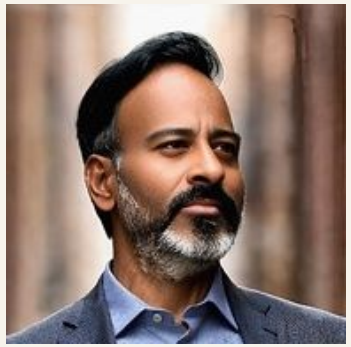
Scaling OpenTelemetry in a Financial Enterprise

Lessons from Capital One's Journey



Sateesh Mamidala
Staff Engineer
Capital One



Chandra Siva
Sr. Staff Engineer
Capital One

Five Years to One of the Largest OTel Footprints

From an early proof of concept to a vendor-agnostic, production grade observability platform, running across one of the largest cloud-native environments in financial services.

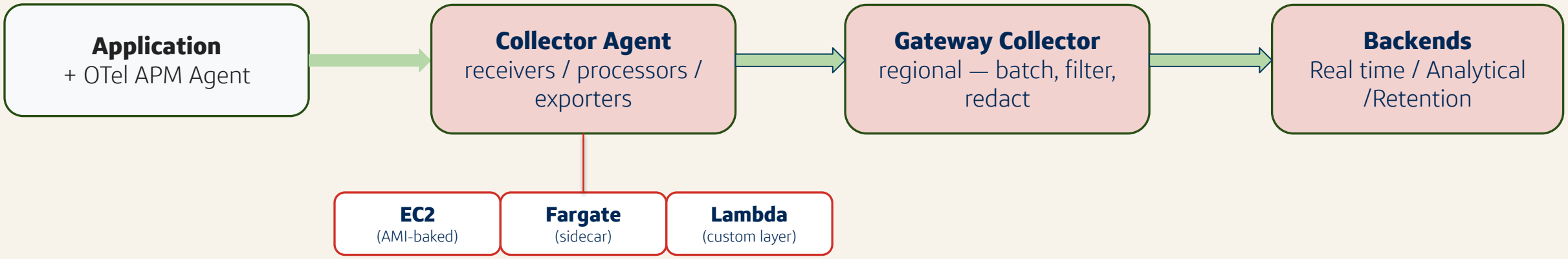
85,000+ compute instances
Each running the OTel Collector agent

Lambda invocations
Instrumented through a custom Lambda layer

Thousands of containers
Fargate and Kubernetes workloads

Five-year journey
Proof of concept → enterprise-standard platform

The Platform — Gateway Collector Architecture



Layer	What it is	Where it runs
OTel APM Agent	Auto-instrumentation libraries generating traces/metrics	Java image; packages for Python, Node.js, PHP, .NET; manual-only for Go
OTel Collector Agent	Receivers/processors/exporters — same binary as the Gateway, configured differently	AMI, Sidecar on Fargate, custom Lambda layer for Lambda
OTel Gateway Collector	Regional aggregation point, batches, filters, redacts, exports	Auto Scaling Group, Multiple AZ's, deployed in multiple regions
Backends	Multiple	Multiple exporters configured per Gateway



High-Trust Enterprise vs. Standard Software Development

Why this matters here: every guardrail we built into the OTel platform traces back to one of these differences.

Dimension	Standard Default	High-Trust Enterprise
Audit trail	**For debugging:** ad-hoc developer logs, rapid troubleshooting	**Evidence for examiners:** who, what, when, why, and approved-by
Change management	**Merge → deploy:** developer autonomy, single peer approval	**Segregation of duties:** automated risk sign-off, change advisory gates
Data handling	**Policy-driven:** internal developer guidelines, lightweight privacy controls	**Mandate-driven (GLBA/SOX/PCI-DSS):** residency, PII scrubbing at source
New tooling & vendors	**Add a package:** direct npm/pip install, self-serve SaaS adoption	**Third-party risk review:** data-flow mapping, legal contract & infosec sign-off
Record integrity & retention	**Best-effort logging:** occasional data loss is acceptable	**Records are legal evidence:** deliberate retention policy; zero silent loss
Rollout pace & blast radius	**Feature-flag straight to prod:** live iteration, rapid hotfixes	**Staged rings:** Non-prod → Prod gated by golden AMI rehydration

How This Maps to Observability

Each constraint becomes a specific design decision in the OTel platform:

1. **Vendor/third-party risk**
   Favor extending open-source OTel agents over commercial APM/SaaS agents

2. **Phased rollout & Blast Radius**
   Staged promotion, blast-radius limits at all layers

3. **Data handling (PII/PAN/residency)**
   Redaction/scrubbing before data enters Monitoring platforms

4. **Record integrity & retention**
   Telemetry is evidence, what gets kept is a deliberate, policy-driven choice

5. **Audit trail and Change management**
   Collector/pipeline changes get versioned review and staged rollout, not just app code

1



Vendor/third-party risk

Favor extending open-source OTel agents over commercial APM/SaaS agents



How We Closed Gaps

Agent Extensions — Closing Visibility Gaps

Extended the OTel Java agent with three more capabilities that close visibility gaps commercial APM tools cover

Spring framework instrumentation

Captures spans for Spring's internal framework methods, not just HTTP entry points, surfacing the full in-framework call path the stock agent leaves invisible

Per-thread metrics

Reports state for each individual thread rather than one aggregate figure across the pool, so you can see exactly which thread is blocked, running, or waiting, not just a JVM-wide total

Continuous JFR profiling

A custom Java Flight Recorder extension streams always-on, low-overhead JVM profiling through the OTel pipeline , code-level hot-spot visibility without a separate profiler

Fixing OpenTelemetry's Lambda Cold-Start Tax

Three changes cut cold start, response time, and export errors

Cold start — Decoupled Processor

Switched from synchronous ADOT export to the open-source layer's decoupled processor: the function returns immediately, telemetry flushes in the background. Traded about 1 second of extra cold start for freedom from Gateway-side failures.

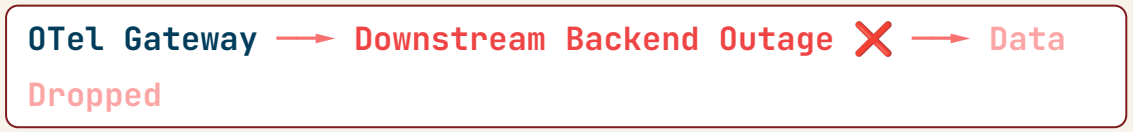
gRPC to HTTP/1.1

Replaced the heavier gRPC exporter with HTTP/1.1, cutting cold starts by roughly 180–350ms depending on the stack

Export errors — Greedy TLS Handshake

Lambdas open 3–5x more connections to the Gateway than typical workloads, causing TLS negotiation errors under our export timeout. Pre-warming the handshake instead of doing it lazily cut TLS errors by about 60%.

New Capability — Contributed the Failover Connector Upstream





Without Failover

Availability Capped: Telemetry pipeline SLA is hard-capped to the lowest common denominator among downstream storage backends.

Cascading Timeouts: Backend rate-limits or temporary degradation cause exporter timeouts and memory backpressure.

Manual Recovery: Incident resolution requires engineer redirection of pipelines

With failoverconnector (Upstream Open Source)

Dynamic Circuit Breaking: Automatically senses health/timeouts on the primary pipeline and reroutes spans to an alternate target

Automated Data Continuity: Telemetry flow stays intact; backfill routines run when the primary incident clears.

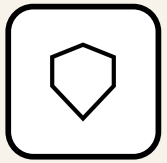
Enterprise Availability Decoupled: Gateway SLA can now exceed individual downstream backend SLAs.

2



Phased rollout & Blast Radius

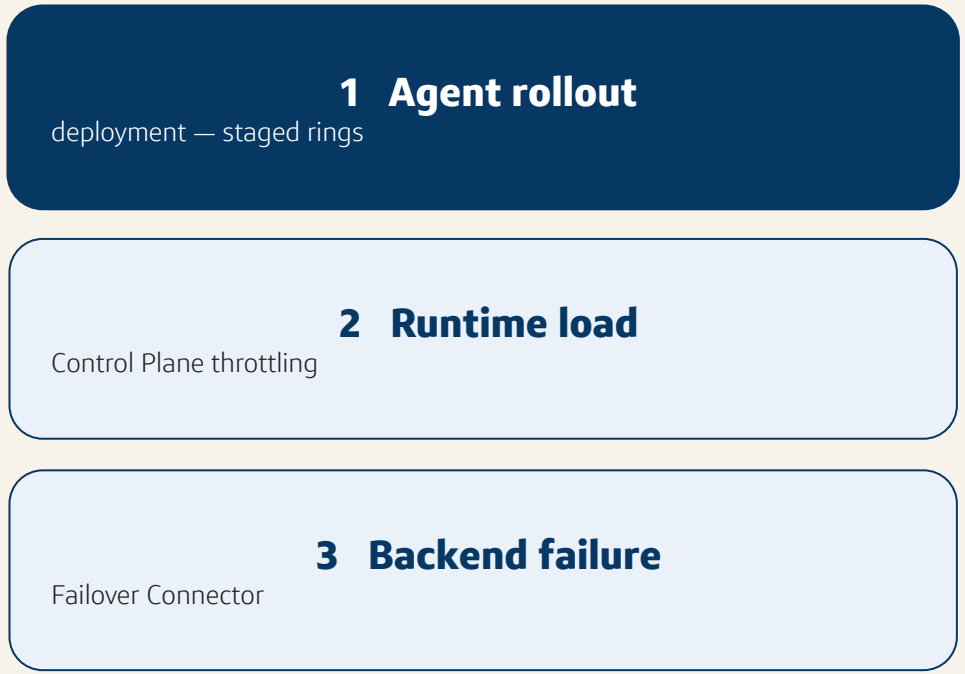
Staged promotion, blast-radius limits at all layers



Containing Blast Radius

Blast Radius — Layer 1: Agent Rollout

The failure contained here: a single bad agent version going everywhere at once.



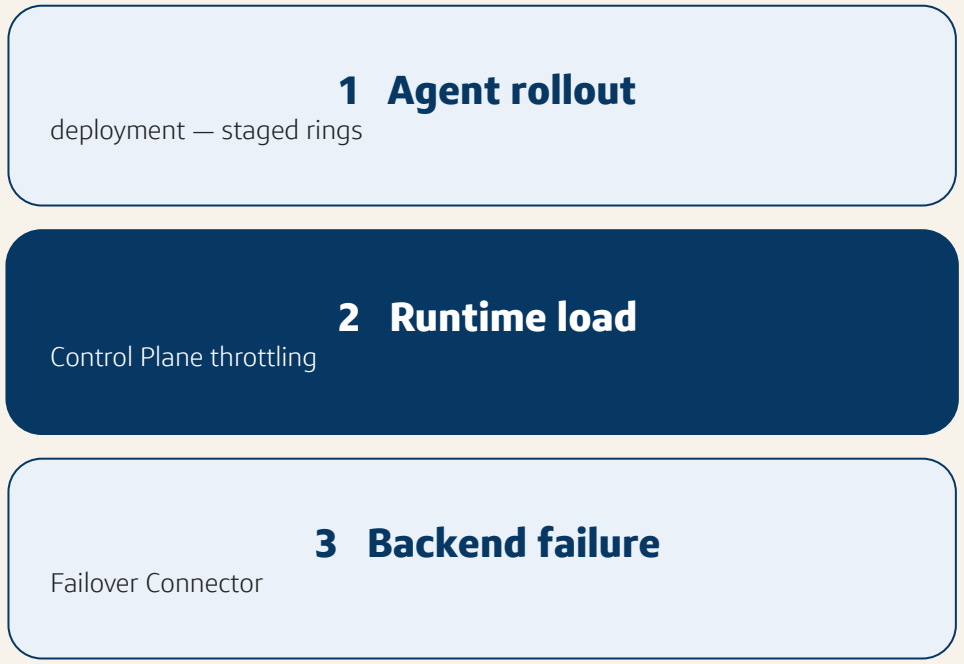
- **Baked into images —** A new agent version reaches an instance only when it rehydrates onto the new image — never an instant fleet-wide swap

- **Promoted in staged rings —** Non-prod → prod, segment by segment, so a regression is caught before it reaches everyone

- **Independent per runtime —** EC2, ECS, Fargate, and Lambda each promote separately — one runtime's bad build can't touch the others

Blast Radius — Layer 2: Runtime Load

The failure contained here: one noisy agent flooding the pipeline for everyone.



- **Per-agent throughput cap —** A default bandwidth limit means one misbehaving agent can't consume the shared pipeline's capacity
- **Centrally managed —** The cap is set and adjusted from the Control Plane, not per host — enforced consistently across the fleet
- **Protects shared gateways —** Gateways serving many applications stay healthy even when one application's telemetry suddenly spikes

Blast Radius — Layer 3: Backend Failure

The failure contained here: a backend outage dragging down the whole pipeline.

1 Agent rollout
deployment — staged rings

2 Runtime load
Control Plane throttling

3 Backend failure
Failover Connector

- **Failover Connector —** Detects a primary backend failure and reroutes telemetry to a secondary target automatically
- **Availability decoupled —** The gateway's effective availability no longer tracks any single backend's uptime
- **Backfill on recovery —** Historical telemetry is backfilled once the primary recovers — no permanent gap in the record

3

Data handling (PII/PAN/residency)

Redaction/scrubbing before data enters Monitoring platforms

Constraints on Data

Data Handling — Redaction, Defense in Depth

Redaction isn't one control, it's a layered defense, with the strongest guarantee (deletion at the source, on the host) backed up by a second check at the agent or gateway before data ever reaches a backend.

Producer responsibility

App teams are expected to keep sensitive data out of span attributes. The first and cheapest layer of defense

Collector Agent (at the source)

The OTel Collector Agent's Redaction Processor runs on the host itself, deleting attributes matching sensitive-data patterns before telemetry ever leaves the host

Gateway Collector

A second redaction pass runs at the regional Gateway, catching anything that slipped past the agent layer before it's exported to any backend

Redaction — Why Regex Alone Isn't Simple at This Scale

Constraint

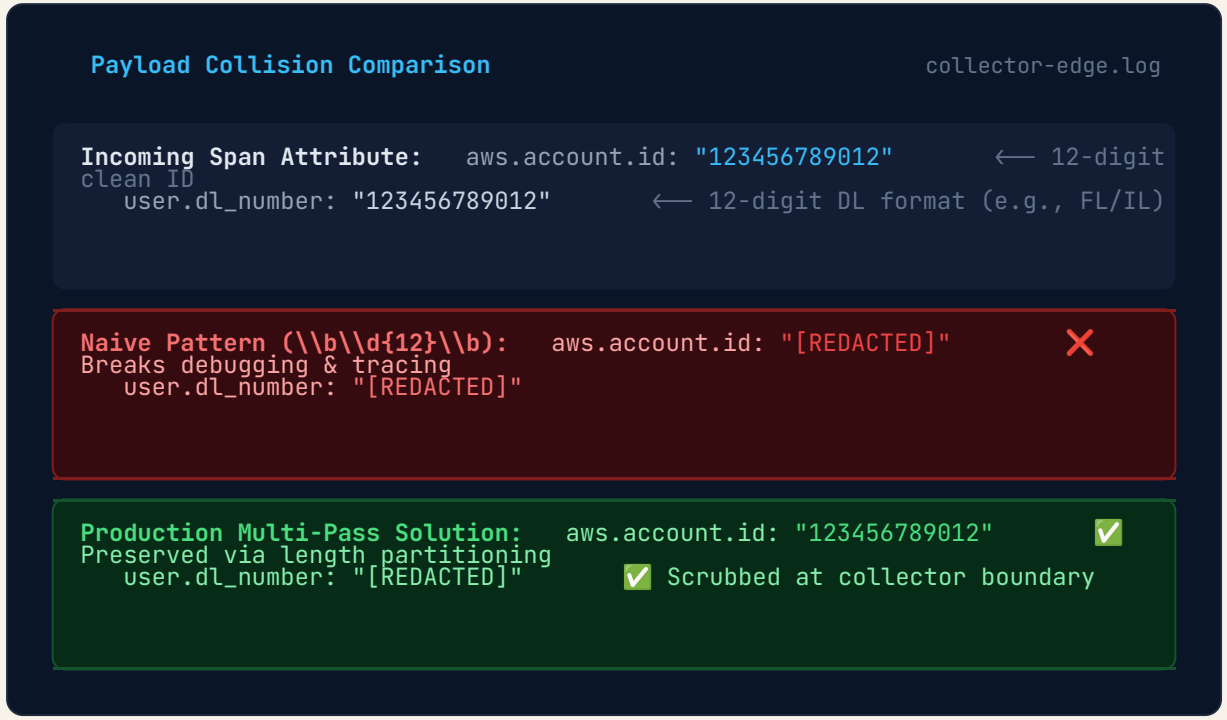
Golang's regexp lacks lookahead/lookbehind

Account IDs collide with driver's license formats

12-digit numbers overlap both formats, requiring per-state regex tuning to preserve account IDs while redacting driver's licenses.

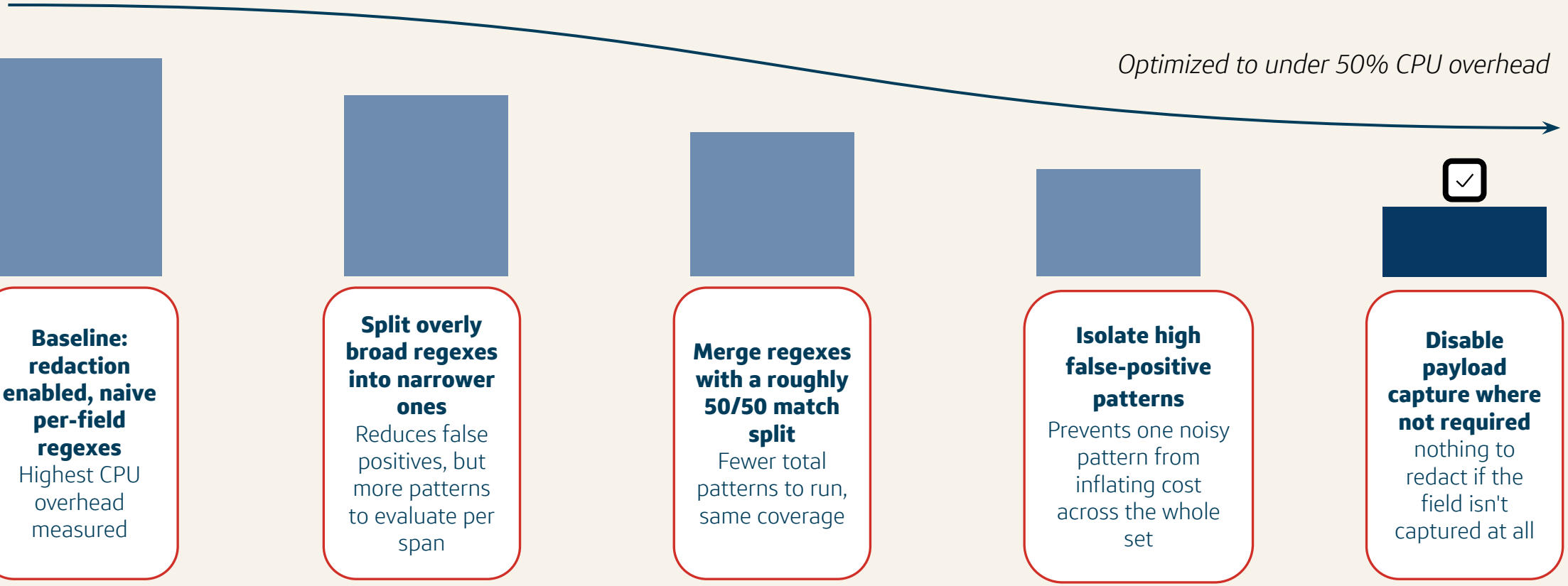
Scale Factor

15+ US states, each with a distinct driver's license format



Redaction — What It Actually Costs in CPU

The redaction processor's CPU overhead was brought down to a defensible, bounded increase, benchmarked under realistic production throughput on representative hardware.



4



Record Integrity & Retention

Telemetry is evidence, what gets kept is a deliberate, policy-driven choice



Constraints on Volume

The Cost Problem When Scale Outgrows the Backend

Spans disappear

Each backend account enforces a "Spans per minute" ingestion cap. Once exceeded, spans are dropped, no signal to the client

Dashboards show a false dip during peak traffic

Drops aren't random across time, they concentrate exactly when span volume peaks, so the busiest, most important moments are the most degraded

Traces appear fragmented

Dropping is per-span, not per-trace, one span survives, its siblings vanish, leaving a broken, misleading trace

Blind spots in service dependency graphs

Dropped spans break trace propagation and service topology, hiding critical upstream root causes during outages

Sampling, Phase 1 — Probabilistic Sampling at the Gateway

This is the open-source probabilisticsamplerprocessor from the OTel Collector Contrib repo, configured as a seamless, invisible layer in front of the backend's ingestion — the fastest, lowest-risk fix available.

Lightweight

Hashes the Trace ID, no buffering, no delay, no cross-span coordination needed

Complete

Every span in a trace shares the same Trace ID, so the whole trace is kept or dropped together, never fragmented

No client-side changes

Runs entirely in the Gateway Collector config, zero code change for application teams

Consistent and explainable

Same sampling percentage applied uniformly across gateways, simple to communicate: "we keep N% of traces, chosen at random, intact"

Sampling, Phase 2 — Targeted Sampling for the Traces That Matter

Rule	What it catches	Sample rate
HTTP 206	Partial content, often a sign of a slow, long-running job	100%
HTTP 4xx	Client errors, useful for debugging	100%
HTTP 5xx	Server errors, useful for triage during incidents	100%
Span status = ERROR	Catches failures in non-HTTP contexts, DB spans, internal spans	100%
Long-running (> 2s trace duration)	Traces that may not complete synchronously and need separate handling	100%
Everything else	Falls back to Phase 1's probabilistic sampling, to preserve a representative baseline	~10%

Rare, high-value traces get kept at 100% regardless of volume, while routine traffic still gets thinned — so the data that survives sampling is weighted toward what engineers actually need during an incident, not just what's statistically common.

Sampling, Phase 3 — Making the Ratio Adaptive, Not Fixed

The sampling ratio adapts per application, in real time, so a noisy, repetitive, low-value workload doesn't consume the same "sampling budget" as a workload that's actually generating useful traces. adjusting the ratio in real time based on what each application is actually sending.

Span Frequency Measurement

Gateway nodes measure each application's span generation rate in real time, continuously, not as a one-time capacity-planning estimate

Dynamic Ratio Suppression

Applications producing high-frequency, repetitive, low-value traces automatically get a lower probabilistic sampling ratio applied to them specifically

Adaptive Tail Evaluation

Traces that meet sampling conditions (errors, latency anomalies) are still preserved and exported at full fidelity, regardless of that application's current suppression ratio

Sampling — Why It Needs Its Own Three-Tier Architecture

Tier 1: Global Ingress

Global Balancing Gateway

Initial ingress point for every distributed span enterprise-wide.

Mechanism: Deterministic Trace-ID hashing routes spans to the correct regional cluster—eliminating cross-region egress overhead.

Tier 2: Regional Router

Regional Routing Gateway

Uses Contrib loadbalancingexporter for node-level dispatch.

Mechanism: A second consistent hash maps spans of Trace ID X to the identical tail-sampling node buffer.

Tier 3: Tail Evaluator

Tail Sampling Gateway

Buffers, evaluates, and samples traces completed in memory.

Policy: Evaluates rules and latency; exports 100% of errors/anomalies while throttling routine success volume.

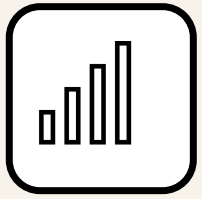
The core constraint: every span from the same trace must land on the same tail-sampling node — enforced by two layers of deterministic Trace-ID hashing (global, then regional), using the OTel Collector Contrib loadbalancingexporter.

5



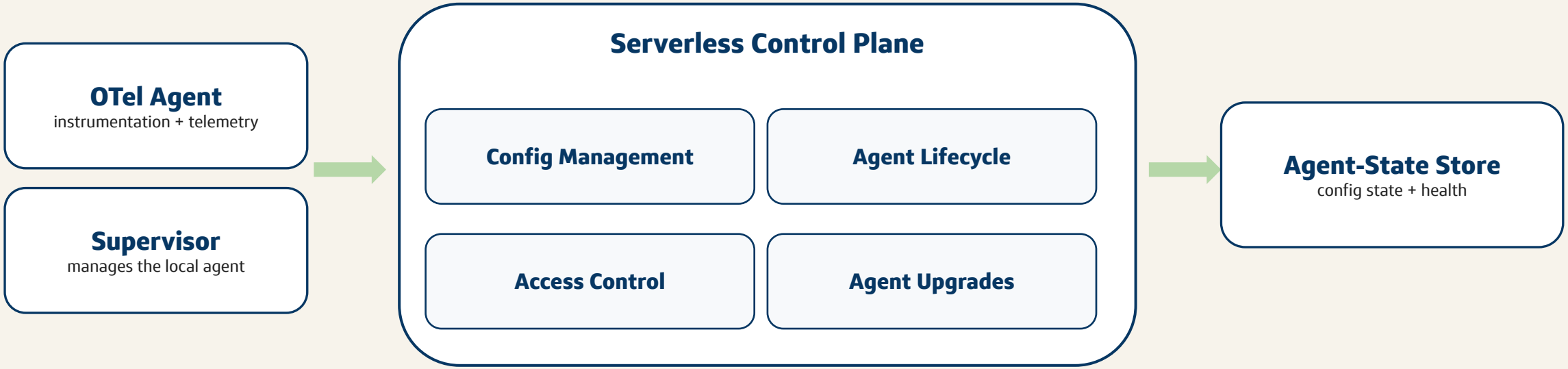
Audit trail and Change management

Collector/pipeline changes get versioned review and staged rollout, not just app code



Constraints on Operating the Fleet

Control Plane — Inside the Fleet Management Architecture



Built on OpAMP (Open Agent Management Protocol), the CNCF-aligned open standard for remotely managing OTel agents



Control Plane — Fleet-Wide Visibility

You can't govern what you can't see — the first thing the control plane delivers is a live, single view of every agent running in production.

- **Live health of every agent**
  Instantly know which agents are healthy, degraded, or silent, across every environment, from one place

- **Which config each agent is running**
  Confirm at a glance that every agent is on the approved, current configuration

- **Ownership-scoped views**
  Each team sees the agents they own, resolved automatically from identity

- **Alerting on unhealthy or silent agents**
  A degraded or disconnected agent raises an alert instead of quietly becoming a blind spot

Control Plane — Built-In Governance Capabilities

Tiered access control

Non-production access is scoped to each service's own owners; production access is restricted to admins only, reserved for defined, documented emergency scenarios

Org-level visibility

A breakdown dashboard, count of reporting agents per business unit

Self-healing config drift detection

The on-host agent periodically compares its running config against the centrally registered version and automatically reverts any local drift back to the enforced config

Throughput throttling

A default bandwidth cap, managed and adjustable centrally rather than per-host

Auto-discovery

New agents are automatically discovered and get their application-specific config pushed, without a manual onboarding step

Fleet management is a complete governance layer, not just a dashboard — access control, versioned config, self-healing, and auto-onboarding all work together so the platform stays correct without manual intervention.

Lessons for Teams Starting This Journey

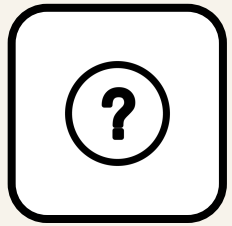
If another bank, insurer, or healthcare platform team is beginning this today, four lessons transfer directly

Absorb the Compliance weight
Bake redaction, lineage, and staged rollouts. The platform bears the regulatory burden so app developers just ship.

Sampling is Integrity, Not Just Cost
Multi-tier deterministic tail-sampling converts raw ingestion limits into an auditable retention policy keeping 100% of errors.

Open Standards Give True Leverage
Standardizing on CNCF OpAMP and OTel Collector provides vendor-independent fleet governance without commercial agent lock-in.

Upstream Instead of Forking
Close enterprise gaps by hardening the upstream ecosystem (Failover Connector) rather than maintaining fragile, private internal forks.



Questions?

Thank you