

I need you to take on the role of a researcher and network expert to investigate the feasibility of collecting flow data from network switches. The idea would be to use something like graph databases on the local switches and then send that telemetry to a main collector. The idea is to understand all network flows, both intra-switch and inter-switch. Switches can process a lot of traffic, so we must think about it at the extreme level because we want to make sure the solution scales.

Thought for 1m 1s

You can absolutely collect *flow* telemetry from switches at scale — but “understand **all** flows (intra-switch + inter-switch)” needs a precise definition, because the feasibility depends on whether you mean:

- **Routed L3/L4 flows (5-tuple)** (most practical; widely supported), vs.
- **Pure L2 switched conversations** inside a VLAN (harder; often requires sampling/packet telemetry, not true per-flow accounting), vs.
- **Full-fidelity, unsampled per-packet visibility** (generally *not* feasible from switches at extreme scale without packet capture infrastructure).

Below is what's feasible and an architecture that scales.

What switches can realistically provide today

1) Flow export (NetFlow / IPFIX / sFlow): feasible, but typically sampled

Most modern switches can export flow-like telemetry using:

- **NetFlow / IPFIX** (flow records; commonly 5-tuple based)
- **sFlow** (packet sampling + interface counters)

This is the proven way to do “flow visibility” without melting the control plane. Vendors lean heavily on **hardware-assisted sampling** to keep impact low. For example, Arista notes IPFIX sampled flow tracking uses **hardware samples**, then an EOS agent on the CPU turns those into flow records and exports them. [Arista Networks](#)

There are also “NetFlow-Lite” style approaches explicitly designed to be hardware-assisted and scalable with minimal control-plane impact. [luca.ntop.org](#)

Cisco NX-OS also supports configuring NetFlow on Nexus platforms. [Cisco](#)

Key point for “extreme scale”: if you try to do *unsampled* per-flow accounting at data-center pps rates, you will hit limits (flow table size, export bandwidth, CPU). So the scalable baseline is **sampled flows + intelligent aggregation**.

2) Streaming telemetry (gNMI/OpenConfig): great for state, not a substitute for flows

Model-driven telemetry (gNMI/gRPC, OpenConfig) is excellent for:

- Interface counters, queue/buffer drops
- MAC/ARP/route tables (depending on platform)
- Control-plane and system health

Cisco explicitly positions Nexus streaming telemetry as continuous near-real-time data streaming, with platform requirements (e.g., Nexus with **<8GB memory** doesn't support telemetry). [Cisco](#)

Cisco also documents Nexus gNMI/OpenConfig telemetry use-cases. [Cisco](#)

SONiC supports telemetry services and gNMI is commonly used to configure/stream device state in SONiC environments. [GitHub+1](#)

Key point: gNMI telemetry complements flow export. It usually won't give you "every conversation," but it gives you the switch *context* needed to interpret flows (what's connected where, what changed, what's congested).

Intra-switch vs inter-switch visibility: what you can and can't get

Inter-switch flows (across fabric)

This is the "easy" case: every switch on the path sees the traffic on ingress/egress ports, and you can export sampled flow records (or sFlow samples) and correlate centrally.

Intra-switch flows (hosts on the same switch)

This splits into two cases:

1. **Traffic is routed (hits an SVI / L3 interface):**
You can get strong L3/L4 flow visibility via NetFlow/IPFIX on the L3 boundary.
2. **Traffic is pure L2 switching within a VLAN:**
Traditional NetFlow/IPFIX won't always capture this well because there may be no L3 observation point. You typically rely on:
 - **sFlow packet sampling** (can see L2+L3 headers in samples)

- Vendor-specific L2 flow features / flexible keys (some platforms allow matching “datalink” fields as part of flow keys). [Cisco](#)

Translation: “all intra-switch flows” is feasible **as sampled visibility** (sFlow) and/or **as L3 boundary flows** (NetFlow/IPFIX). Full unsampled L2 conversation accounting is not generally something switches export at extreme scale.

The “graph database on each switch” idea: feasible only as a *lightweight edge summarizer*, not a real DB

What *won’t* scale

Putting a full graph database (Neo4j/JanusGraph-style) on every switch and ingesting per-packet/per-flow events locally will run into:

- control-plane CPU constraints
- limited storage endurance (flash writes)
- operational complexity (upgrades, failures, version skew)

What *can* work

A local, in-memory “graph sketch” / **edge cache** per switch is realistic *if* it consumes:

- **sampled flow records (IPFIX/NetFlow-Lite) and/or sFlow samples**, not raw packets
- aggregates into **time-bucketized edges** (e.g., 10s/60s windows)

Then it exports **compressed edge deltas** upstream.

This aligns with how platforms already treat telemetry: hardware samples → software agent → export. [Arista Networks+1](#)

A scalable architecture that matches your goal

Tier 0: Switch ASIC / OS (data source)

- Enable **sFlow** (for L2+L3 sampled visibility) and/or **IPFIX/NetFlow** (for 5-tuple flows where supported)
- Enable **gNMI streaming telemetry** for device/context (interfaces, neighbors, MAC/ARP/routing where available) [Cisco+1](#)

Tier 1: On-switch lightweight “FlowGraph Agent” (optional, only if you truly need edge processing)

Runs as a small process/container where supported (common in SONiC/containerized NOS patterns; SONiC telemetry services exist as components). [GitHub](#)

Responsibilities:

- Consume local flow/sFlow feed (or mirror from OS telemetry pipeline)
- Normalize into a canonical “edge event”:
 - src, dst (IP and/or MAC), service (ports/proto), direction
 - switch_id, ingress_if, egress_if, vlan/vrf, timestamp_bucket
 - counters: bytes/packets, samples_count
- Maintain an **in-memory adjacency map** for a short TTL (e.g., 1–5 minutes)
- Emit **deltas** upstream (only edges that changed)

Important: this is not a “graph DB.” It’s a bounded cache + compressor.

Tier 2: Rack/Pod collectors (strongly recommended at extreme scale)

Instead of every switch sending to a single “main collector,” use **hierarchical collectors**:

- Each collector ingests a set of switches (per rack/pod/site)
- Deduplicates, enriches (CMDB, IPAM, DHCP, identities), and forwards upstream
- Buffers during upstream outages

Tier 3: Central “Global Flow Graph”

- Stores the time-evolving graph (edges are time-series)
- Supports queries like:
 - “Show me all flows touching this workload”
 - “Show east-west dependencies”
 - “What changed in the last 10 minutes?”

Scaling reality: why sampling + aggregation is mandatory

At “extreme,” a single modern switch can forward traffic at packet rates that make per-flow/per-packet telemetry impossible to export raw.

So you design around:

- **Sampling (sFlow / sampled IPFIX)**
- **Heavy-hitter detection** (top talkers get higher fidelity)

- Time-bucket aggregation
- Edge compression + hierarchical collectors

This is exactly why hardware-assisted sampling/export exists as a “minimal impact” method.
luca.ntop.org+1

Feasibility verdict and what I'd recommend

Feasible

- Building a **global flow graph** from switches using **sampled IPFIX/NetFlow and/or sFlow**
- Including both **intra-switch** and **inter-switch** visibility at a *modeled / probabilistic* level (excellent for dependency mapping, security analytics, anomaly detection)

Not feasible (in the strict sense) at extreme scale

- “All flows” as **complete, unsampled, per-conversation truth** for all L2/L3 traffic from switches alone

Best practical approach

- Keep switches doing what they’re good at: **hardware sampling + export**
 - Use on-switch compute only for **lightweight edge aggregation** (optional)
 - Do the real graph database centrally (or in pod collectors), not on every switch
-

A concrete MVP plan (that will prove or disprove the concept fast)

1. **Pick 1–2 switch platforms** you care about (e.g., SONiC-based + a mainstream vendor)
2. Enable:
 - sFlow and/or sampled IPFIX/NetFlow [Arista Networks+1](#)
 - gNMI streaming telemetry for interface + neighbor + table context [Cisco+1](#)
3. Build a collector pipeline that:
 - normalizes events into `edge(time_bucket, src, dst, service, switch, if_in, if_out)`
 - merges edges into a global graph view
4. Validate:
 - accuracy vs packet capture on a small test segment
 - overhead on switch (CPU/mem) and export bandwidth

- how well you detect east-west dependencies inside a VLAN (sFlow will matter here)

If you tell me which switch OS/vendor family you're targeting (NX-OS, EOS, Junos, SONiC, etc.), I'll map the exact "data sources available" and the realistic upper bounds (sampling limits, export formats, on-box app support), then propose the tightest design for that environment.

Let's examine this using only Cisco Catalyst 9300s, where the majority of the switches can utilize App Hosting. We just need the metadata of the flows; we don't need the full packet. Thought for 2m 38s

Yes — with **Catalyst 9300 + App Hosting + Flexible NetFlow**, your idea is feasible *as a scalable "flow-metadata graph" system, including intra-switch and inter-switch visibility*, as long as you lean on **hardware sampling + aggregation** (not per-packet capture).

Here's how it maps specifically to **C9300**.

What the Catalyst 9300 can do for “flow metadata”

Flexible NetFlow supports *Layer 2 + Layer 3* flow types on C9300

Cisco's C9300 Flexible NetFlow guide explicitly states:

- **Layer 2, IPv4, and IPv6 traffic types are supported** [Cisco](#)
- You can apply flow monitors to **Layer 2, VLAN, Layer 3, and SVI interfaces** (no tunnels) [Cisco](#)
- On L2 interfaces you can attach `datalink flow monitor` (for non-IP / L2 visibility) and on L3 you can attach `ip flow monitor / ipv6 flow monitor` [Cisco](#)

That's the core enabler for:

- **Intra-switch** conversations (MAC↔MAC/VLAN) via **datalink/L2 monitors**
- **Inter-switch / routed** conversations (IP 5-tuple) via **IPv4/IPv6 monitors**

Sampling is hardware-supported and is the scaling lever

Catalyst 9300 Flexible NetFlow supports **hardware sampling**, with:

- **4 hardware samplers**
- sampling rates **1 out of 2 through 1 out of 1024**
- **random or deterministic modes** [Cisco](#)

There are also important hardware realities:

- NetFlow uses hash tables; **hash collisions can occur**, and effective utilization can be ~80% even with overflow CAM [Cisco](#)
This matters a lot when you're thinking "extreme scale."

Key limitations you must design around

From C9300 release notes:

- You **cannot export NetFlow using the Ethernet Management port (Gi0/0)** [Cisco](#)
 - You **cannot apply a flow monitor to L2 port-channels** (but you *can* apply to member ports) [Cisco+1](#)
 - You can't apply multiple monitors of the same type on the same interface/direction [Cisco+1](#)
-

App Hosting on the C9300: how it helps your design

Cisco's IOS XE programmability guide for Application Hosting on C9300 states:

- C9300 supports app hosting on **management interface and front-panel ports**
- **USB 3.0 SSD provides an extra 120 GB storage** for application hosting
- apps can be **control plane** (mgmt) or **data plane** (front panel) [Cisco](#)

It also calls out constraints you'll want to account for:

- "Application hosting is not VRF-aware (VRF-aware)" (i.e., routing/VRF design can constrain how the container reaches things) [Cisco](#)
- For third-party apps, dedicated storage (SSD/etc.) is a big deal; bootflash behavior differs by release [Cisco](#)

Practical takeaway: App Hosting is great for an **on-switch "FlowGraph Agent"** that ingests exported flow records locally, aggregates, and forwards summaries upstream — but it should remain lightweight.

The right interpretation of "graph database on each switch"

A full graph DB on every switch is usually unnecessary overhead. The scalable pattern is:

On-switch: *graph sketch / edge cache*

- In-memory adjacency map with short TTL
- time-bucketed "edges" (src→dst + counters + interface/VLAN metadata)

- periodic delta export upstream

Central: real graph store / graph analytics

That gives you the benefits you want (local reduction + global graph) without turning the switch into a database server.

How to capture intra-switch and inter-switch flows on C9300

1) Intra-switch (east–west on the same access switch)

Use **Layer 2 / datalink monitoring** on **access ports** (input direction). This produces metadata like:

- src MAC, dst MAC
- VLAN
- input interface (which is gold for endpoint attachment)
- bytes/packets, first/last timestamps

Cisco's guide shows the interface syntax explicitly, including datalink monitors:

```
{ip flow monitor | ipv6 flow monitor | datalink flow monitor} name [sampler  
name] {input | output} Cisco
```

You can also apply “bridged NetFlow” at the VLAN level (`vlan configuration ... ip flow monitor ... input`) [Cisco](#) — useful if you prefer VLAN-scoped capture in some designs.

2) Inter-switch (traffic that traverses uplinks / routed boundaries)

Use **IPv4/IPv6 flow monitors** on:

- SVIs (default gateways)
- uplink member ports (since L2 port-channel interfaces are excluded) [Cisco+1](#)

For a *flow graph* (who talks to whom), you typically only need **input** monitoring at the edge. Monitoring both input+output everywhere can explode volume and complicate dedupe.

A concrete “C9300 FlowGraph” architecture

On each Catalyst 9300

1. **Flexible NetFlow configuration**

- Create:
 - an **IPv4 record** (5-tuple + counters)
 - a **datalink record** (MAC/VLAN + counters)
- Create monitors and apply them:
 - datalink flow monitor ... input on access ports
 - ip flow monitor ... input on SVIs and/or uplink member ports
- Add a **hardware sampler** (start around 1/256 or 1/512 and tune)

Sampling is the primary safety valve for scale [Cisco](#), and NetFlow table behavior/limits reinforce why you want sampling + aggregation [Cisco](#).

2. Export destination = local container

- Deploy a small container via App Hosting
- Give the container an IP (AppGigabitEthernet / management vNIC approach)
- Configure NetFlow exporter destination to the container IP/port
- Remember: **you can't export via the Ethernet management port** [Cisco](#)

3. FlowGraph Agent in the container

Ingest IPFIX/NetFlow records, then:

- Normalize into edges:
 - `edge_key = (src, dst, vlan/vrf, service/proto, direction, switch_id, in_if)`
- Time-bucket (10s/60s) and aggregate counters
- Emit **deltas** to the central collector (gRPC/HTTPS/MQTT — your choice)

Central collector

- Merge edges from all switches
 - Enrich with:
 - LLDP/CDP topology
 - MAC/ARP/DHCP context (so you can map identities)
 - Write to your graph store / TSDB
-

What will make or break “extreme scale” on C9300

The knobs that matter most

- **Sampling rate** (hardware supported 1/2..1/1024) [Cisco](#)
- **Where you monitor**
 - access edge only vs everywhere
 - avoid L2 port-channel interface limitation by monitoring member links [Cisco+1](#)
- **Record size** (collect only fields you need)

- Cache timeouts (active/inactive) to reduce churn

The biggest “gotchas”

- If your campus uses lots of **L2 port-channels**, you need an operational plan to apply monitors on **member ports** consistently [Cisco+1](#)
 - You’ll hit practical limits from **NetFlow table behavior** and collisions, so you should treat this as *probabilistic visibility* at massive scale [Cisco](#)
 - App Hosting constraints (storage, VRF considerations) can affect how you wire container connectivity [Cisco](#)
-

If you want the next step

Tell me:

- your IOS XE target (e.g., **17.9.x / 17.12.x / 17.14.x**),
- whether you want to monitor **every access port** or just **SVIs + uplinks**,
- and your typical access-switch load (rough endpoints + uplink speed),

...and I’ll propose:

1. an exact **monitor placement strategy** (what to enable where to get intra+inter with minimal duplication), and
2. a **reference NetFlow/IPFIX record set** tuned for “graph edges” (minimal fields, maximum utility), using the correct C9300 CLI forms shown in Cisco’s config guide. [Cisco+1](#)

The ultimate goal of this project is to develop an application that enables customers to create a policy based on network activity. Customers complain how hard it is to establish a trustsec policy. I want to build a system that helps them achieve their desired state, understand the observed state and then reconcile everything into better policies.

Thought for 1m 30s

That’s a really good “north star” for this: **mine real network behavior into a TrustSec policy matrix**, then give customers a safe path from *today* → *desired*.

On Catalyst 9300s, the key enablers line up nicely:

- TrustSec policy is *literally* a **source-SGT × destination-SGT permissions matrix**, where each cell can contain an **ordered list of SGACLs**. [Cisco](#)
- TrustSec works as **ingress tagging + egress enforcement** (SGT carried across the domain; egress uses SGT+DGT to select the matrix policy). [Cisco](#)

- You can export **TrustSec fields in Flexible NetFlow**, including **source SGT and destination SGT** (enterprise field types like `CTS_SRC_GROUP_TAG` and `CTS_DST_GROUP_TAG`). [Cisco](#)
- C9300 App Hosting lets you run **Docker apps**, and you can use **AppGigabitEthernet** as an internal, hardware-switched interface for the application's traffic path. [Cisco+1](#)

Below is a pragmatic system design that matches your goal.

The product you're building (in one sentence)

“TrustSec Policy Copilot”: observe flows → infer dependencies and “effective access” → compare to intent → generate/validate SGACL + matrix changes → roll out safely → continuously detect drift.

1) Observed state: turn traffic into an “SGT-to-SGT communication graph”

What to export from C9300

Use **Flexible NetFlow** with a *minimal* record (metadata only). Flexible NetFlow is designed for custom records (key + non-key fields) and template-based export. [Cisco+1](#)

For your use case, the “sweet spot” fields are:

Keys (define the edge)

- `src_sgt, dst_sgt` (TrustSec fields) [Cisco](#)
- `14_proto, src_port, dst_port`
- optionally: `vrf, vlan, ingress_if` (helps with topology/attachment)

Non-keys (describe the edge)

- bytes, packets, start/end timestamps
- TCP flags (optional), ToS/DSCP (optional)

Export format:

- NetFlow v9 is fine; IPFIX (v10) is also supported by Flexible NetFlow. [Cisco](#)

A reality you must handle

C9300's TrustSec NetFlow export can output **SGT=0** in common situations (packet has no SGT, SGT=0 from a trusted interface, or IP–SGT lookup misses). [Cisco](#)

So your app needs a fallback strategy:

- If `src_sgt/dst_sgt` missing/zero → map via **IP/MAC** → **identity/SGT** using the TrustSec/ISE source of truth (or local SGT mappings), then backfill those edges.
-

2) Desired state: capture intent in a way customers can actually answer

This is where most TrustSec projects stall—people can't translate business intent into SGT taxonomy + matrix.

A workflow that tends to work:

1. **Define assets and roles (human language)**
 - “Finance Users”, “HR Apps”, “Domain Controllers”, “Printers”, “IoT Cameras”, “Contractors”, etc.
 2. **Define “allowed outcomes”, not ports**
 - “Finance Users can access ERP”
 - “Workstations can talk to DNS/NTP/AD”
 - “Printers only accept print from user subnets”
 - “IoT can reach only its controller + DHCP/DNS”
 3. The system translates outcomes into:
 - recommended **SGTs**
 - recommended **SGACL building blocks** (reusable “contracts”)
 - recommended **matrix entries** (SGT→SGT with ordered SGACL list) [Cisco](#)
-

3) Reconciliation: convert “what happened” into “what should be permitted”

Once you have a graph of observed edges, the reconciliation engine does 4 things:

A) Build the “Observed Policy Matrix”

For each `(src_sgt, dst_sgt)` pair:

- list top services (dst ports/proto) and volumes
- mark confidence (based on sampling rate + duration)
- highlight “rare” vs “persistent” flows

B) Propose policy deltas

For each matrix cell, propose:

- **Allow-list SGACL** based on stable dependencies
- **Deny** for everything else (default stance)
- **Exceptions** with expiration (temporary rules for migration)

C) Identify “tagging problems” vs “policy problems”

Common TrustSec pain:

- wrong SGT assignment makes the matrix look “too open” or “too broken”

Heuristics:

- If a host’s SGT changes frequently → identity/tagging issue
- If an SGT talks to “everything” → SGT taxonomy too coarse
- If SGT is often 0 in exports → tagging propagation/gaps (needs TrustSec domain health checks)

D) Generate “explainable” recommendations

Every suggested matrix change should include:

- top flows that justify it (counts, time window)
- impacted endpoints/apps
- risk/impact score (“this blocks AD/ DNS / critical app?”)

4) Safe rollout: prove policies before enforcing them

This is a must-have feature if you want customers to trust the system.

Catalyst supports **SGACL Monitor Mode**, including a global monitor mode and per (SGT, DGT) monitor enablement. [Cisco](#)

So your workflow can be:

1. **Propose changes** (new SGACLs + matrix updates)
2. **Deploy in monitor mode first**
3. Collect hit counts / would-have-dropped stats
4. Only then flip to enforce

This turns policy rollout into an iterative “tighten the net” loop instead of a big-bang outage event.

5) Where App Hosting fits (and where it shouldn’t)

App Hosting is best used for **edge reduction + local resiliency**, not heavy analytics.

On C9300:

- App hosting supports **Docker apps** [Cisco](#)
- AppGigabitEthernet can be configured as trunk/VLAN-specific and is hardware-switched to front-panel ports [Cisco](#)

Practical pattern:

- On-switch container runs a **Flow Edge Aggregator**:
 - receives NetFlow records locally
 - aggregates into time buckets (e.g., 30s/60s)
 - ships compressed deltas upstream
- Central system does:
 - long-term storage, graph DB, policy mining, UI, approvals, push-to-ISE

Also note app hosting constraints you’ll design around:

- It’s **not VRF-aware**, and it expects underlay/SVI connectivity patterns. [Cisco](#)
-

MVP scope that proves the concept fast

MVP-1: “Observed Matrix”

- Ingest FNF with TrustSec fields (SGT/DGT) [Cisco](#)
- Build:
 - SGT→SGT matrix heatmap
 - per-cell top services
 - “unknown SGT/0 SGT” troubleshooting view

MVP-2: “Suggested Policy”

- Generate SGACL suggestions per matrix cell (allow-list)
- Export as:
 - human-readable proposal report

- machine-readable intended matrix config

MVP-3: “Validate then Enforce”

- One-click deploy to **monitor mode** for selected cells [Cisco](#)
 - Show “would deny” counters + affected apps
 - Promote to enforce when clean
-

If you want, I can turn this into a **reference architecture + data model** (the exact edge schema, time-bucketing strategy, and the recommendation algorithm outline), scoped specifically to “C9300 + ISE TrustSec policy matrix”, including how to handle SGT=0 cases cleanly.

One thing to understand is that in most cases, customers will not have any SGT defined, etc. The goal is to help the customer define and or refine their trustsec deployments.

Thought for 20s

Totally — in fact, “**no SGTs exist yet**” is the *normal* starting point, and your application’s real value is getting customers from **identity chaos → a working TrustSec model** without needing them to be TrustSec experts.

Here’s a concrete way to design the system so it works when **SGTs are zero/unknown**, then smoothly transitions into a true **SGT→SGT policy matrix**.

The right mental model: 3 phases, one continuous workflow

Phase 1: Observe reality without SGTs

Start by building a **ground-truth activity graph** from flow metadata (MAC/IP/port/proto/VLAN/interface) — no TrustSec required.

Phase 2: Propose + apply an initial SGT taxonomy

Your app recommends a small set of **Security Groups (SGTs)** based on what it observed, then helps the customer **tag** those assets using low-friction methods first.

Phase 3: Generate and refine the TrustSec policy matrix

Once assets have SGTs, you can drive policy as intended: **SGACL + matrix** (ideally managed in ISE), with a safe “monitor first” rollout.

Cisco's TrustSec overview describes exactly why this model works: policy is based on identity/group, not IP, and SGACLs govern access **between security groups**. [Cisco+1](#)

Phase 1: Build “Observed State” even when SGT = none

What you collect from Catalyst 9300

You only need metadata, so export flow records with:

- src/dst IP (and MAC when possible)
- L4 proto + ports
- VLAN, ingress interface, switch ID, timestamps, bytes/packets

This yields a graph like:

```
(endpoint A) -[tcp/443, vlan 20]-> (endpoint B/service)
```

What your app produces immediately (before TrustSec exists)

Outputs customers will love:

1. **Application dependency map** (who talks to what, and on which services)
2. **“Allowed vs suspicious” baseline** (steady dependencies vs rare/one-off)
3. **Blast radius preview** (“if you segment printers, what breaks?”)

This becomes the factual backbone for policy.

Phase 2: Help them *create* SGTs and get tagged (with low-friction options first)

Customers struggle because TrustSec “policy” is useless until **tagging** exists. Your system should provide a **tagging maturity ladder**.

Tagging ladder (start easy, get better over time)

Level 0 → Level 1: Subnet-to-SGT mapping (fastest on-ramp)

Catalyst supports **Subnet-to-SGT mapping**: bind an SGT to all hosts in a subnet, and TrustSec imposes that SGT on incoming packets whose source IP is in that subnet. [Cisco+1](#)

Your app can generate these mappings as a *starter kit*:

- “User VLANs → User SGT”
- “Server subnets → Server SGT”
- “Printers VLAN → Printer SGT”
- “IoT VLAN → IoT SGT”

This gives customers **immediate tagging coverage** with minimal project effort.

Level 1 → Level 2: Identity-driven tagging (the “real” end-state)

Then the app helps them transition to ISE-driven tagging (802.1X/MAB/posture/etc.), where SGT assignment is based on identity and context (more precise than subnet buckets). (TrustSec is fundamentally about assigning devices/users to security groups and enforcing between groups.) [Cisco+1](#)

Key product insight: your app should treat subnet-to-SGT as an *accelerator*, not the finish line.

How your app recommends an initial SGT taxonomy

Don’t start with 50 groups. Start with **6–12** that map to common trust boundaries:

- Users (corp managed)
- Guests / BYOD
- Servers (apps)
- Domain services (AD/DNS/NTP)
- Printers
- IoT / OT
- Network infrastructure (if needed)
- Admin workstations (if needed)

Your system proposes these by clustering observed endpoints using:

- VLAN/subnet membership
- switch/port location (edge vs uplink)
- behavior (who they talk to, which services)
- naming/DHCP/NetBIOS hints if available

Then it generates “candidate SGTs” + “what endpoints belong” + “confidence score”.

Phase 3: Generate better TrustSec policy from observed behavior

Once SGTs exist, your graph becomes an **SGT→SGT communication graph**, which maps directly to how TrustSec is implemented: a **policy matrix** (source SGT × destination SGT) enforced using SGACL. [Cisco+1](#)

What your app outputs

1. Observed Matrix

- For each (SGT_src, SGT_dst): top ports/protos, volume, and “persistence”

2. Recommended Policy

- For each matrix cell: recommended allow-list SGACL rules
- “Default deny” posture (with explicit exceptions)
- Human explanation: “Why is this allowed?” (top flows + endpoints)

3. Drift detection

- “Observed state deviates from desired state” alerts

Safe rollout: monitor mode first

Catalyst supports **SGACL Monitor Mode**, specifically to test policies without enforcing them and to identify policies that don’t function as intended before enforcement. [Cisco+1](#)

Your workflow should be:

- Stage policy → enable monitor mode for chosen cells → review “would drop” hits → refine → enforce

(And if needed, you can enable/disable enforcement globally/interface using `cts role-based enforcement`.) [Cisco](#)

What makes your product “feel magical” to customers

1) A guided “TrustSec from zero” wizard

- “What are your major zones?” (users/servers/IoT/etc.)
- Import VLANs/subnets
- Auto-generate **Subnet-to-SGT mappings** as a first deployment [Cisco](#)

2) A “what breaks if I enforce this?” simulator

- Before any enforcement, show:
 - impacted flows
 - impacted endpoints
 - predicted business impact

3) A policy hygiene engine

- detect over-broad groups (“Users talk to everything”)
 - suggest splitting SGTs when behavior clusters diverge
 - detect mis-tagging / missing tagging (large unknown bucket)
-

A practical MVP roadmap (Catalyst 9300-focused)

MVP 1 — Observed State

- Flow ingestion + dependency graph
- Top talkers + top services
- VLAN/subnet grouping

MVP 2 — Tagging Bootstrap

- SGT catalog + tagging recommendations
- Generate Subnet-to-SGT mappings [Cisco](#)
- Show “coverage” (% of traffic attributable to a proposed SGT)

MVP 3 — Policy Matrix

- Build observed SGT→SGT matrix
- Generate SGACL proposals (ISE-oriented) [Cisco+1](#)

MVP 4 — Validate and Rollout

- Monitor-mode deployment workflow [Cisco+1](#)
 - Enforcement promotion (with rollback plan)
-

If you want, I can propose the **exact minimal edge schema** (what fields to keep so you can build matrix + explainability without bloating storage), plus the **recommendation algorithm** for “SGT creation from zero” (including the heuristics that decide when to suggest splitting/merging groups).

One thing that I left out is "Identity", what are these devices, users, etc? This project must also incorporate data from external sources, to build an identity mapping, to truly understand what is

talking to what, beyond the IP address. For example, pulling in data from ISE, AD, and other identity sources.

Thought for 38s

You're exactly right: without **identity**, “flow telemetry” tops out at *IP:port talks to IP:port*, which is not what customers need to build TrustSec. The core of your product becomes an **Identity Resolution + Policy Mining engine** that continuously turns raw flow metadata into:

User / device / app / location → communications → recommended TrustSec groups + policy

Below is a practical, scalable architecture for doing that with Catalyst 9300 telemetry *plus* ISE + AD (and room for other sources).

1) What “identity mapping” must produce

For every observed flow edge, your system should be able to answer:

- **Who** initiated it? (user identity + user groups/roles)
- **What device** did it come from? (endpoint identity: MAC, hostname, posture/type, ownership)
- **Where** is it? (switch, port, VLAN, site, building)
- **What is the destination** really? (application/service identity, not just an IP)
- **Confidence + time validity** (because identity changes with DHCP, roaming, VPN, etc.)

This needs to be **time-versioned**, not static: *the same IP can be a different device tomorrow.*

2) Cisco ISE is your “real-time identity oracle” (even before SGT exists)

pxGrid Session Directory = live user/device ↔ IP/MAC/context

Cisco documents pxGrid Session Directory as providing **active session info** including **user identity and context** such as **IP address, user group, location, and device type**, and you can retrieve it via query/bulk download and notifications. [Cisco DevNet+2](#)[Cisco DevNet+2](#) Cisco’s ISE admin guide also describes pxGrid as sharing this session-directory context with other systems. [Cisco](#)

Use it like this (core join):

- **Flow telemetry** gives you: (src_ip, src_mac?, ingress_if, vlan, time)

- **pxGrid Session Directory** gives you: `src_ip → (username, endpoint_mac, endpoint_profile, location, groups, posture, ...)`
- Result: you can label a flow as “**Steve (Finance) on LAPTOP-123 in Building A → SAP on tcp/443.**”

ISE ERS / OpenAPI = inventory + grouping metadata

Use ISE’s REST APIs for “static-ish” objects:

- Endpoint groups (MAB groupings, IoT buckets, etc.) are exposed via ISE APIs. [Cisco DevNet](#)
- TrustSec groups are defined in ISE; ISE assigns users/devices to security groups in the SGA domain. [Cisco](#)

Why this matters early (before SGT):

- Even if customers haven’t built SGTs, many *do* have ISE sessions, endpoint profiles, authZ results, and endpoint identity groups. That’s already “identity.”
-

3) Active Directory is your “authoritative identity backbone”

AD/LDAP (or Entra ID later) gives:

- **Stable user identifiers + naming attributes** (UPN, sAMAccountName, DN, etc.) [Microsoft Learn](#)
- **Stable security principals** (SIDs) that remain consistent even if display names change. [Microsoft Learn](#)
- Group membership (Finance, HR, Admins) — which is what customers *actually* want policy to align to.

Use AD to enrich ISE identities:

- pxGrid says “username = jdoe” → AD says “jdoe ∈ Finance, SOX-Users, SAP-Users”
 - That lets your product propose TrustSec groups that map to *business roles*, not VLANs.
-

4) Identity resolution engine: how you map “IP flows” → “who/what”

A. Maintain a time-aware “Identity Binding Table”

Think of this as a continuously updated cache keyed by IP/MAC:

Bindings (examples):

- IP → MAC (time-bounded)
- MAC → Endpoint (hostname, device type/profile, endpoint group)
- IP/MAC → User (username, AD groups) from pxGrid session
- MAC → Switch/Port/VLAN (from flow ingress_if + topology data)
- IP → Application (reverse DNS, CMDB, APM tags, static “known services” list)

B. Use confidence scoring + conflict rules

Identity data will disagree. You need deterministic rules:

- **Most recent session wins** (within TTL)
- **Authenticated session > passive inference**
- **ISE session identity > DNS hostname guess**
- Flag collisions (“IP reused”, “MAC randomization”, “shared workstation”)

C. Emit “Identity-labeled edges” into your graph

Instead of storing only:

- 10.1.2.3 → 10.9.8.7 : tcp/443

Store:

- (User:Steve, Device:LAPTOP-123, Location:NYC-3F) → (App:SAP-Prod, Tier:Prod, Owner:ERPTeam) : https

This is the unlock for policy generation.

5) How identity powers the TrustSec “from zero” journey

When customers have **no SGTs**, your product should drive a staged adoption path:

Stage 1 — Build a business-readable observed graph

- By **AD groups, ISE endpoint profiles, site/location, device class**
- Output: “Finance users talk to ERP + AD + DNS; IoT talks only to controller + DHCP/DNS; printers accept 9100 from Users only.”

Stage 2 — Recommend an initial TrustSec taxonomy (small!)

- Suggest 6–12 groups derived from behavior + identity:
 - Finance Users, Corp Users, Guests, Admin Workstations
 - ERP Servers, Domain Services, Shared Services
 - Printers, IoT, Network Infra

Stage 3 — Bootstrap tagging quickly

Even early, some customers want a fast-start option:

- **Subnet-to-SGT mapping** is a known on-ramp for basic tagging coverage (bind SGT to all hosts in a subnet). [Cisco](#)
 Your product can generate these starter mappings *while pushing them toward identity-based tagging over time.*

Stage 4 — Generate matrix + SGACL from observed identity-labeled dependencies

Once SGTs exist, you transition from “identity-labeled edges” to an actual **SGT→SGT observed matrix**, and your recommendations become directly implementable TrustSec policy.

6) External sources beyond ISE + AD (optional but huge value)

If you want “what is that destination IP really?”, plan connectors for:

- DNS/DHCP (hostname + lease history)
- CMDB / ServiceNow (app ownership, environment tags)
- EDR (CrowdStrike/Defender) for device identity + posture
- Cloud identity (Entra ID / Okta) for group/role truth
- App telemetry (APM) for app-to-app dependency truth

You don’t need these for MVP, but they dramatically reduce “unknowns.”

7) MVP that proves identity + policy value (fast)

MVP-1: Identity-labeled flow graph

- Flow ingest (C9300 FNF)
- pxGrid session pulls + subscriptions (for near-real-time updates) [Cisco DevNet+1](#)
- AD/LDAP enrichment (users + groups)

- UI: “Who/What talks to What” + “Unknown identity bucket”

MVP-2: TrustSec readiness + taxonomy recommender

- Recommend 6–12 groups
- Show tagging coverage estimates (“% of flows attributable to recommended groups”)
- Generate starter subnet-to-SGT mappings (optional fast-start) [Cisco](#)

MVP-3: Policy recommender

- Suggested SGACLS per group-to-group dependency
 - “Why” explanations: top flows + affected users/apps + confidence
-

If you tell me what the **minimum** customer environment is for your MVP (e.g., “ISE is present but only MAB”, “no 802.1X yet”, “AD only”, “DNA Center present or not”), I’ll give you a crisp **data contract** (exact fields you ingest from flows + pxGrid + AD), plus a **canonical identity graph schema** you can implement (nodes/edges, TTLs, conflict resolution, confidence scoring).

The challenge is getting sample data; I don’t have a live lab with switches. I would like to provide the concept first with a large amount of artificially created data. Data that would have come from Switch, AD, and ISE. Think about a campus with 100 switches, 10,000 employees. Can we generate CVS or find them on the Internet? I would like to build the data lake(s) with the aim of emulating a real-world scenario. I’m not worried about getting data of the boxes for now, I just to be able to use the data, to build the backend to see if we can make a system that would a true trustsec copilot for customers to adopt, use.

Thought for 1m 10s

Thought for 24s

Thought for 9s

Thought for 2m 45s

You’re not going to find a **single** public dataset that looks like “campus flow metadata + ISE session identity + AD directory” end-to-end. That combination is usually too sensitive to publish. What *does* exist publicly are (a) flow/traffic datasets (often security-oriented), and (b) vendor API docs so we can model realistic identity/session fields—then we stitch it together with synthetic generation.

On the **Catalyst 9300** side, the practical way to get “flow metadata (not packets)” is typically **Flexible NetFlow / IPFIX export** (records/monitors/exporters) rather than trying to persist a full graph DB on every switch. [Cisco+1](#)

And for your “run something on the switch” angle, **App Hosting** on Catalyst 9000 (incl. 9300) is the supported containerized model. [Cisco+1](#)

Option A: Use public flow datasets (good for realism, not for ISE/AD joins)

If you want *realistic* distributions of bytes/packets/ports/app mixes, you can borrow from:

- **CIC** flow datasets (CICFlowMeter-derived CSVs are widely used in research). [GitHub+1](#)
- **UNSW-NB15** (popular labeled network dataset with flow-like features). [UNSW Sites](#)
- **CAIDA** traces (packet traces; you'd convert to flows yourself). [CAIDA](#)

These help make your “flows.csv” realistic, but they won’t come with campus topology, switchport location, AD users/groups, or ISE session context.

Option B: Generate a complete synthetic “campus + identity + flows” lake (best for your backend)

I generated a **ready-to-ingest synthetic dataset** that matches your scenario:

- Campus: **100 switches**
- Directory: **10,000 employees (AD-like)**
- Endpoints: laptops/phones/printers/IoT/servers
- Identity: **ISE session directory-style sessions** (username, auth method, endpoint profile, location, etc.)
- Flow telemetry: **5-tuple + byte/packet + exporter switch + ingress interface + timestamps + (mostly missing) SGTs**

Download: trustsec_copilot_synth_campus.zip

What's inside (CSV files + counts)

- switches.csv (100)
- interfaces.csv (5,440)
- ad_users.csv (10,000)
- ad_groups.csv (18)
- ad_group_membership.csv ($\approx 23k$)
- endpoints.csv (13,650)
- ip_assignments.csv (13,000)
- ise_sessions.csv (13,300)
- services.csv (42) (AD/DNS/DHCP/ERP/HRIS/FileShare/WebProxy hosted on “servers”)
- trustsec_sgts.csv (9) (seed/proposed; endpoints mostly Unknown)
- flows.csv (106,814)
- flow_truth.csv (106,814) (*optional “ground truth” joins for testing your enrichment pipeline*)
- DATA_DICTIONARY.csv

Why the ISE session fields are shaped this way

Cisco pxGrid Session Directory “getSessions” exposes fields like timestamps, MAC, IPs, username, NAS details, endpoint profile, TrustSec security group fields, etc. [Cisco DevNet](#)
My synthetic `ise_sessions.csv` is a simplified, CSV-friendly version of that concept so you can build your joins and enrichment logic now, then swap in real pxGrid/ERS data later.

How to use this dataset for a “TrustSec Copilot” backend

This lets you prototype the exact thing you care about: **Observed state → identity-resolved graph → policy suggestions → reconciliation**.

The core joins you’ll implement

1. **Flow → device**
 - o `flows.src_ip + flows.src_mac`
 - o join to `ip_assignments` (`ip/mac → device_id`)
2. **Device → user (when known)**
 - o `endpoints.device_id → owner_user_id`
 - o join to `ad_users` and `ad_group_membership`
3. **Flow → service**
 - o `flows.dst_ip → services.ip → service_id/service_name`
4. **Optional: “location context”**
 - o `flows.exporter_switch_id + flows.ingress_interface`
 - o correlate with `ise_sessions.location` patterns

The graph you can build immediately

Nodes (examples):

- User, Group, Device, Switch, Port, Service, IP

Edges (examples):

- `(User) -[:OWNS] -> (Device)`
- `(Device) -[:ATTACHED_TO] -> (Port) -[:ON] -> (Switch)`
- `(Device) -[:TALKS_TO {bytes, pkts, proto, ports, time}] -> (Service)`
- `(User) -[:MEMBER_OF] -> (Group)`

Scaling the synthetic data up

The current ZIP is intentionally “workable on a laptop.” If you want “extreme scale” simulation (tens/hundreds of millions of flows), you’ll keep the same schemas and:

- increase flows per endpoint per day

- extend time range to weeks/months
- add more sites/subnets/services and more “internet” destinations
- emit in **partitioned Parquet** (date/hour) instead of CSV for performance

If you want, tell me what “extreme” means for you (e.g., **flows/sec**, retention window), and I’ll give you a concrete sizing model and partition strategy that matches how Flexible NetFlow/IPFIX export from 9300s behaves in the real world.