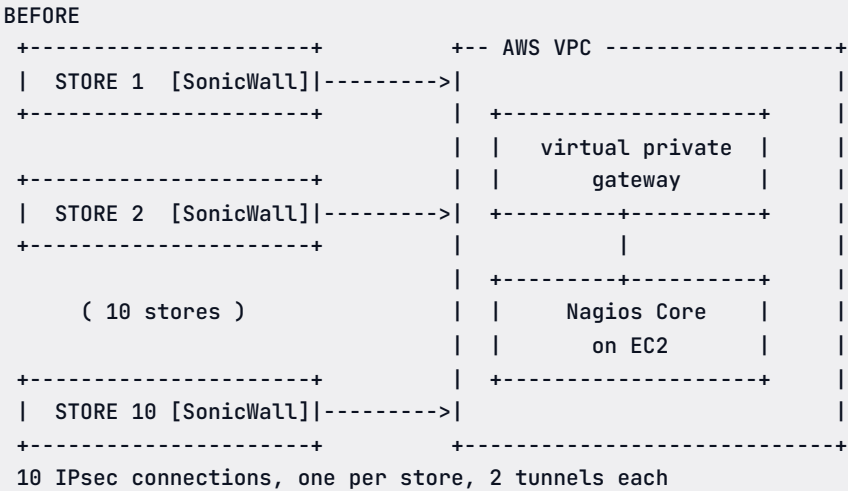
CHALLENGE

Paying for 10 VPN connections to carry pings

The retailer ran Nagios Core on an EC2 instance inside an AWS VPC (virtual private cloud, the private network an AWS account runs in), polling 20 to 30 devices in every store: IP cameras, point-of-sale (POS) systems and the payment terminals attached to them, switches, wireless access points, receipt printers, and the store firewall itself. Each store's SonicWall held an IPsec connection to AWS (IPsec: the standard that encrypts a site-to-site link), and all 10 terminated on a virtual private gateway attached to that VPC. Each connection ran over the store's main internet line only. The firewalls could fail over to a cellular backup for the store's own traffic, and the monitoring tunnels could not follow.

AWS pins a customer gateway to a static address unless it authenticates with a private certificate from AWS Private Certificate Authority on a public VPN; on that path AWS does not check the address when none is set, so the device could move to a new address without the connection being reconfigured.

That authority costs \$400 a month in general-purpose mode, on top of a charge for each certificate, against the \$365 all ten connections cost. The certificate has to come from a subordinate CA, so signing it with an AWS root rather than an external one doubles that. AWS sells a cheaper CA mode at \$50 a month whose certificates last at most seven days, and ACM will not issue from a CA in that mode at all, so a customer gateway cannot use one. The retailer stayed on pre-shared keys and left the second line uncovered. On that path AWS needs a static address for each customer gateway, which a cellular backup rarely has, and covering both lines at every store would have meant 20 connections against the 10 the next section describes. A store that lost its main line dropped out of monitoring until the line came back. Those connections carried only monitoring checks: ICMP pings, and SNMP queries that ask a device to report its own health. Each one is built from two IPsec tunnels on the AWS side for redundancy (AWS delivers every connection as a pair of endpoints, so it can service one without dropping the link), so the SonicWalls held 20 tunnel configurations between them. AWS Site-to-Site VPN is priced by the hour, \$0.05 per connection-hour for a standard connection at US East list prices and \$365 a month across the 10 stores, and that price is the same whether a connection carries heavy traffic or, as these did, kilobytes per minute.



• **Already at AWS’s default connection limit**

AWS allows 10 Site-to-Site VPN connections per virtual private gateway by default, and a gateway attaches to one VPC at a time. With 10 stores connected, the retailer was already at that limit while the business opened toward 20. Store 11 could not join without asking AWS to raise the limit, adding a second gateway and VPC, or moving to a Transit Gateway.

The second gateway would not have helped on its own, because a VPC peering connection cannot carry traffic from one VPC over another VPC's VPN, so the monitoring server would have needed its own path into the second VPC. A Transit Gateway charges \$0.05 an attachment-hour at US East list prices for every VPN connection and one more for the VPC. At 10 stores that is 11 attachments, about \$4,800 a year on top of the VPN charge already being paid. AWS marks the per-gateway limit adjustable, so the cheapest route past it was to ask for a higher one. That leaves the \$4,380 a year, the uncovered second line, and the twenty tunnel configurations where they were, so the store plan reopened the design rather than the paperwork.

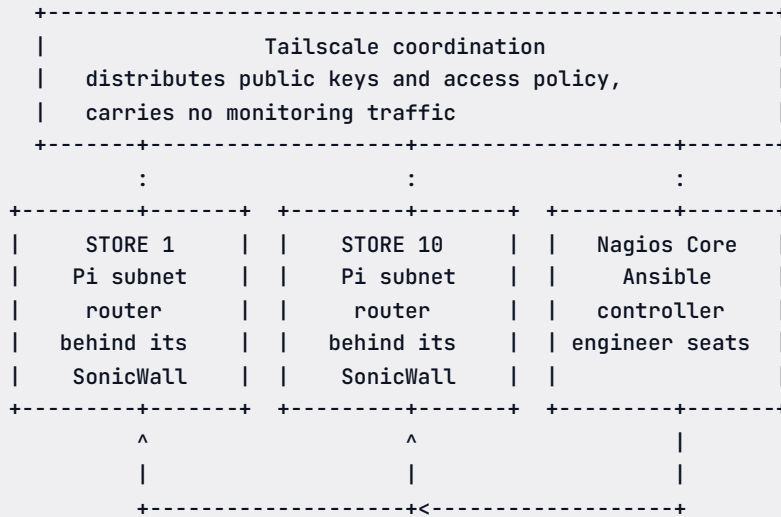
AWS has since added a VPN Concentrator, aimed at 25 remote sites or more and carrying up to 100, at \$1.95 an hour plus \$0.01 an hour for each connection through it. Every site shares its single Transit Gateway attachment, so with that attachment and the VPC one it comes to about \$18,800 a year at this store count. It attaches to a Transit Gateway rather than replacing one, and AWS supports it with neither Cloud WAN nor a virtual private gateway. Neither figure changes the comparison below.

One subnet router per store

The solution had to give Nagios a path to every device without changing settings on any of them, open nothing new to the internet at any store, run day to day with no technical staff at the store, and keep monitoring running through the migration.

Each store received a Raspberry Pi running Tailscale as a subnet router: a device that makes everything on the store's network reachable through Tailscale, with nothing installed on the devices themselves. The Nagios host joined the same tailnet (the private network Tailscale creates) as an ordinary member, advertising no routes of its own, and polled store devices at their local addresses.

AFTER



STORE 1 and STORE 10 stand for all 10. Dotted: control plane, carries no monitoring traffic. Solid: WireGuard, over DERP if there is no direct path. Only the right-hand box opens connections; routers never reach each other. That box groups the three by what they do rather than where they run

The tunnels in that diagram run WireGuard, the encryption protocol Tailscale is built on. Traffic arriving through a subnet router appears by default to come from the router itself, so polls reached every camera, terminal, and printer from an address inside the store instead of from the VPC. Where a device restricts management traffic by source, that rule has to allow the management VLAN. None of the two to three hundred here needed a change. Store devices therefore log the router's address for every session, so a poll from Nagios, a run from the Ansible controller, and an engineer's session all look alike to them. Attribution for that traffic lives in Tailscale's network flow logs rather than on the device, and those are a Premium feature the compliance section below prices.

Access runs one way. The tailnet policy lets the monitoring server open connections to the store networks and lets nothing on those networks open one back, and a camera on a store LAN is not a tailnet member, so it has no route to the monitoring server or to another store. The Pi is the device that holds that boundary, since it sits on the store network and carries a tailnet identity, so it is the one to harden and keep patched. Tailscale's own guidance for a subnet router is to leave the device's firewall denying forwarded traffic by default, so the router carries tailnet traffic and nothing else.

The policy has to name it as well: Tailscale's default lets any node in a tailnet reach any other, so without a rule for the routers' tag a Pi in a shop could open a connection to the monitoring server and to the other stores. Three things open connections into the store networks and the policy names each one: the monitoring server, the Ansible controller that builds and repairs the routers, and our own engineers, who hold seats. The engineers' access is the one an assessor asks about, since it reaches a network with payment terminals on it.

Each Pi dials out to join the mesh, so no store needs a static public address or an inbound firewall rule. Dialing out also means the connection follows whichever internet line the firewall is using, so a store that fails over to its cellular backup comes back in Nagios on its own; the retired tunnels ran only on the main line. No store now runs an inbound listener for this traffic, where each had run an IPsec endpoint before. In its place each store has a managed Linux host on its own

network, reachable over SSH from the Ansible controller and enrolled through Tailscale's coordination server, so the design trades an inbound endpoint for a device to look after.

COMPONENT	ROLE	STATUS
Nagios Core on EC2	Monitoring server	Retained, joined the tailnet
AWS Site-to-Site VPN, virtual private gateway	Store-to-cloud transport	Retired
SonicWall firewall, per store	Store perimeter firewalling, dual WAN failover	Retained
Tailscale	Overlay network, key distribution, access policy	Added
Raspberry Pi, per store	Subnet router advertising the store's VLAN subnets	Added
Ansible	Fleet build, configuration, and ongoing state	Added

COMPLIANCE

What the standard asks of this design

Payment terminals are in-scope system components for PCI DSS. Where they send account data across the store network, or a payment application runs on it, the segment and anything with unrestricted connectivity to it comes into scope with them, and so does this design. Where the store runs a validated PCI-listed P2PE solution (point-to-point encryption, where the terminal encrypts the card data and only the payment provider can read it), the terminals are the only systems handling account data, and their presence does not by itself bring the store network into scope. The standard adds that a listed P2PE solution does not remove PCI DSS from the merchant environment altogether. Settle which case applies with the assessor before the first device ships. What follows is what the standard asks of the design once it is in scope.

The standard counts a remote access server among the systems that connect to or could affect the cardholder data environment (CDE), so the subnet router, the monitoring server, and the Ansible controller belong on the inventory requirement 12.5.1 asks for, each with a description of what it does, kept current as stores open. Tailscale runs the coordination server on its own infrastructure, so whether that belongs on the same inventory is a scoping call for the assessor. Where the Ansible controller runs decides the same question for it.

Replacing the transport for every store is a significant change to the in-scope environment, which requirement 12.5.2 has the organization confirm the scope against. That same change also triggers 6.5.2, which has the organization confirm every applicable requirement is in place on the changed systems, 11.3.1.3 for an internal vulnerability scan, and 11.4.2 and 11.4.3 for internal and external penetration tests. Budget those alongside the 11.4.5 segmentation test below.

Each store's router sits on the management VLAN, which the implementation section below describes. That VLAN already routed to every other segment, and the router now gives an outside network a way into it, so what keeps the CDE apart has changed. Where segmentation isolates that environment, requirement 11.4.5 has the segmentation controls penetration tested after any change to them, on top of the test it requires at least every 12 months. The firewall change in the relay section below is a change to a network security control, which requirement 1.2.2 puts through change control. The engineers reach the store networks from outside the organization's own network, so requirement 8.4.3 puts multi-factor authentication on their accounts, and requirement 8.2.7 has a third party's access switched on only for the period it is needed, with its use monitored for anything unexpected. A standing seat does not meet the first half of that. Suspending a user in the Tailscale admin console stops their devices connecting and their API tokens and auth keys working, and vacates their seat. Vacating a seat does not reduce the bill by itself; the retailer pays for every purchased seat, occupied or not.

Suspending a user takes an Owner, Admin, or IT admin seat, so one of those sits with the retailer: 8.2.7 puts the duty on the organization being assessed, and a third party cannot switch its own access off on its behalf. A tailnet has one Owner, and it belongs on the retailer's side, with a second admin-capable person so an engineer can still be suspended

and a lost router still deleted when the first is away. The cost table prices one retailer seat, so that cover is an adopter decision.

The retailer's administrator suspends the engineers between engagements and restores them for a stated piece of work, both actions landing in the configuration audit log. Where a seat has to stand, the access policy carries the control instead: the engineers' tag is added for the work and removed when it ends, with the change recorded.

The Ansible controller signs in as a system account. Where such an account can also be used to log in interactively, requirement 8.6.1 has that use prevented except in an exceptional circumstance, approved, and attributable to an individual person. Requirement 8.6.3 applies whichever way the account is configured: its password is changed on a schedule the organization sets from its own risk analysis and on any suspected compromise, and is complex enough for how often that change happens. Where the account can be used interactively, requirement 8.6.2 also keeps that password out of scripts and configuration files, the places an Ansible controller tends to put it; keeping it out is worth doing whether or not the requirement reaches the account.

Tailscale holds the access policy for every store and relays the traffic itself whenever two peers cannot hold a direct path, though it cannot read what it relays, since the WireGuard keys never leave the devices. The engineers support the routers over remote access. Expect the assessor to treat both as third-party service providers: requirement 12.8.1 has them on a list with the service each provides, and 12.8.2 has a written agreement with each, carrying that provider's own acknowledgment that it is responsible for the security of the account data it holds or could affect. A general responsibilities clause does not meet that, and a standard set of SaaS terms rarely carries the acknowledgment.

Bringing a provider in at all is what 12.8.3 covers, which has the organization run due diligence before it engages one. Because the security work is split three ways here, 12.8.5 has the organization record which requirements each provider manages, which it manages itself, and which are shared, and 12.8.4 has it check each provider's compliance status at least once every 12 months.

Requirement 10.2.1 puts audit logging on every in-scope system component and 10.5.1 keeps at least 12 months of it, with at least the last three immediately available. Tailscale's configuration audit log covers the last 90 days in the admin console on every plan, and it records changes to the policy rather than what an engineer reached through the tunnel, which is the half of 8.2.7 that asks for their access to be monitored. Ninety days runs a little under the three months 10.5.1 wants immediately available, and the window is not configurable.

Tailscale's API exports the log, so give that export a named owner and a schedule inside 90 days, and hold the remaining nine months in the organization's own log store. Streaming it out instead, and network flow logs, are Premium and Enterprise features at \$18 a user a month, against the \$8 the cost table below uses. On the Standard plan nothing in Tailscale records what an engineer reached, so 8.2.7's monitoring half has to come from the router's own session records or the store firewall's logs. Network flow logs answer it directly, so a business taking that record from Tailscale instead of from its own hosts reads the Premium figures the cost table sets out below.

BUILD Implementation

Ansible reached each Pi over SSH with key-based authentication, and one Ansible role did the full setup for a store router: install Tailscale, enable IP forwarding, enroll the device, and advertise the store's subnets. An advertised route carries no traffic until the tailnet approves it. Approving the routers' tag in the policy, rather than each device in the console, is what keeps a new store to one inventory line.

A store router sits unattended and cannot re-authenticate itself. Tailscale disables key expiry on a device the first time it is tagged and authenticated, so enrolling each Pi with a tagged auth key settles this at build time; a device enrolled without a tag follows the tailnet default of 180 days and stops passing traffic when its key expires. Check the state on the Machines page instead of assuming either. The monitoring server is tagged for the same reason, since it runs unattended as well and an expired key there stops every store at once.

Expiry off also removes the only control that would retire that key on its own, so a lost or stolen router has to be deleted from the tailnet by hand, by a named person holding an administrator seat, inside a stated time. Inside PCI DSS scope that loss is a suspected security incident, which requirement 12.10.1 has a plan ready for. The router's tailnet identity sits in `tailscaled.state` on the SD card, and the option to encrypt that file needs a TPM (a chip that holds keys the operating system cannot read) the Pi does not have, so whoever holds the device holds the identity until it is deleted.

The tagged key the role enrolls with can enroll any other device, which then takes the key's tags, so it belongs outside the playbook. Tailscale caps a key at 90 days and will not extend an expired one, so 90 days is the outside limit on the

replacement schedule and the next store build needs a fresh key. The role describes the finished state rather than a list of steps, so running it again on a misbehaving store is safe: it either changes nothing or fixes what drifted and reports it. A store's devices sit across several VLANs (virtual LANs, the separate network segments a switch keeps apart), and a subnet router only carries traffic for a subnet it can route to, so each Pi went on the management VLAN, where the firewall routes to every other segment. That placement is what gives a \$120 device in an unstaffed shop a route to every segment in the store, payment terminals included, so the firewall rules from the management VLAN have to be scoped to the ports the monitoring uses, leaving a compromised router reaching what Nagios polls and nothing else. That scoping is what requirement 1.3.1 asks of any traffic entering the cardholder data environment, and the ports it allows, along with the outbound rule for the router, are what 1.2.5 has the organization identify and approve.

Sites migrated one at a time, and each cutover (the switch from the old link to the new one) took minutes. Every Pi was built and verified before it went near a store, so the store visit was just placing the device, confirming Nagios could poll through the tailnet, and removing the VPN connection. A network engineer on the client side did the physical work while we ran the migration remotely. Adding a store still takes hardware, shipping, and someone on site; what went away is the AWS side. The store firewall still takes the settings the relay fix below describes.

The architecture is smaller as well as cheaper. Store connectivity had been 10 customer gateways, 10 VPN connections, 20 tunnel configurations, and routing kept in step on both ends of every link, each one a thing to document, monitor, and get right during a change. It is now one tailnet policy, one Ansible role, and an inventory. A new store adds a line to that inventory and three settings on its own firewall, with no VPN configuration anywhere and nothing to change in AWS.

Each store now depends on one device where it used to depend on the firewall it already had. A Pi that fails takes that store out of monitoring, and the symptom looks the same as a store outage, so the two have to be told apart before anyone drives out to a shop. That needs one signal that does not pass through the Pi, such as a check on the store's line or its firewall that runs outside the tailnet. That check is also the first sign a router has been taken rather than has failed, and that sign starts the deletion above.

The old design's second tunnel covered AWS servicing an endpoint and never covered a failure at the store. What is new is a store-side device that can fail, and the new design carries no second one. What it has instead is a build that costs nothing to repeat: a replacement runs the same Ansible role, so recovering a store is a Pi, an SD card, and someone to plug it in.

The design also adds a vendor. Tailscale now holds key distribution and the access policy for all 10 stores, so its availability and its terms belong in the vendor register beside AWS. A coordination outage does not drop the tunnels, since no monitoring traffic passes through the coordination server and each device enforces the access policy it already holds. Until the service is back, nobody can add a store, change the policy, or delete a lost router. The routers, the monitoring server, and the Ansible controller are tagged with expiry off, so the tunnels hold rather than decaying as keys lapse.

ISSUE The relay problem

Some months in, Nagios began losing contact with every device in every store at once, recovering, and doing it again. Every store failing in the same moment meant they all depended on something shared, and the shared piece turned out to be a relay.

The store firewalls handed out a different external port depending on the destination. That rewriting is NAT (network address translation), and the kind these firewalls did was not the endpoint-independent mapping RFC 4787 asks for, so the Pis could not negotiate direct encrypted paths. It is still widely called symmetric NAT. Every one of them fell back to the same shared relay, so a brief interruption on that path took all 10 stores down at once.

`tailscale status` showed every store router connected through a relay (DERP, Tailscale's shared relay service) instead of directly, and `tailscale netcheck` reported the firewall's mapping varying by destination address, which rules out endpoint-independent mapping without separating the two address-dependent kinds. On one store, the external port changed eight times in a single hour. After the fix, the same two commands showed direct connections and stable ports. Three settings changed in the same visit, so the case for the relay is that every store was relayed through one and that direct paths held afterwards, and no test isolated it on its own.

A hard NAT at either end forces a relay unless the other end has no NAT in front of it at all, so check the cloud end before touching ten retail firewalls. Tailscale's AWS reference architecture goes further and says a node behind an AWS NAT Gateway will not hold a direct connection at all, and its connectivity matrix relays an easy NAT against a hard one as well.

AWS still maps an instance's public address to its private one, so an EC2 host is not plainly the no-NAT case, and what settles it there is the instance's security group: a peer behind a hard NAT reaches the instance directly only where that group allows inbound UDP on the Tailscale port, and Tailscale says a device on a public subnet with a public address does hold direct connections.

The monitoring server here was not behind a NAT Gateway, so the hard NAT sat at the store end. Security groups are stateful: once the store mapping was endpoint-independent, the instance could send first and the reply came back on the same flow, so fixing the store end was enough and the cloud end was never tested on its own. A peer that stays behind a hard NAT is the case that needs the group to allow inbound UDP on the Tailscale port, and that is an inbound listener on a system this design puts in PCI scope, so weigh it against the relay it removes.

• **The fix was on the firewalls**

Enabling Consistent NAT, SonicWall's name for handing out the same external port whatever the destination, let each Pi build its own direct tunnel, with the UDP timeout raised to 300 seconds so those tunnels stay open between keepalives, and the encrypted flow exempted from deep packet inspection (DPI). DPI cannot decrypt WireGuard, and SonicWall's application control still matches Tailscale by signature and can log or block what it matches; the exemption prevents that.

Consistent NAT opens no inbound port and changes no access rule. In SonicOS 7 it is one checkbox under NETWORK | VoIP, firewall-wide with no way to scope it to one application, and the article gives the paths for 6.5 and for 6.2 and earlier too. SonicWall records the trade-off as a slight decrease in overall security, because the address and port pairs become predictable, and says not to enable it unless something on the network requires it, so turn it on only at the sites that carry this traffic. RFC 4787 disagrees on the security half: it says the three mapping behaviors make no difference to the security properties of a NAT, which the filtering behavior sets instead, and Consistent NAT changes only the mapping. Record both positions for the assessor.

RFC 4787 asks the same behavior of every NAT, REQ-1 requiring endpoint-independent mapping and REQ-5 saying a UDP mapping timer must not expire inside two minutes and recommending five minutes or more, which is the 300 seconds above. SonicOS sets that timer at 30 seconds by default, under the RFC floor; it is set for the whole firewall under the UDP flood protection settings and per rule on an access rule's User & TCP/UDP tab in classic mode, or on a security action profile in Gen 7 policy mode. Put both the timeout and the DPI exemption on the outbound rule that matches the subnet router's own traffic, which leaves by default from UDP port 41641.

UPnP, NAT-PMP, and PCP, the three protocols a device uses to ask its router for an inbound port, stayed off. Tailscale can use any of them to reach a direct path, which was the alternative to Consistent NAT here, and letting every device on a retail network open its own inbound hole costs more than that path is worth.

Nagios was already polling every store, so the standing check belongs there: `tailscale status --json` names the relay for each peer, and an alert on any store router showing one would have fired months before the first outage. A store on its cellular backup may show a relay for good reason, so check the firewall's WAN state before treating one as a fault. The fix reaches only the store's own firewall: Tailscale says a change in network conditions can drop a direct connection back to a relay, so a store running on its cellular backup may be relayed whatever that firewall does.

RESULTS **What it cost, before and after**

These are list prices, so the table is a reconstruction at current rates rather than a copy of the client's invoices.

LINE ITEM	BEFORE	AFTER
VPN connections	10 x \$0.05/hr x 8,760 hr = \$4,380/yr	Retired
Overlay devices	n/a	10 subnet routers, the monitoring server, and the Ansible controller, 12 tagged resources inside the 50 included = \$0

LINE ITEM	BEFORE	AFTER
Subscription seats	n/a	4 engineers who support the routers and one retailer administrator, on the Standard plan x \$8/month = \$480/yr, priced as standing all year
Hardware	n/a	10 x \$120 one time = \$1,200
Year one	\$4,380	\$1,680
Year two onward	\$4,380	\$480

Year one cost 62% less and every year after that 89% less, a recurring saving of \$3,900, with hardware paid back in 3.7 months. Suspending the engineers between engagements frees their seats, and the bill falls only if the retailer also reduces the purchased seat count, which takes effect the following month. For an engagement inside a single month there is no saving, so the table prices all five seats as standing. At the Premium plan, the plan network flow logs and log streaming need, five seats cost \$1,080 a year, the recurring saving is 75.3% instead, and the hardware pays back in 4.4 months rather than 3.7. Tailscale's free plan includes the same 50 tagged resources and is for non-commercial use only, so a business starts at the Standard plan.

The saving grows with the store count. The business was opening toward 20 stores, and the recurring cost stays where it is: the Standard plan includes 50 tagged resources and charges \$1 a month for each one beyond that, and 20 stores means 22 tagged resources.

STORE COUNT	OLD MODEL, PER YEAR	NEW MODEL, PER YEAR
10 stores	\$4,380	\$480
20 stores	\$8,760	\$480

The 20-store figure prices 20 connections at the same hourly rate. AWS marks the 10-per-gateway limit adjustable, so it can be raised on request; the alternatives, a second gateway and VPC or a Transit Gateway attachment, cost more than the table shows.

MEASURE	BEFORE	AFTER
Config to add a site	Firewall policy, customer gateway, VPN connection, routing both ends	One inventory entry, and the store firewall's Consistent NAT, UDP timeout, and DPI exemption on the router's outbound rule
Change windows to add a site	One at the store, one in AWS	One at the store firewall
Time to migrate a site	n/a	Minutes, with the build done in advance
Inbound listeners per store	One IPsec endpoint	None
Scaling limit or price break	10 connections per gateway, 50 per Region by default, both adjustable	No limit. 50 tagged resources included, then \$1 each a month; 22 used at 20 stores

The Nagios EC2 instance costs the same as before, the SonicWall firewalls stayed in place doing store firewalling, and monitoring traffic was small enough that data transfer never counted either way. None of those appear as savings above. The after column counts the subscription and the hardware, and does not price the work of patching and replacing ten more Linux hosts and the Ansible controller, which is the running cost this design adds in place of the AWS charge it removes.

Neither column prices the change itself: the design, the Ansible role, building and shipping ten devices, and the store visits. A business running this itself counts its own engineers the same way.

Those eleven hosts also carry the standing PCI requirements: a configuration standard covering them under 2.2.1, which the Ansible role already is, patches for critical vulnerabilities within a month under 6.3.3 and the rest on a timeframe the organization sets, a place in the quarterly internal scan under 11.3.1, audit logging under 10.2.1 with 12 months kept under 10.5.1, and either anti-malware or the periodic evaluation 5.2.3 asks for.

LESSONS

What to take from this

AWS charged the same hourly rate for a connection whether it carried gigabytes or a few kilobytes a minute, so a link that exists only for monitoring is worth auditing in any business with more than a handful of sites. This retailer was already at AWS's default 10-connection limit with plans to double, and no report anyone was reading showed it, so read a provider's service quotas as closely as its prices.

Then check that each site's connection is direct, and alert on it instead of checking by hand: an overlay network keeps working when NAT traversal fails by sending traffic through a shared relay, it raises no alert of its own, and one interruption at that relay takes every site down at once. Where direct paths cannot be held, Tailscale now supports peer relays you run yourself, so the fallback lands on a machine you control instead of one shared server.

If you suspect you are paying for connectivity that is oversized for what it carries, we can help you find out. Get in touch at anobytes.com.

REF

References

AWS: Site-to-Site VPN pricing (connection-hour and data transfer rates) · aws.amazon.com/vpn/pricing

AWS: Site-to-Site VPN quotas (per gateway and per Region) · docs.aws.amazon.com/vpn/latest/s2svpn/vpn-limits.html

AWS: VPN Concentrators, their gateways and site limits · docs.aws.amazon.com/vpn/latest/s2svpn/vpn-concentrator.html

AWS: Private Certificate Authority pricing (per certificate authority, per month) · aws.amazon.com/private-ca/pricing

AWS: Private CA short-lived mode · docs.aws.amazon.com/privateca/latest/userguide/short-lived-certificates.html

AWS: VPN tunnel authentication · docs.aws.amazon.com/vpn/latest/s2svpn/vpn-tunnel-authentication-options.html

AWS: Transit Gateway pricing (per-attachment hour and data processing) · aws.amazon.com/transit-gateway/pricing

AWS: VPC peering and its edge-to-edge restriction · docs.aws.amazon.com/vpc/latest/peering/vpc-peering-basics.html

Tailscale: pricing, plans, and included resource allowances · tailscale.com/pricing

Tailscale: subnet routers, route advertisement, and source NAT behavior · tailscale.com/kb/1019/subnets

Tailscale: suspending a user during offboarding · tailscale.com/docs/features/sharing/how-to/offboard

Tailscale: node key expiry, and why a tagged device has none · tailscale.com/kb/1028/key-expiry

Tailscale: tailscaled, its state file, and the option to encrypt it · tailscale.com/kb/1278/tailscaled

Tailscale: peer relays, for a fallback path you run yourself · tailscale.com/blog/peer-relays-ga

SonicWall: Configuring Consistent NAT, article KA1VN0000000MgVOAU · sonicwall.com/support/knowledge-base/configuring-consistent-nat-network-address-translation/KA1VN0000000MgVOAU

Tailscale: firewall ports, and the default UDP port for a direct connection · tailscale.com/kb/1082/firewall-ports

Tailscale: the port mapping client, and the three protocols it uses · pkg.go.dev/tailscale.com/net/portmapper

Tailscale: device connectivity, and which NAT combinations relay · tailscale.com/docs/reference/device-connectivity

Tailscale: AWS reference architecture and NAT Gateway · tailscale.com/docs/reference/reference-architectures/aws

AWS: the NAT behind a public IPv4 address · docs.aws.amazon.com/vpc/latest/userguide/vpc-ip-addressing.html

Tailscale: configuration audit logs, their 90 days and API export · tailscale.com/docs/features/logging/audit-logging

Tailscale: network flow logs, and the plans that include them · tailscale.com/kb/1011/log-mesh-traffic

Tailscale: log streaming, and the plans that include it · tailscale.com/kb/1255/log-streaming

Tailscale: a coordination server outage · tailscale.com/kb/1091/what-happens-if-the-coordination-server-is-down

Tailscale: connection types, and dropping back to a relay · tailscale.com/kb/1257/connection-types

Tailscale: tailnet policy file, and the default that lets every device reach every other · tailscale.com/kb/1018/acls

RFC 4787: NAT behavior for unicast UDP (mapping, REQ-1, REQ-5) · rfc-editor.org/rfc/rfc4787.html

SonicWall: the UDP connection timeout, its 30-second default and where to change it · sonicwall.com/support/knowledge-base/change-tcp-or-udp-inactivity-timeout-in-gen-7-policy-mode/220510130748923

SonicWall: App Control, and its blocking and logging actions · sonicwall.com/support/technical-documentation/docs/sonicos-7-0-0-0-security_services/Content/App_Control/app-control.htm

SonicWall: the Tailscale application signature · software.sonicwall.com/applications/app/index.asp?ev=appd&appid=1889

AWS: security groups and their inbound rules · docs.aws.amazon.com/vpc/latest/userguide/vpc-security-groups.html

PCI Security Standards Council: PCI DSS v4.0.1, June 2024 (scope, significant change, 1.2.2, 1.2.5, 1.3.1, 2.2.1, 5.2.3, 6.3.3, 6.5.2, 8.2.7, 8.4.3, 8.6.1, 8.6.2, 8.6.3, 10.2.1, 10.5.1, 11.3.1, 11.3.1.3, 11.4.2, 11.4.3, 11.4.5, 12.5.1, 12.5.2, 12.8.1, 12.8.2, 12.8.3, 12.8.4, 12.8.5, 12.10.1) · pcisecuritystandards.org/document_library

PCI Security Standards Council: SAQ P2PE · listings.pcisecuritystandards.org/documents/PCI-DSS-v4-0-SAQ-P2PE.pdf