Observability Day
EUROPE

Scaling telemetry to terabytes

Applying Google's principles to OpenTelemetry with OpAMP

Raphael Menderico, Google

Mar 2026

Agenda

01 Time series observability @ Google

02 Scaling metric policies

03 Running collection at scale

04 Contributing to OpAMP

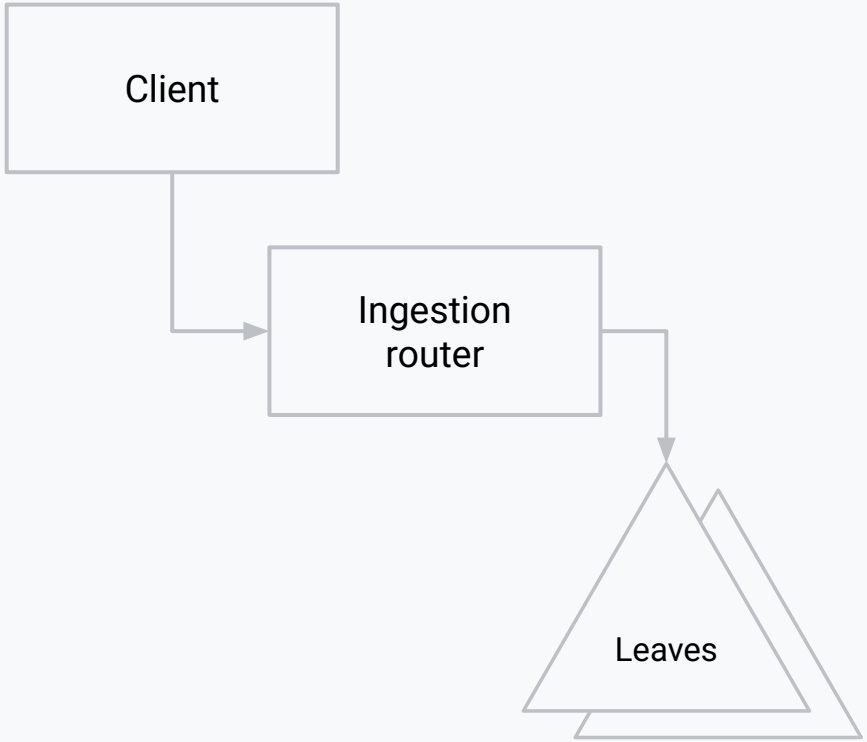
05 Conclusions

01

Time series observability @ Google

Monarch - Time series database

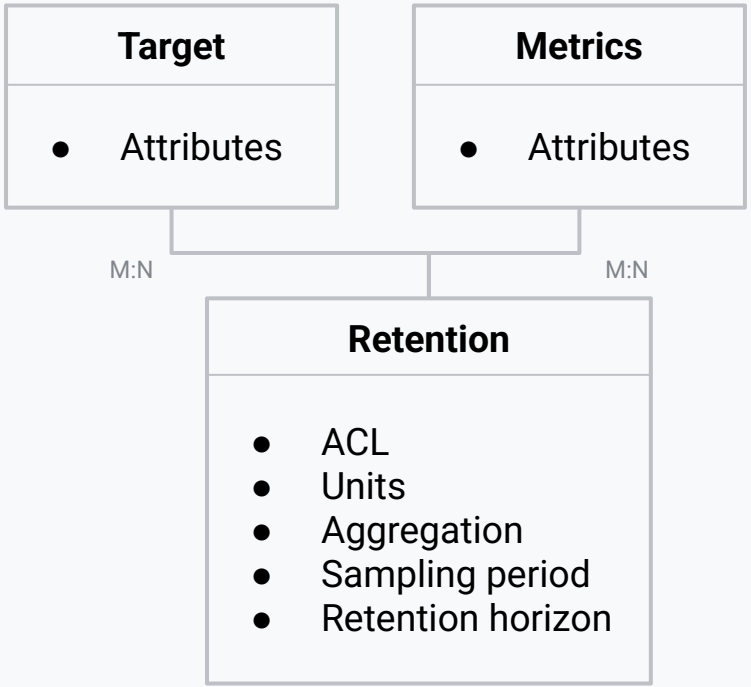
- Globally-distributed in-memory time series database system
- Multi-tenant metric service
- Regionalized architecture integrated by global query and configuration planes into a unified system.
- Numbers (from 2020 paper):
  - 100 Billion time series,
  - 200K individual tasks
  - 800 TiB in-memory stored,
  - 6M queries per second,
  - 2.5 TiB of time series data ingested per second.



Link: https://research.google/pubs/monarch-googles-planet-scale-in-memory-time-series-database/
Google search: Google Monarch Paper

Data model

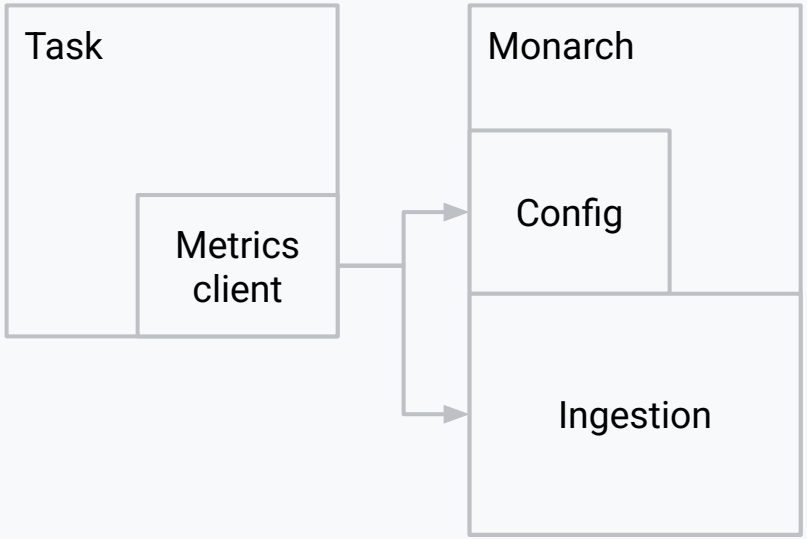
- Target schema
  - What is being monitored
  - Similar to OTel's Resource
- Metric descriptor
  - Similar to OTel's or Prometheus metrics
- Retention policy
  - A tuple of (Target schema, Metric descriptor, collection predicate)
  - Defines policies like sampling period, retention horizon, representation units, aggregation and ACLs
- Multiple retentions can be defined for the same pair of (Target schema, Metric descriptor)



Link: https://research.google/pubs/monarch-googles-planet-scale-in-memory-time-series-database/
Google search: Google Monarch Paper

Metrics client

- Monarch has its own client, similar to OTel's SDK
  - Collect events
  - Pushes data periodically to Monarch
  - Annotates data with exemplars when appropriate
- Original implementation was pull-based, similar to Prometheus
  - Hard to keep track of ALL tasks running in a scalable manner
- Moved to push based model where jobs push data according to the configured retentions



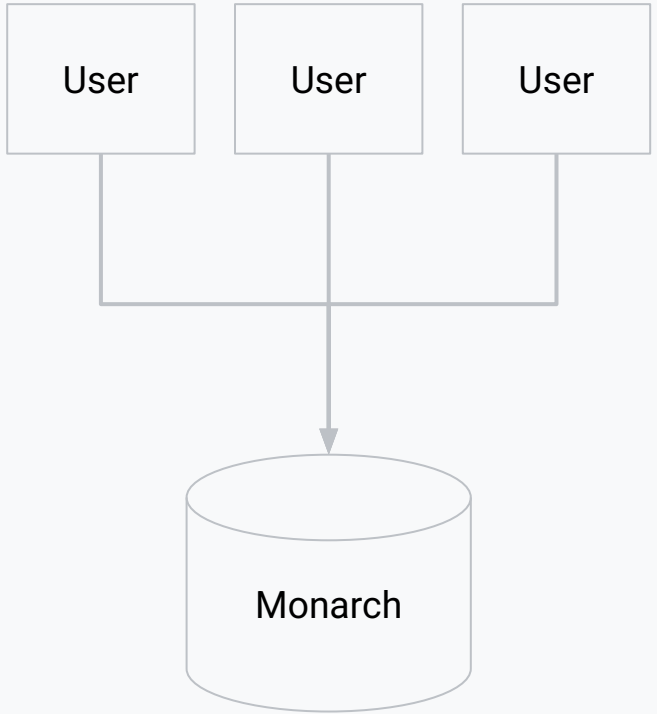
Link: https://research.google/pubs/monarch-googles-planet-scale-in-memory-time-series-database/
Google search: Google Monarch Paper

02

Scaling metric policies

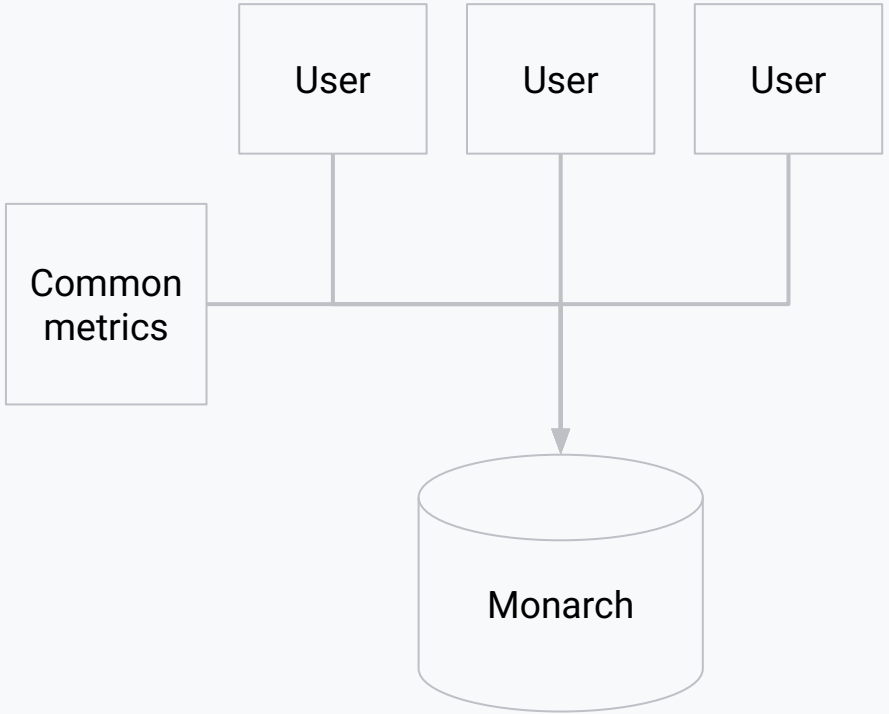
Delegating retention policies to developers

- **Initial policy:** developers are responsible for setting up a retention policy for their own targets
- **Rationale:**
  - A developer knows best what they need
  - Monarch Just provides a config interface
- **In practice:**
  - Lots of repeated configs (everyone needs RPC metrics!)
  - No OOTB monitoring
  - Inconsistent policies across teams
  - Opaque costs



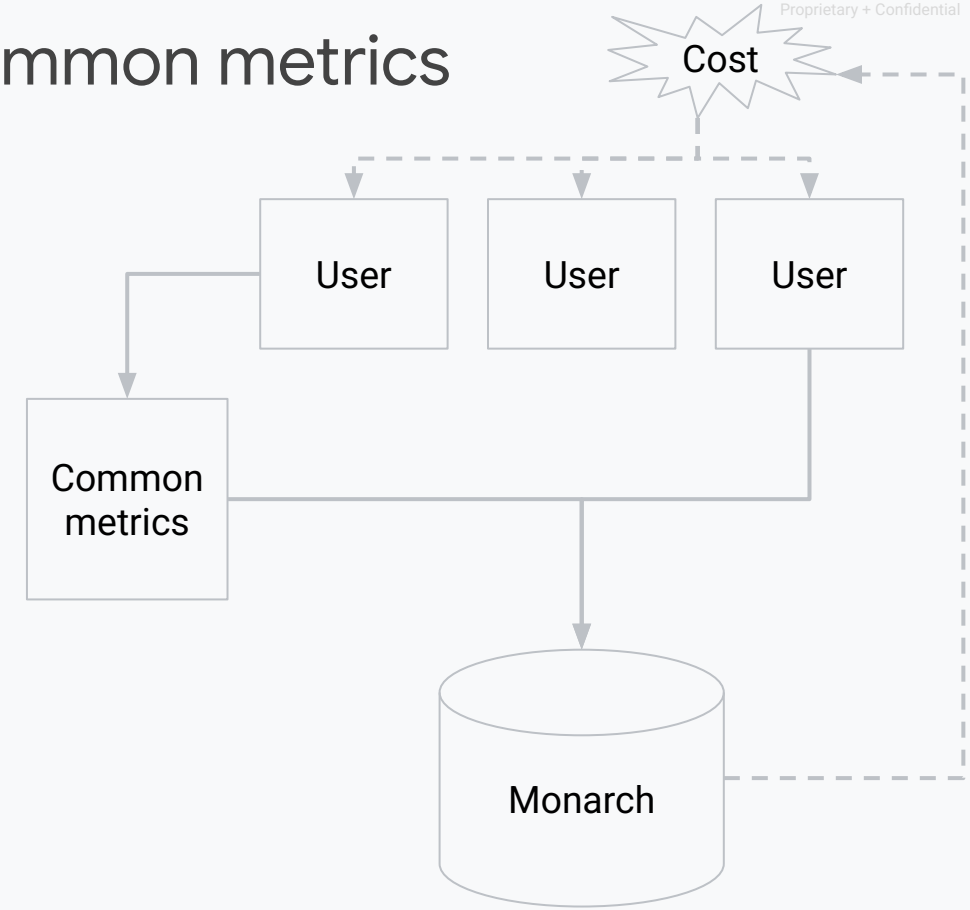
Centralized policies for common metrics

- **New policy:** Retain common metrics for everyone
  - **Rationale:** OOTB telemetry for all jobs!
  - Metrics are collected as soon as a job starts running, no registration required
  - Common set of metrics available for alerting and dashboards
    - Alerts and dashboards also set OOTB for teams
  - Common sampling period (1m) for most teams
    - Custom policies for teams with specific needs
- **Currently managing ~30% of all data stored in Monarch**



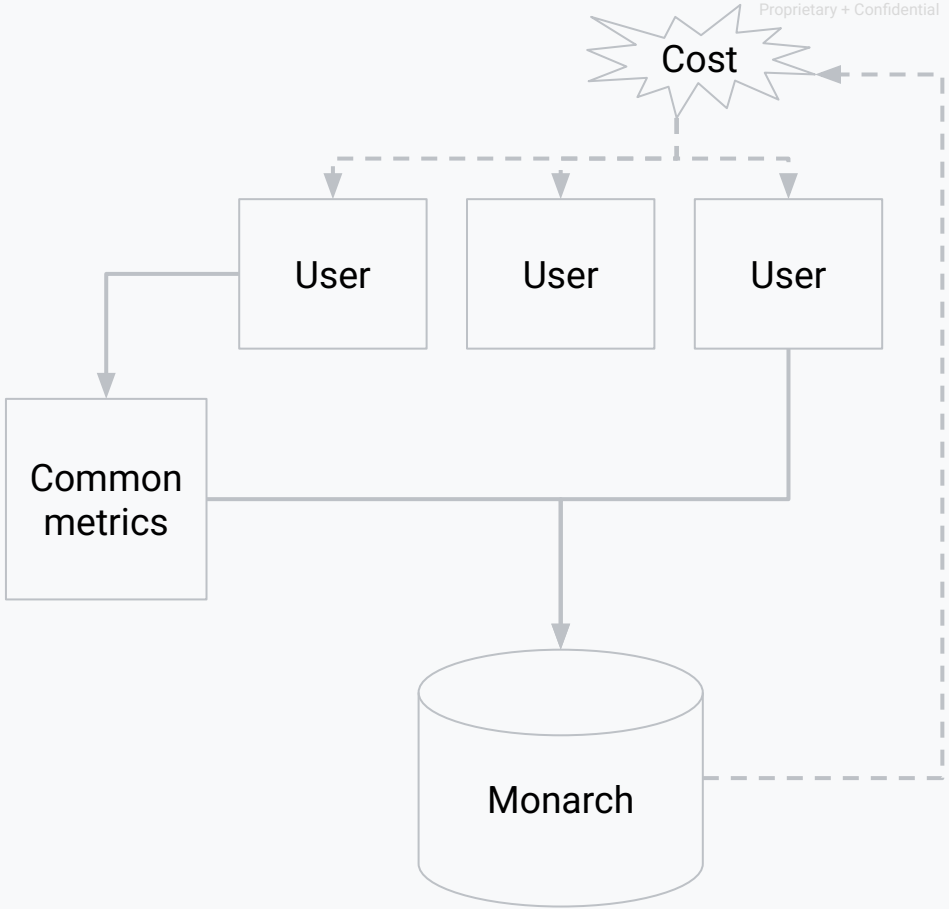
Centralized policies for common metrics

- **How to find which metrics are relevant?**
  - Rely on initial team knowledge
  - Turns out centralizing collection also needs to scale
- **Idea:** Ask the developers, they can suggest metrics
  - Metrics are then vetted and collected if pass checks and meet company policies
- **In practice:**
  - Costs are still opaque, a single team "pays" for the centralized metrics
    - Team charges these back to data producers
  - And now developers (especially library owners) want everything collected for everyone



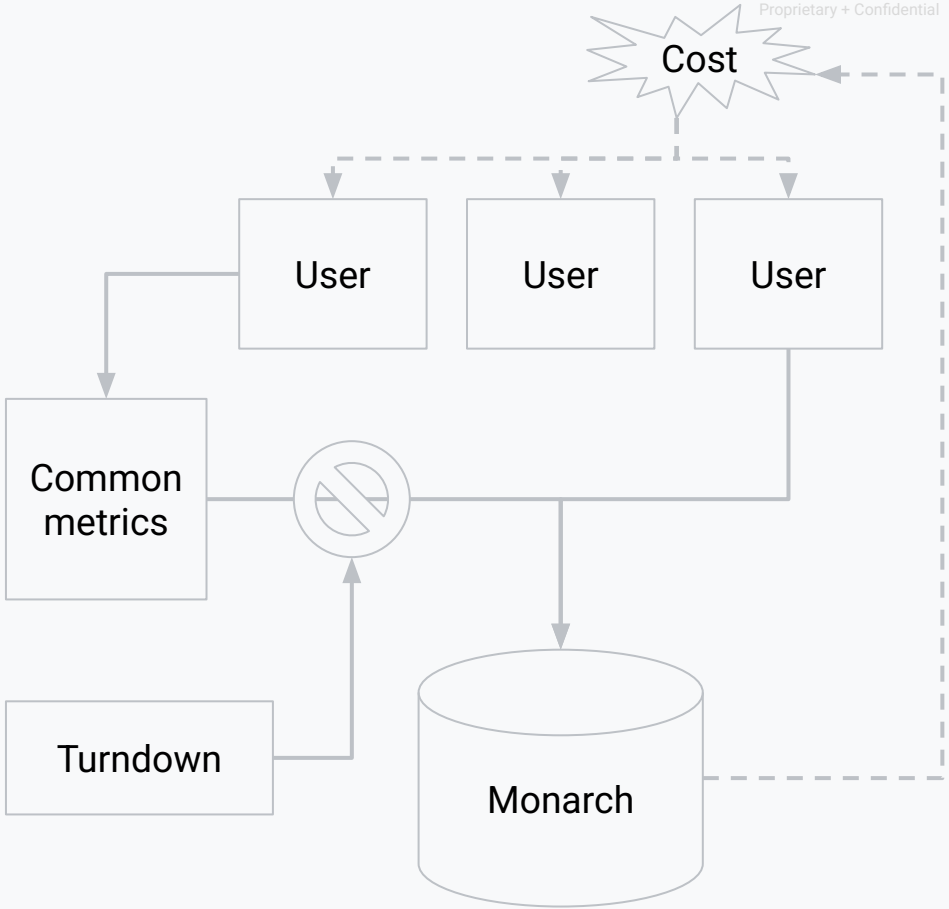
Metric accountability

- **Who owns the metric?**
  - Each metric retention has an owner
- **How much does it cost?**
  - For centralized metrics, Metric owner defines cost policies, these are charged back to teams
- **In reality:** developers might be willing to pay for the cost if it is small enough from their perspective, but these add up quickly across multiple teams!



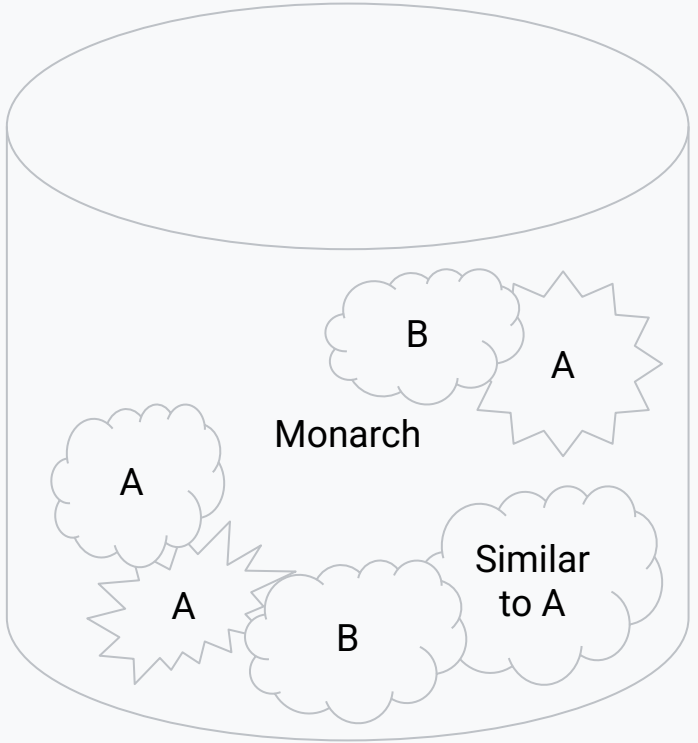
Metric turndown

- **Do we really need all that data? Prove it!**
  - Check metric usage (which alerts use it, does automation queries it, how often is it queried)
  - If unused, notify the metric owner and ask for justification
  - If remains unused, stop collecting it
- **Why metrics get unused?**
  - It was never useful in the first place
    - Policies are not sufficient to capture this
  - System evolution — what is useful today might not be useful tomorrow



Other sources of cost overhead

- **Metric duplication**
  - Some metrics collected centrally had 10000+ policies set
  - Monarch deduplicate metrics but some derived data is computed and stored repeatedly, leading to inefficiencies
  - **Lesson:** check for data duplication
- **Duplicate signal for the same event**
  - Example: RPC latency vs RPC framework latency
  - Example: different aggregations for the same signal
  - **Lesson:** Have a good, well documented list of metrics and aggregations everyone can access

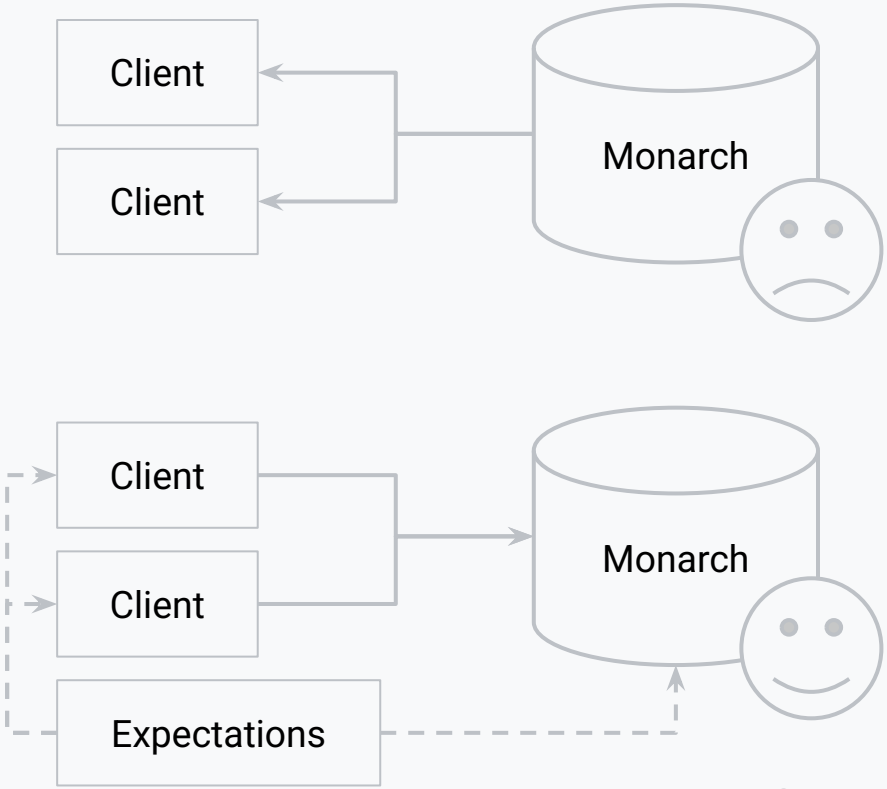


03

Running collection at scale

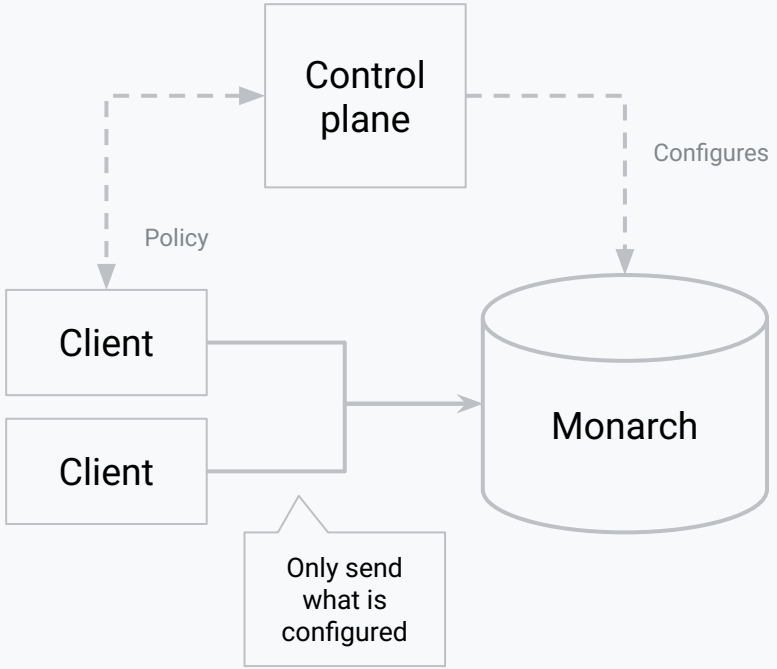
From pull to push

- **Pull collection**
  - Server keeps track of all clients
  - Hard to scale to large number of clients
- **Moving to push**
  - **Pros:**
    - Simper to run, no need to track clients, clients reach out to you
    - Protects service from hotspots and startup spikes.
  - **Cons:**
    - Cannot distinguish between missing and non-existing clients
      - **Solution:** Separate system tracks expectations, so telemetry can distinguish between missing and non-existing clients.



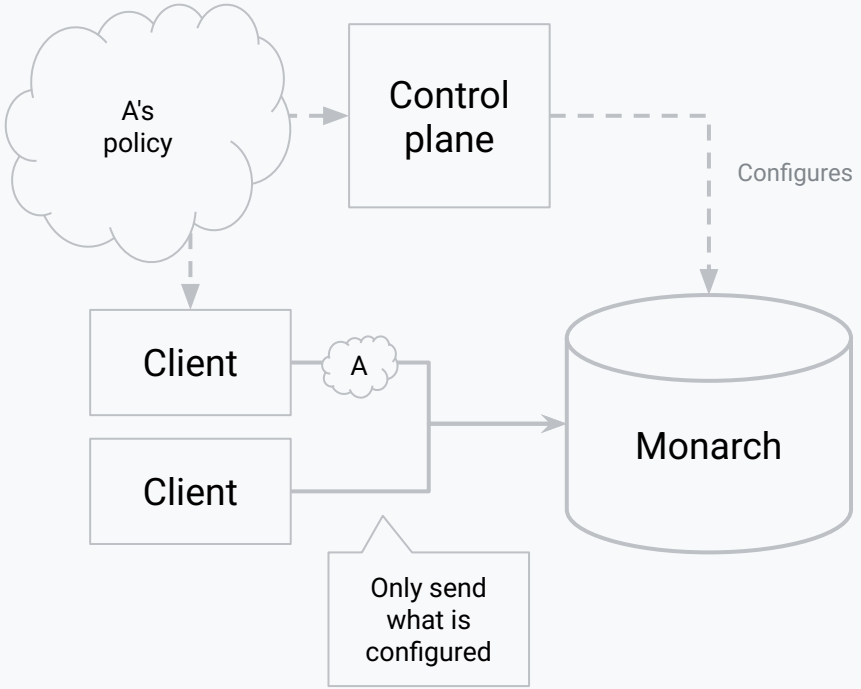
Only collect what you need

- **Telemetry costs money!**
  - Default behavior is to send all data to a server and let it decide what to do
  - Incur **network and storage costs**, even if some of these are temporary
- **The cheapest telemetry is the one you don't send**
  - Applies to **both push and pull** collections
- **Solution: check collection policy before transmitting the data**
  - With right indexing, checking policy is cheaper than collecting data
  - Policy can be reused for multiple writes



Only collect what you need - lessons learned

- **Expensive, complex policies**
  - Sometimes checking the policy ends up being more expensive than just sending the data
  - Need to account for cost of keeping policies in the client
- **Solutions:**
  - Keep clients lean, simplify policies before sending to the client
  - Policies mutate rarely, no need to retrieve them too often
  - Hash policies to reduce transmission cost when policies don't change
  - Provide information in the policy that helps the server to route data to the right place cheaply



04

Contributing to OpAMP

Bringing policies to OpAMP

- **October 2025: OTEP 4672**
  - Propose a control plane similar to the one used in Google
  - OpAMP committee Feedback:
    - Separate policy spec from actual policy
    - Allow policy configs for traces and logging
- **November 2025: Kubecon NA**
  - Agree on address the feedback
  - Policy Prototypes
  - Refined, generic policy proposed (OTEP 4738)
  - Tero's experimental results

05

Conclusions

Main lessons learned

- Centralized policies reduce overall costs
  - Single team has global visibility over telemetry data
  - Common policies helps rationalize telemetry usage
    - Data driven based on logging patterns of dashboard / querying usage
  - Usage tracking removes unused or unnecessary data
- Remote client configs increase cost savings
  - Push model leads to leaner servers, cheaper collection
  - Data that is never expected to be stored is not transmitted to servers, saving network and storage costs

Thanks

(in no particular order)

Google

- Aleksander Ciepiela
- Allie Mazzia
- Pereira Braga
- Ralf Wildenhues

Open Telemetry

- Josh Suereth (Google)
- Jacob Aronoff (Tero)

Links

Monarch paper

OTEP

Experimental results