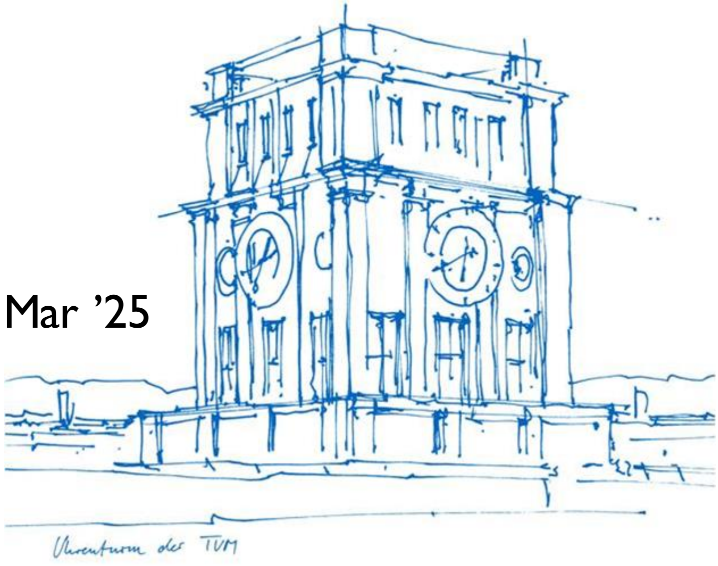
Metrics, Mayhem, and Microservices: Taming the Cloud Observability Beast

Alessandro Cornacchia, Theophilus A. Benson, M. Bilal, **Marco Canini**

KAUST | Carnegie Mellon University | Unbabel

Academic Salon on High-Performance Ethernet: Host Networking and Monitoring (TUM) | Mar '25

sands.kaust.edu.sa | marco@kaust.edu.sa



GitHub Copilot

stability.ai

AI

Data Analytics

Scientific Computing

Entertainment, VoD, Social Networks

Enterprise Services, CRM

Collaborative tools

Storage

Etc.

Source: Flaticon

amazon web services

Alibaba Cloud

Microsoft Azure

DEEM ديم

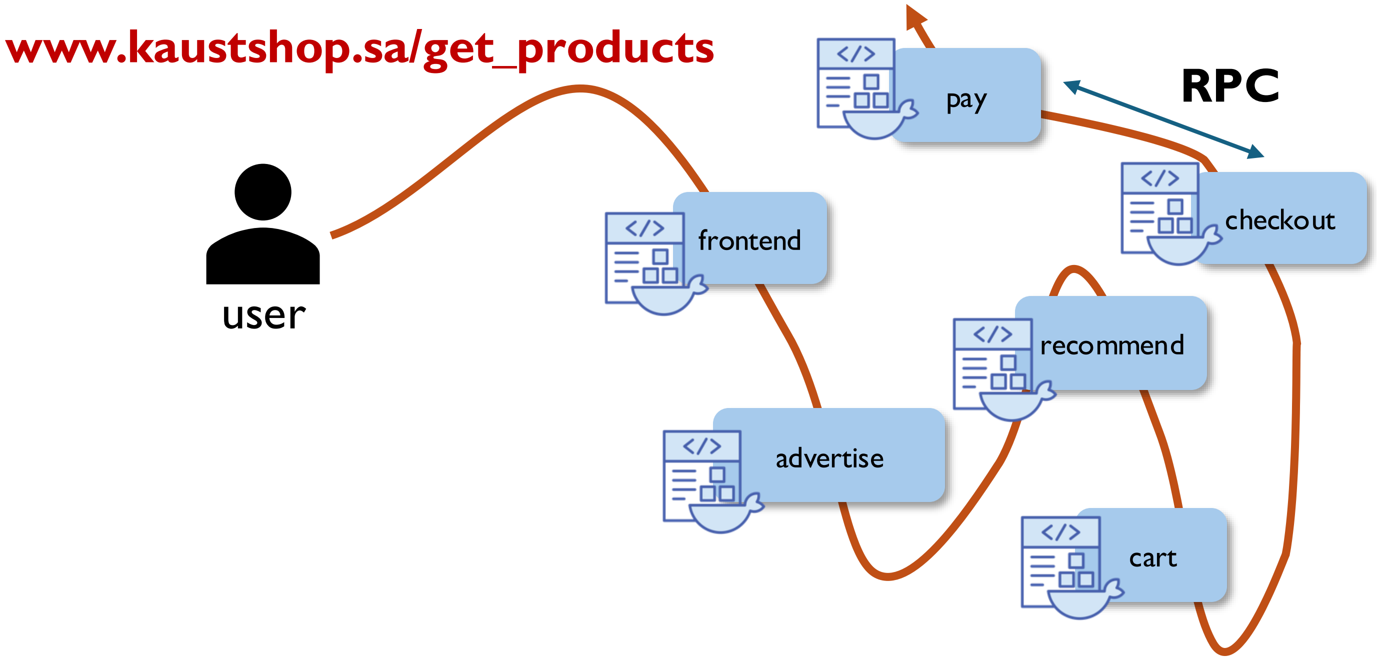
Google Cloud

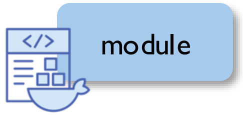
2025 Public cloud computing market size estimated at $723 billion | Statista

Cloud-native applications

Decomposition into independent binaries: microservices

- typically materialized as Linux containers
- interacting through network communication: RPC APIs





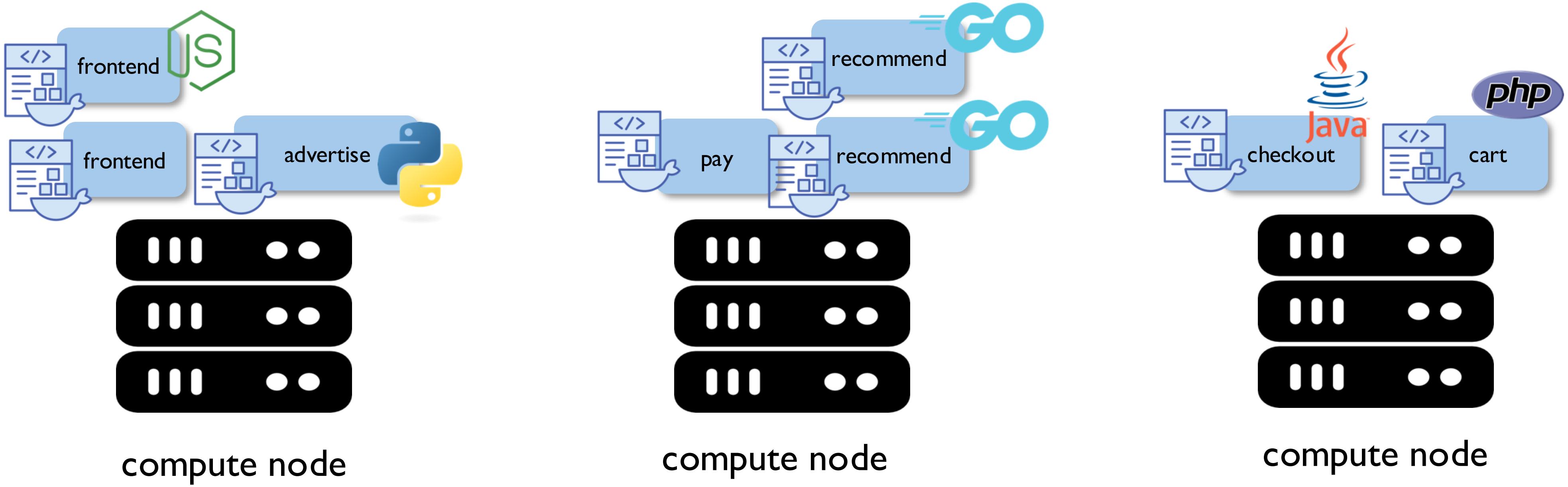
= microservice in e.g., Docker isolated runtime

Cloud-native applications: The GOOD

Decomposition into independent binaries: microservices

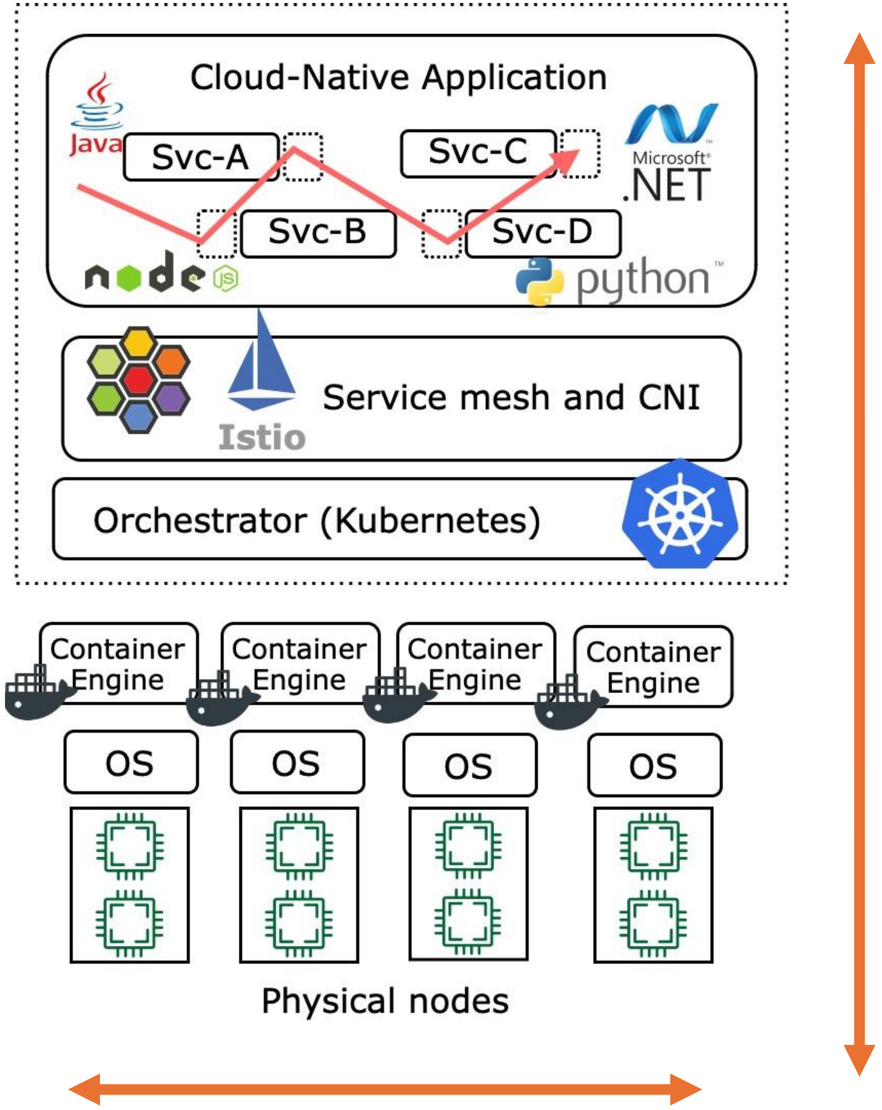
— typically materialized as Linux containers

— interacting through network communication: RPC APIs



- Scaling elasticity + development agility

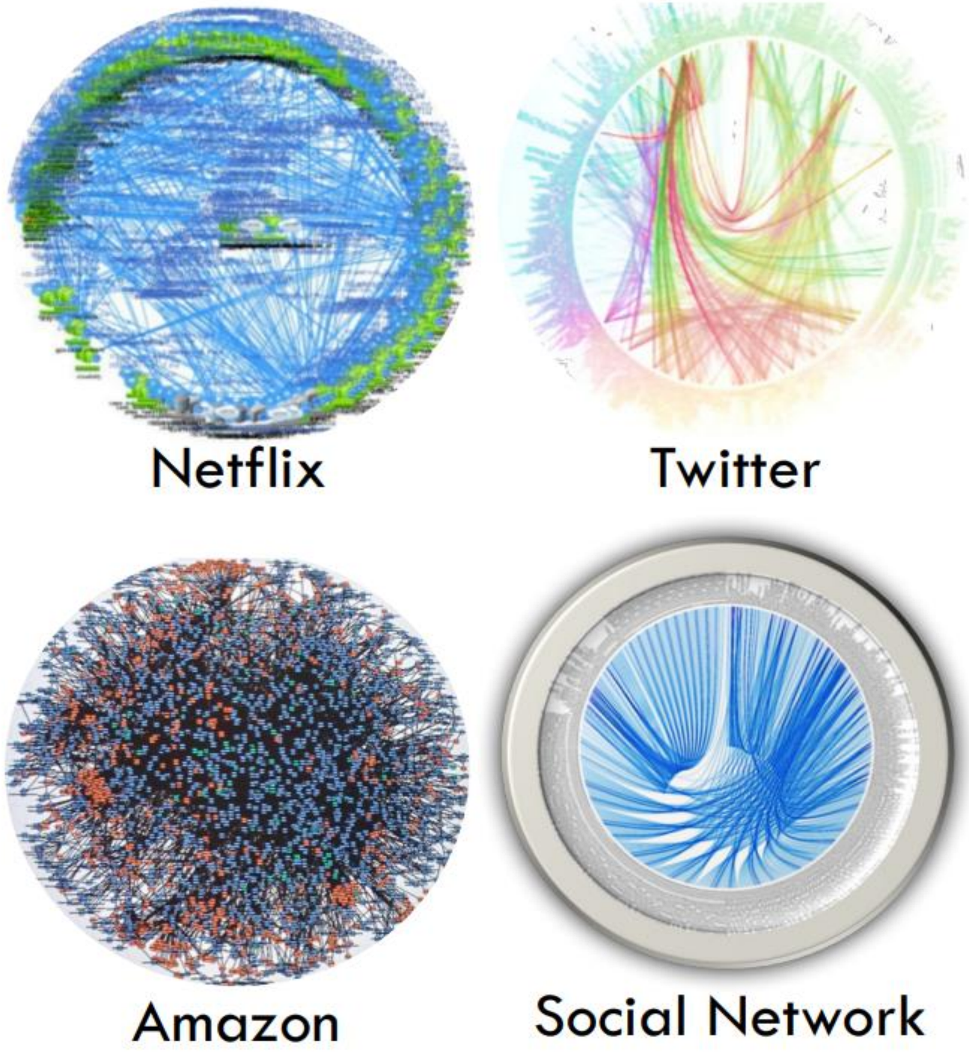
Cloud-native applications: The BAD



Failure surface increases!

- Complex stack of software abstractions
  - (gray) failures
- It's a **networked** system
  - network slowdowns directly translate on application performance drops

Cloud-native applications: The UGLY



Why is my application misbehaving ?

[ASPLOS 2019] *Gan Yu et Al.,* Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices.

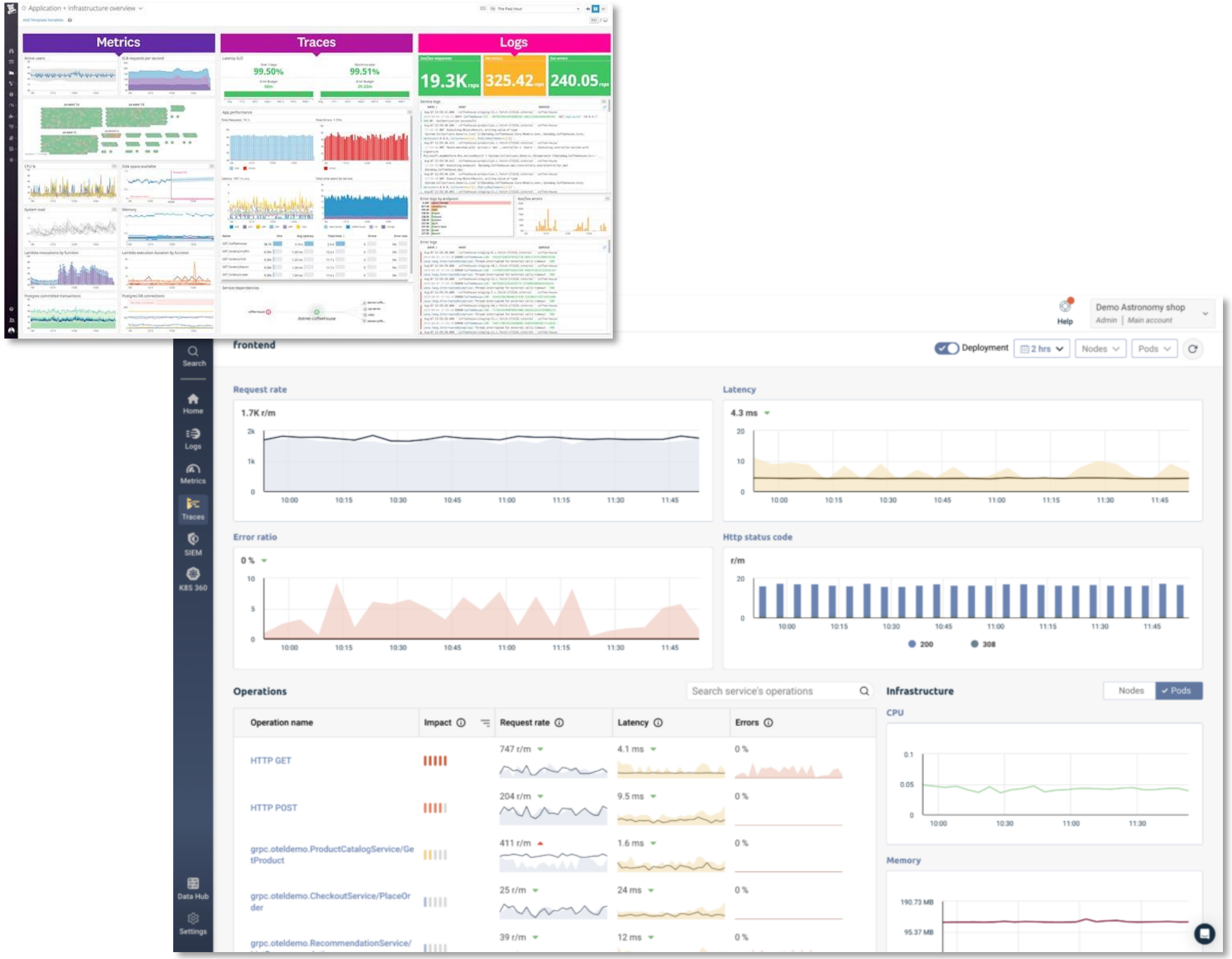
Observability

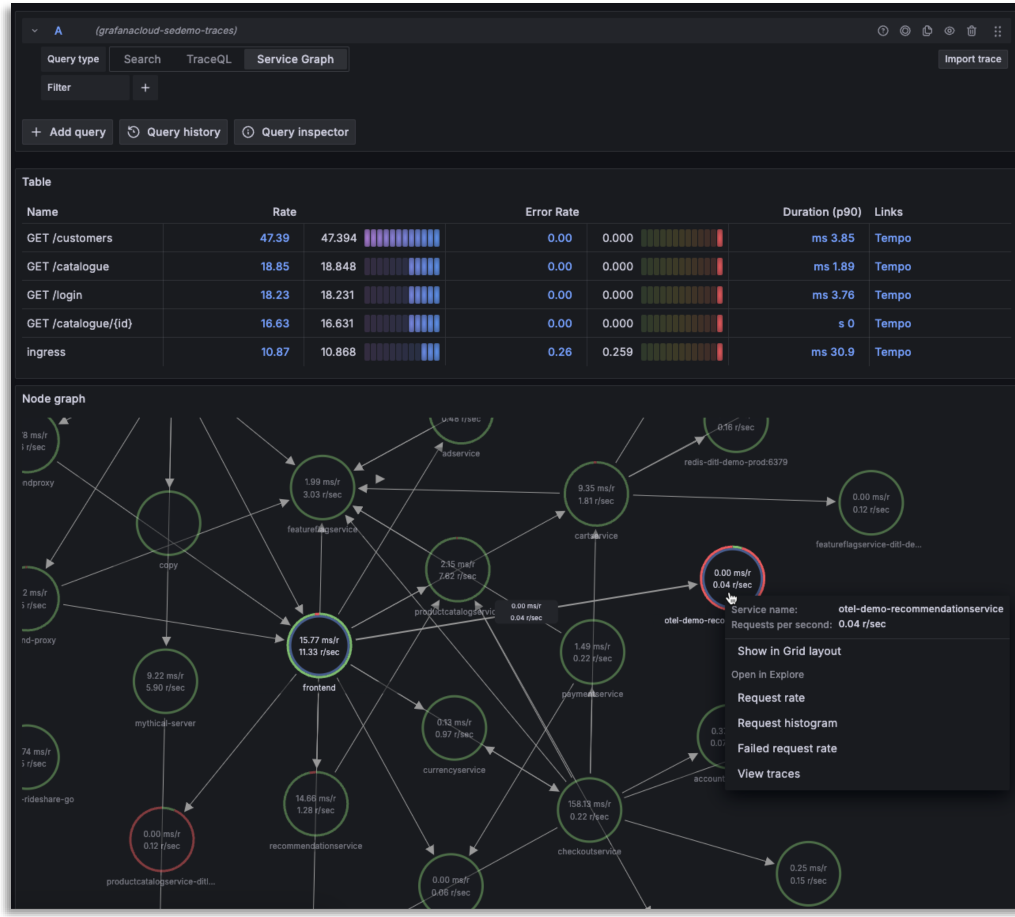
“Is a system property that defines the degree to which the system can generate actionable insights.

It allows users to understand a system’s state from external outputs and take (corrective) action.”

[Cloud Native Computing Fundation (CNCF): https://glossary.cncf.io/observability/]

→ Requires monitoring a wealth of data

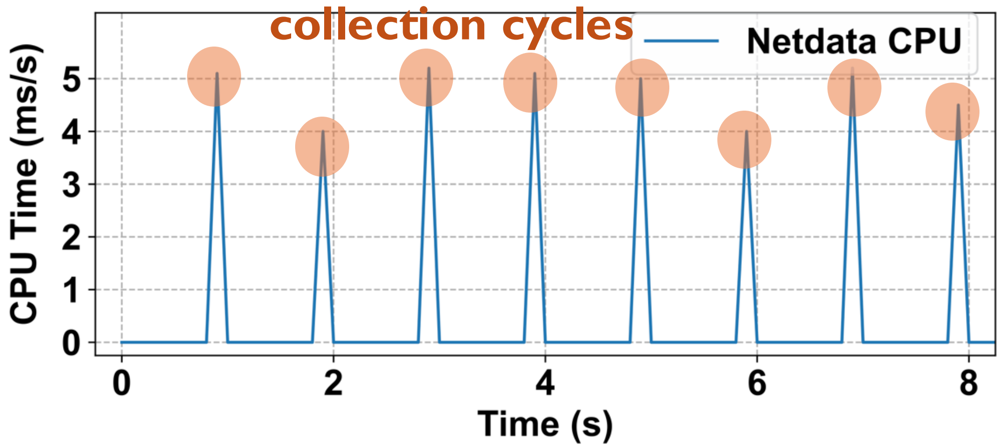


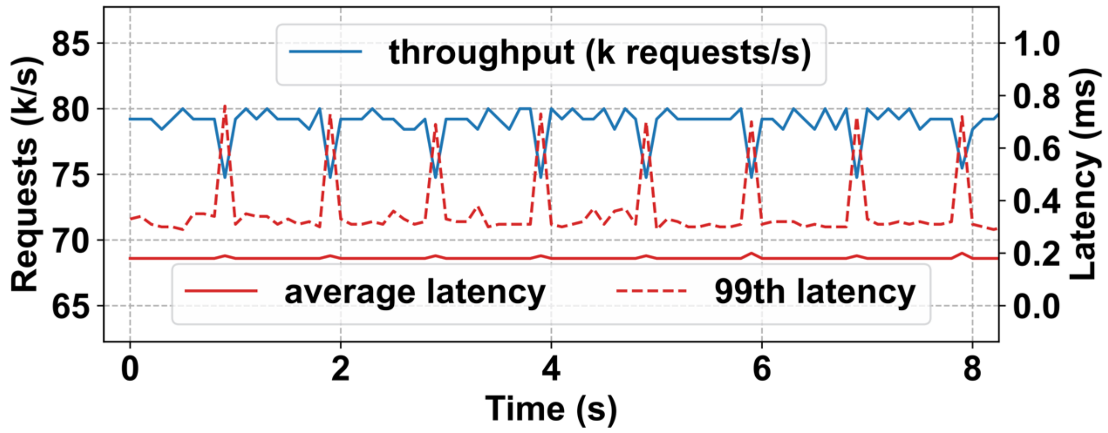


Challenging!

Monitoring affects application performance

- Today's observability: massive centralized data collection
  - e.g., Netdata, Prometheus





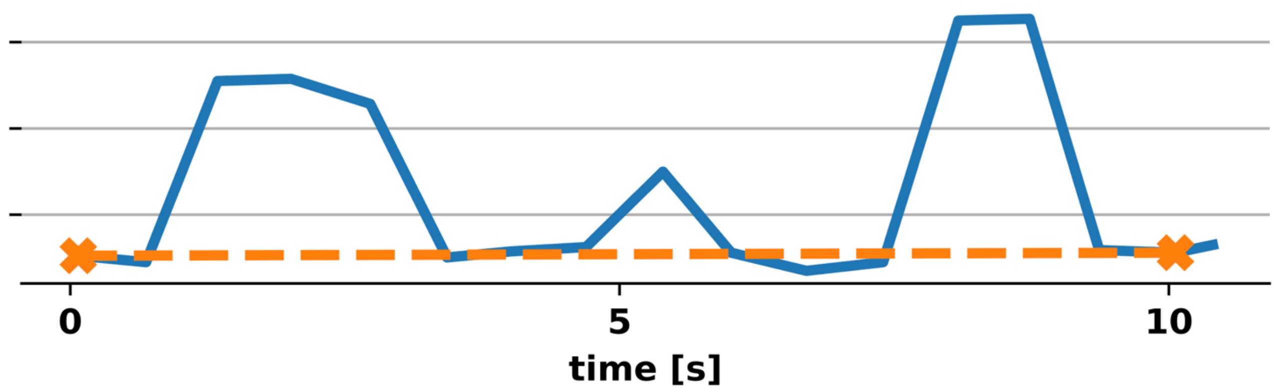
- at scale, observability interferes with application performance
  - monitoring processes and applications share the same resources
  - non profitable CPU cycles occupied for monitoring
  - can hurt end user performance

Don't collect as often?

[ATC 2022] *Zhe Wang et Al.,* Zero Overhead Monitoring for Cloud-native Applications.

Overheads prevent fine-grained analysis

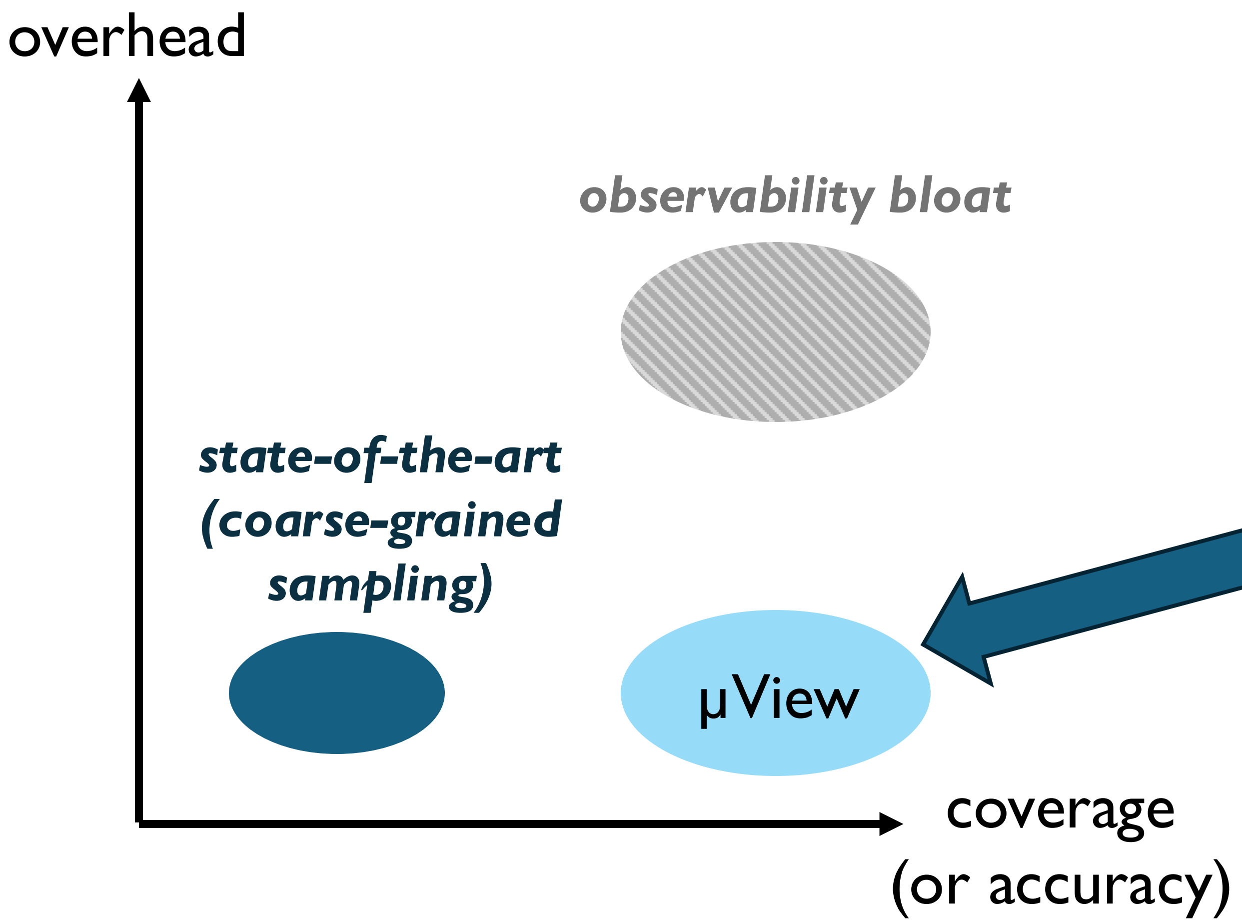
- Microservices update their metrics following RPC requests arrivals
  - subsecond-scale variations
- but we sample at coarse timescale
  - 10s-1m recommended for Prometheus



  - hard to reason about cause-effects
  - cannot precisely correlate SLO violations with system runtime state

	Source	Metric	Timescale
App	KV store [78]	db_keys keyspace_hits commands_duration evicted_key	RPC
	NoSQL DB [69]	mongodb_connections mongodb_op_counters	
Proxies	Envoy [35]	upstream_rq_active upstream_cx_connect_fail cx_rx_bytes_received cx_tx_bytes_sent	RPC
OS	cAdvisor [40]	cpu_usage_seconds network_tcp_usage fs_writes llc_occupancy processes oom_events	tunable

Table 1: Representative examples of metrics generated at different layers of the microservice's stack. Timescale is the frequency at which a metric is generated and/or updated by the source.



overhead
observability bloat
state-of-the-art
(coarse-grained
sampling)
μView
We want to
be here!
coverage
(or accuracy)

Introducing µView

- We build upon three key insights:

1) *Generation* of observability data is cheap, overhead is in *ingestion*

2) Value of *local* data

- can *detect* anomalous microservice states and performance issues locally (e.g., queue sizes, memory bottlenecks, etc.)

3) Rise of IPU accelerators → offload opportunity

- process richer fine-granularity data without imposing CPU overheads on nodes

Introducing μView

- We build upon three key insights:

1) *Generation* of observability data is cheap, overhead is in *ingestion*

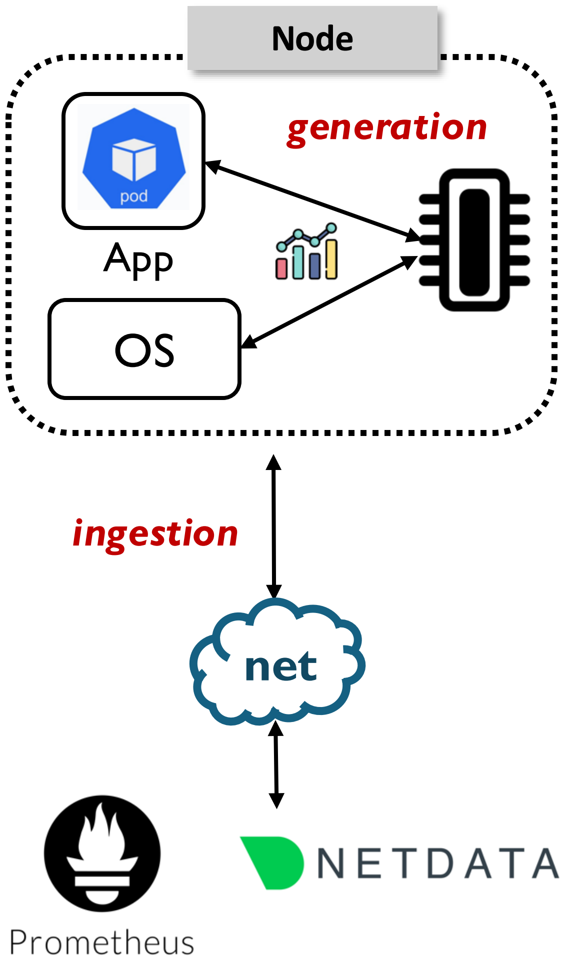
2) Value of *local* data

- can *detect* anomalous microservice states and performance issues locally (e.g., queue sizes, memory bottlenecks, etc.)

3) Rise of IPU accelerators → offload opportunity

- process richer fine-granularity data without imposing CPU overheads on nodes

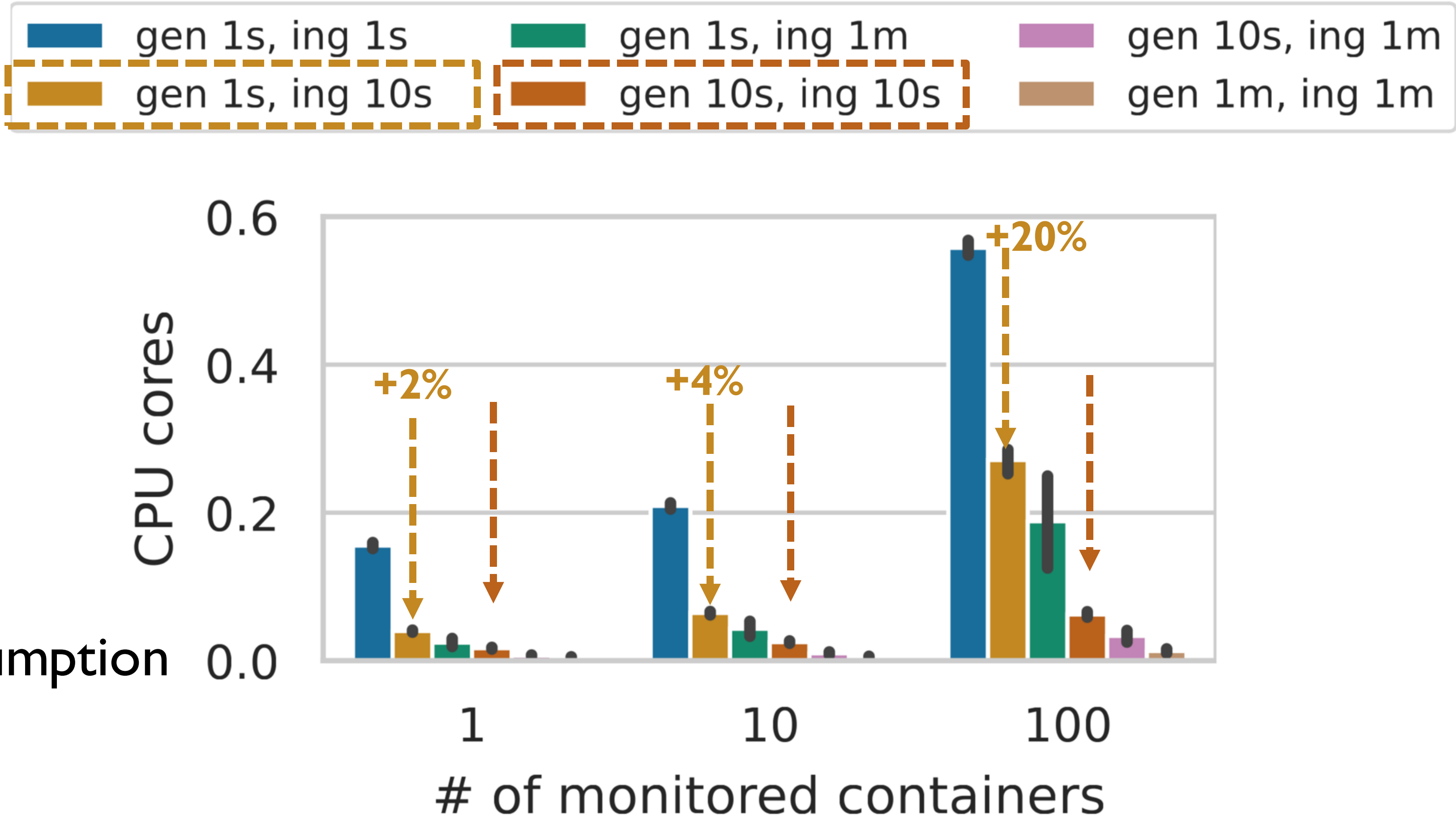
Dissecting the CPU overheads



- metrics **generation & update**
  - memory writes
- metrics **ingestion**:
  - memory copies
  - serialization & network communication

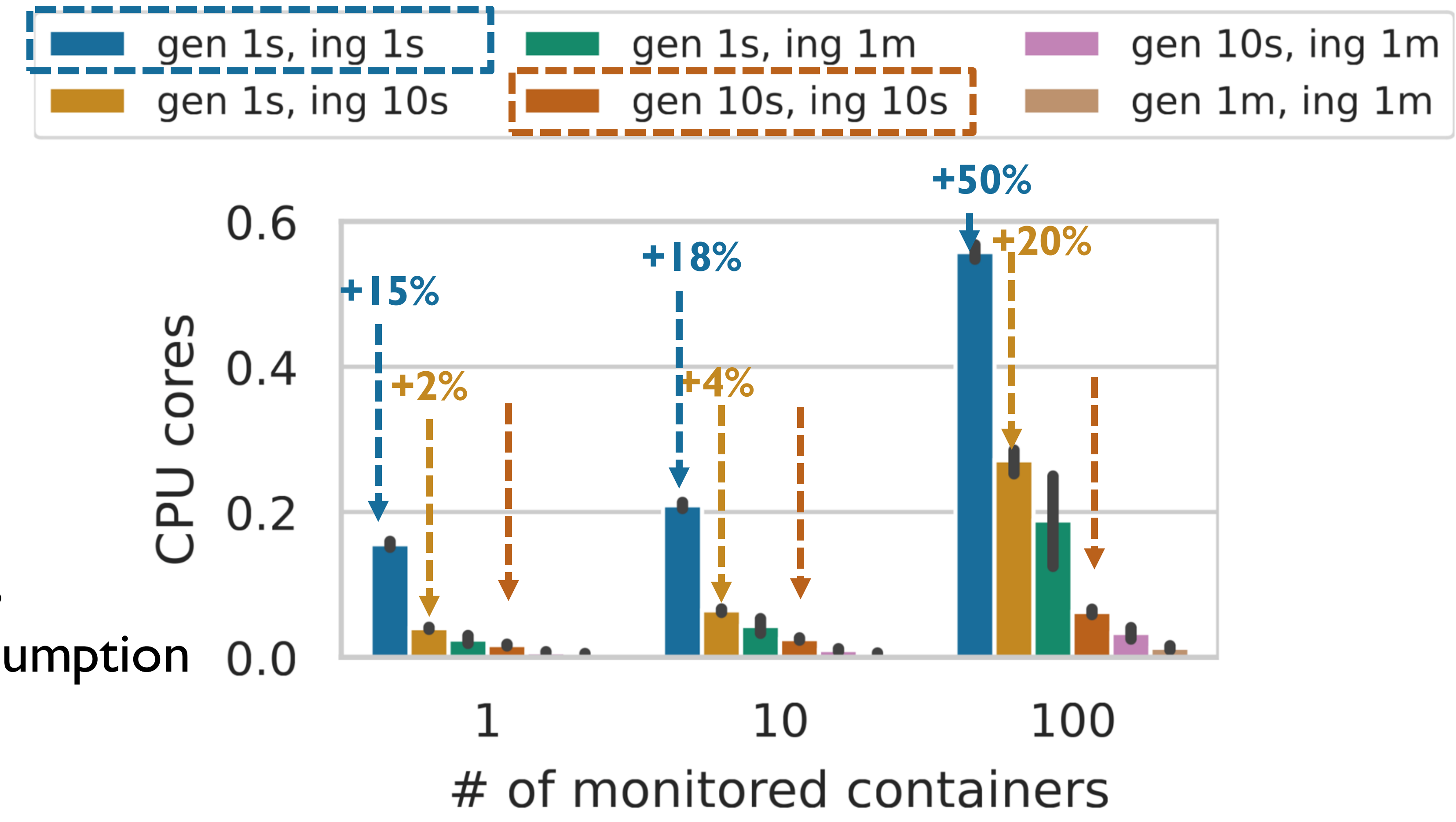
generation $\overset{?}{>=<}$ **ingestion**

- we run a microbenchmark:
  - 1 node for Docker containers +
    1 node with Prometheus collector
  - cAdvisor generating 100+ system metrics for all containers
  - **we vary gen and ing intervals**



cAdvisor’s
CPU consumption

cAdvisor's
CPU consumption



fine-grained metrics incur low generation overhead…

but must use coarse-grained metrics because of ingestion overhead!

This suggests using a wealth of data locally

1) *Generation* of observability data is cheap, overhead is in *ingestion*

2) Value of *local* data

- can *detect* anomalous microservice states and performance issues locally (e.g., queue sizes, memory bottlenecks, etc.)

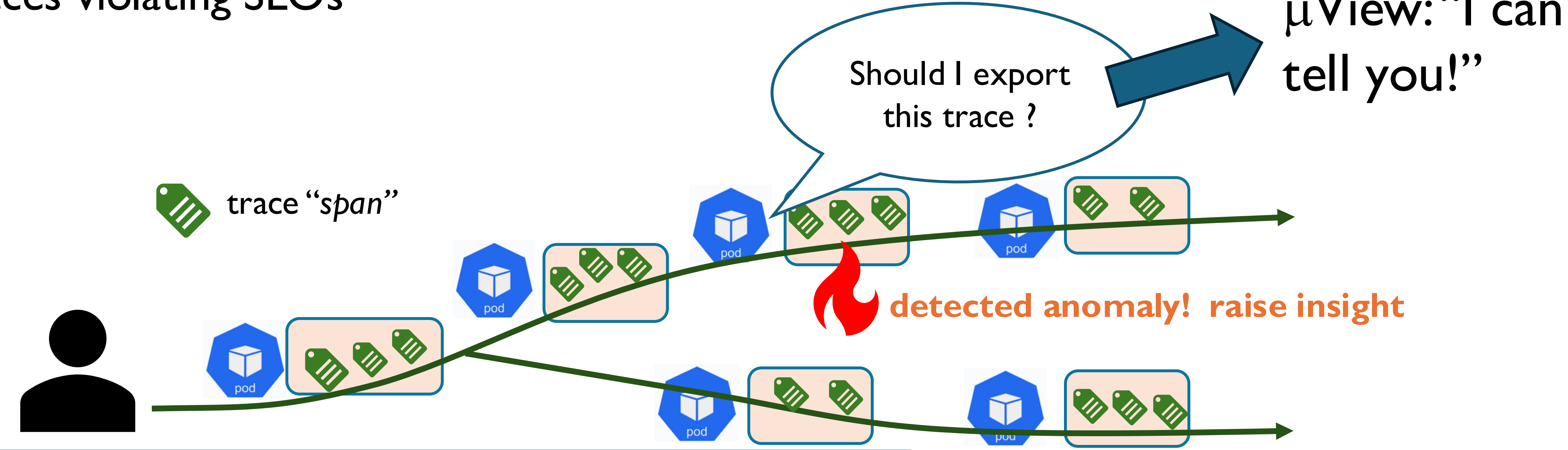
3) Rise of IPU accelerators → offload opportunity

- process richer fine-granularity data without imposing CPU overheads on nodes

Local data: better distributed tracing

- tracing
  - **which services** and **what latencies** during execution graph of user requests

- cannot collect all traces → how to maximize "relevant" traces?
  - traces violating SLOs



Local insights useful but processing needs increase

1) *Generation* of observability data is cheap, overhead is in *ingestion*

2) Value of *local* data

- can *detect* anomalous microservice states and performance issues locally (e.g., queue sizes, memory bottlenecks, etc.)

3) Rise of IPU accelerators → offload opportunity

- process richer fine-granularity data without imposing CPU overheads on nodes

Infrastructure Processing Units (IPUs)

- Programmable network cards (SmartNICs)
  - on-path cores → programmable Data Path Accelerator
  - off-path cores → SoC with general-purpose cores and OS (Linux)

SmartNIC
e.g., NVIDIA BlueField-3
400Gb/s Ethernet
16 ARM A78 off-path CPU cores
16 cores, 256 threads Data Path Accelerator (DPA)

HPC/AI
Networking
Security
Storage

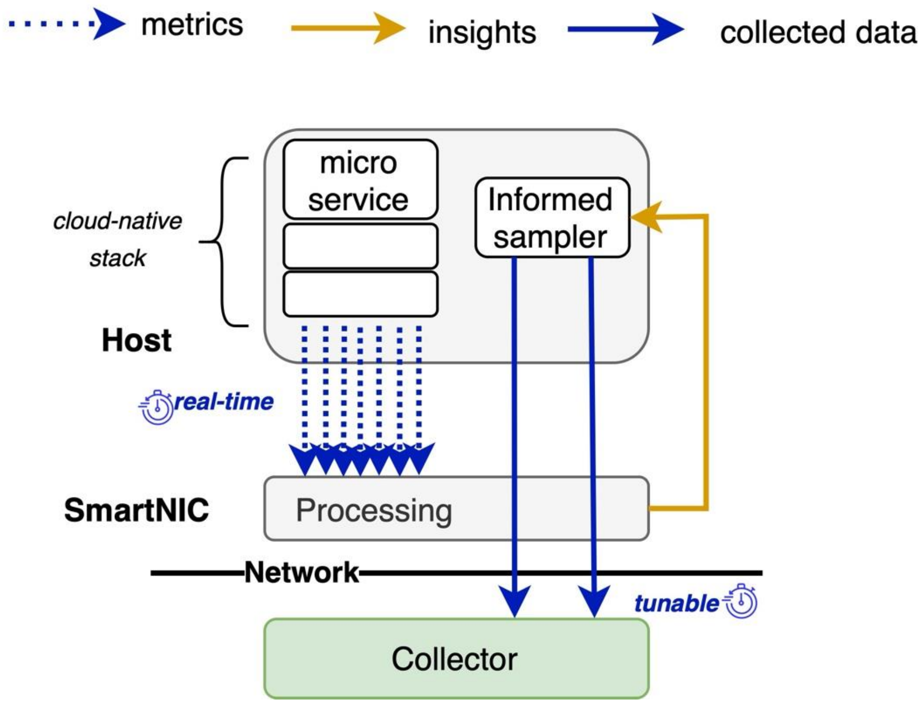
Local processing without CPU overheads on nodes

μView: in-situ observability

- μView *continuously* **locally** monitors metrics at **high temporal** resolution

- μView automatically pinpoints anomalies and triggers actionable insights

- by leveraging μView's insights, observability libraries can improve *sampling quality*
    - capture informative data
    - reduce clutter



Design challenges

Host to SmartNIC data movement

– take data outside the host boundaries, without introducing overhead

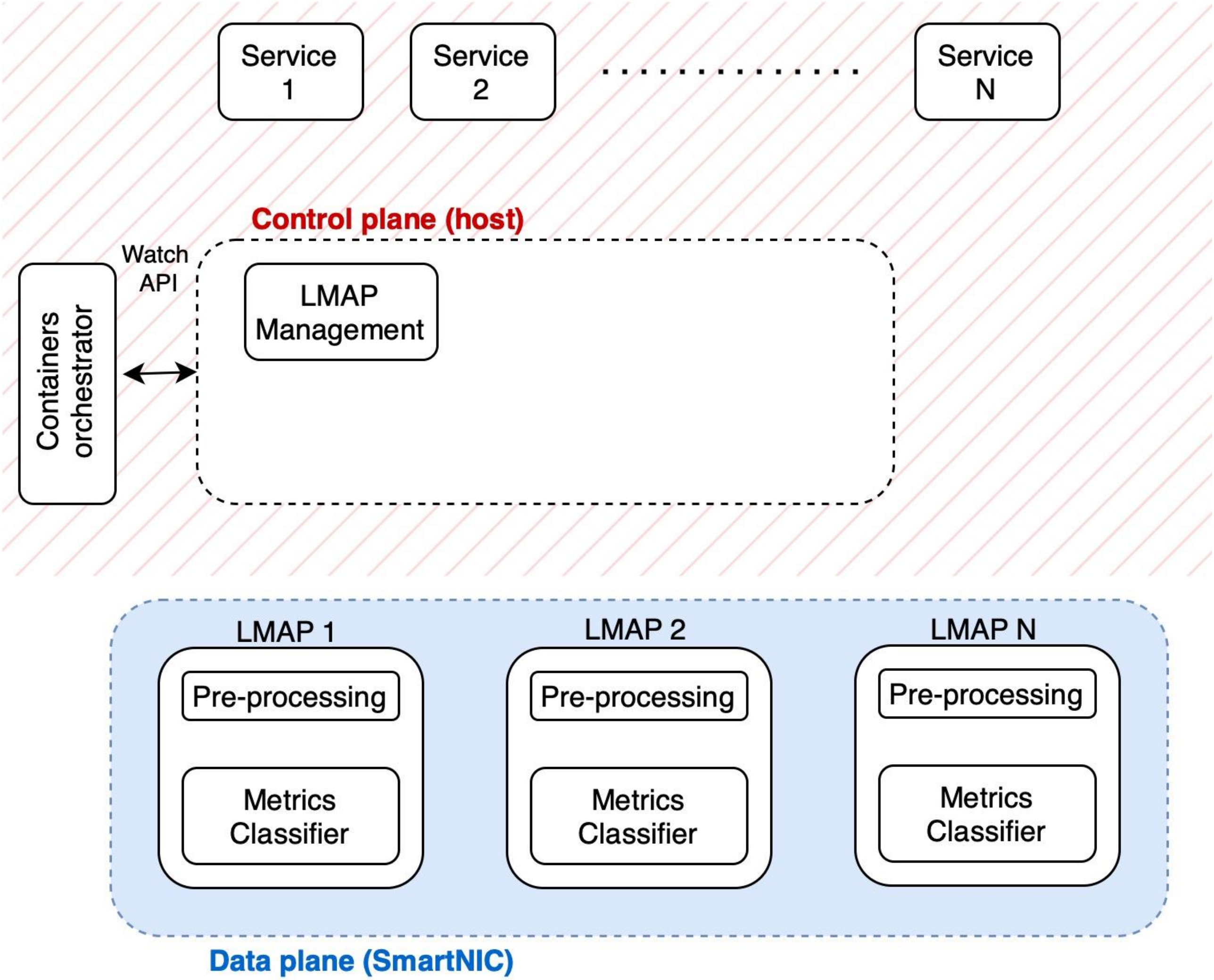
Practicality of anomaly detection

– lightweight to co-exist with other offloads to IPUs
– determine critical metrics for each service
– adjust to workload shift with minimal reconfiguration effort

System architecture

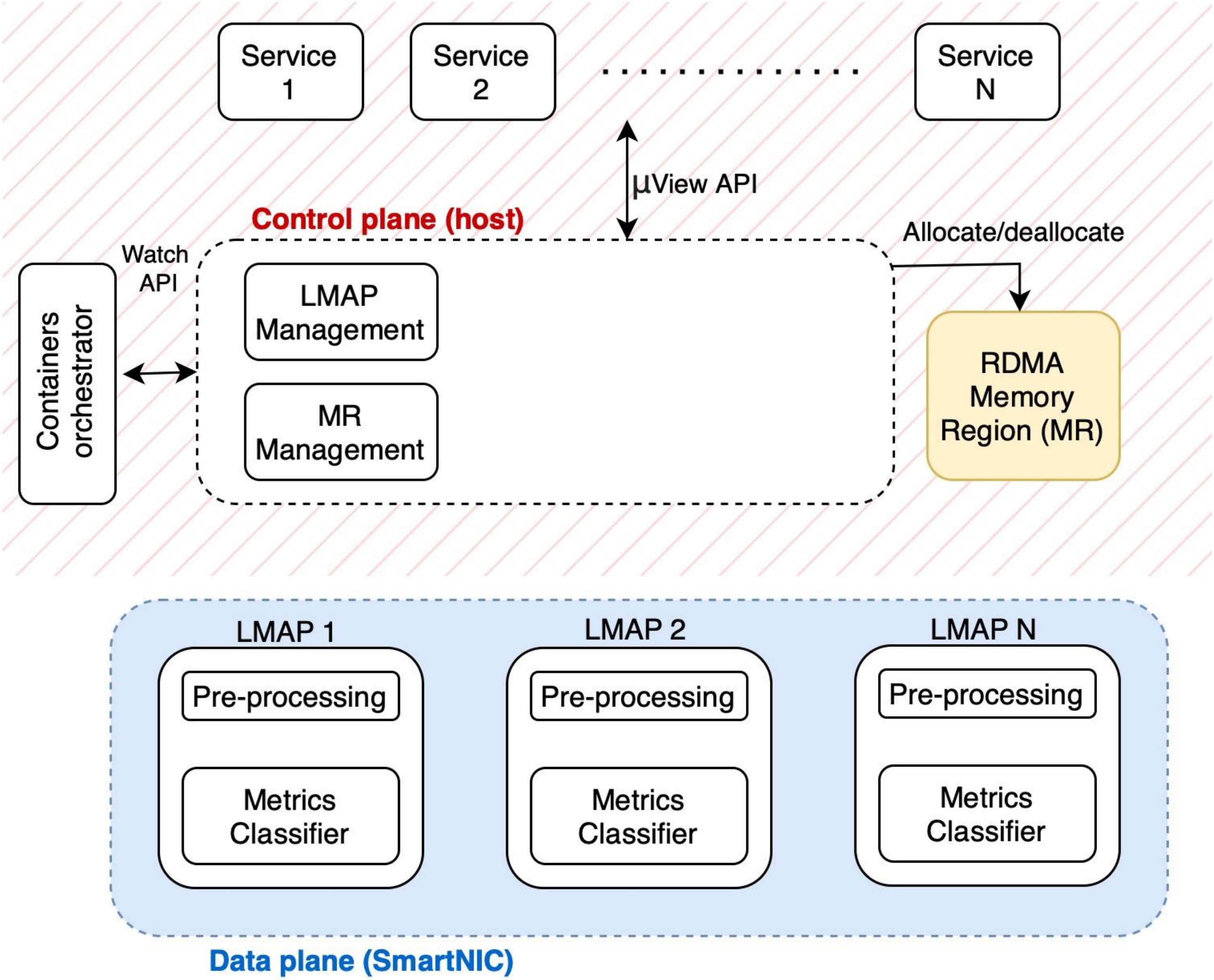
LMAP: Local Metrics Analysis Pipeline

- one LMAP per service



System architecture

- µView API (one-time cost)
  - service registration
  - configure LMAP metrics collection and management
- DMA memory init

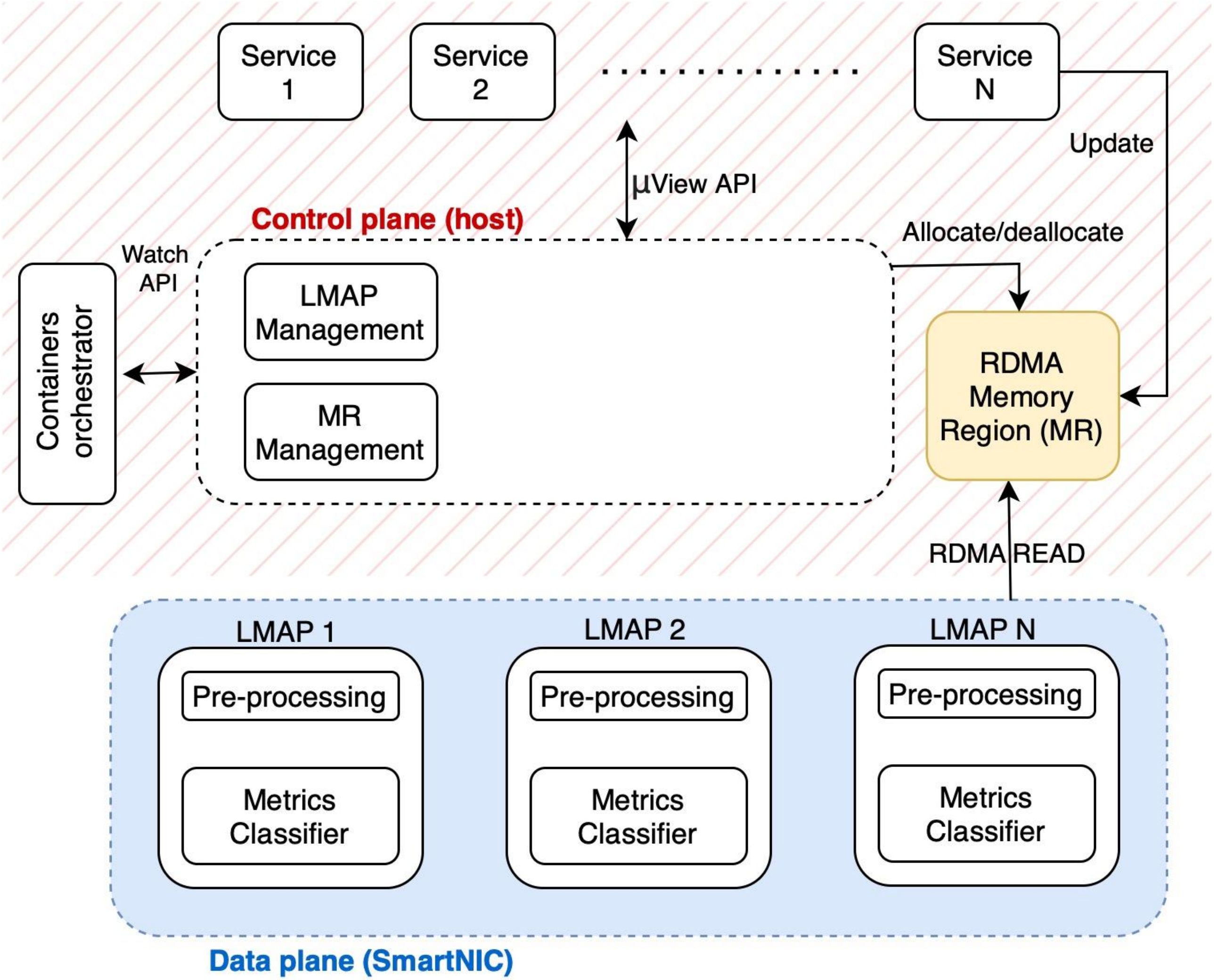


Description	API Call
Manage LMAP	LMAPID **newLMAP**(Config, ServiceID) void **configLMAP**(LMAPID, Dict<ServiceID, List<MetricConfig>>) void **deleteLMAP**(LMAPID)
Configure Metrics	MetricID **addMetric**(LMAPID, Metric, Type, AggType, Frequency) void **deleteMetric**(LMAPID, MetricID)
Add Hooks	HookID **registerHook**(List<LMAPID>, HookFn)
Interface	**Declaration**
HookFn	void _(Feature, Output, AScore)

System architecture

- µView API (one-time cost)
  - service registration
  - configure LMAP metrics collection and management
- DMA memory init
- one-sided RDMA READs
  - to fetch metrics on data-plane
  - **no memory copies overhead!**

Description	API Call
Manage LMAP	LMAPID **newLMAP**(Config, ServiceID) void **configLMAP**(LMAPID, Dict<ServiceID, List<MetricConfig>) void **deleteLMAP**(LMAPID)
Configure Metrics	MetricID **addMetric**(LMAPID, Metric, Type, AggType, Frequency) void **deleteMetric**(LMAPID, MetricID)
Add Hooks	HookID **registerHook**(List<LMAPID>, HookFn)
Interface	**Declaration**
HookFn	void _(Feature, Output, AScore)



Design challenges

Host to SmartNIC data movement ✓

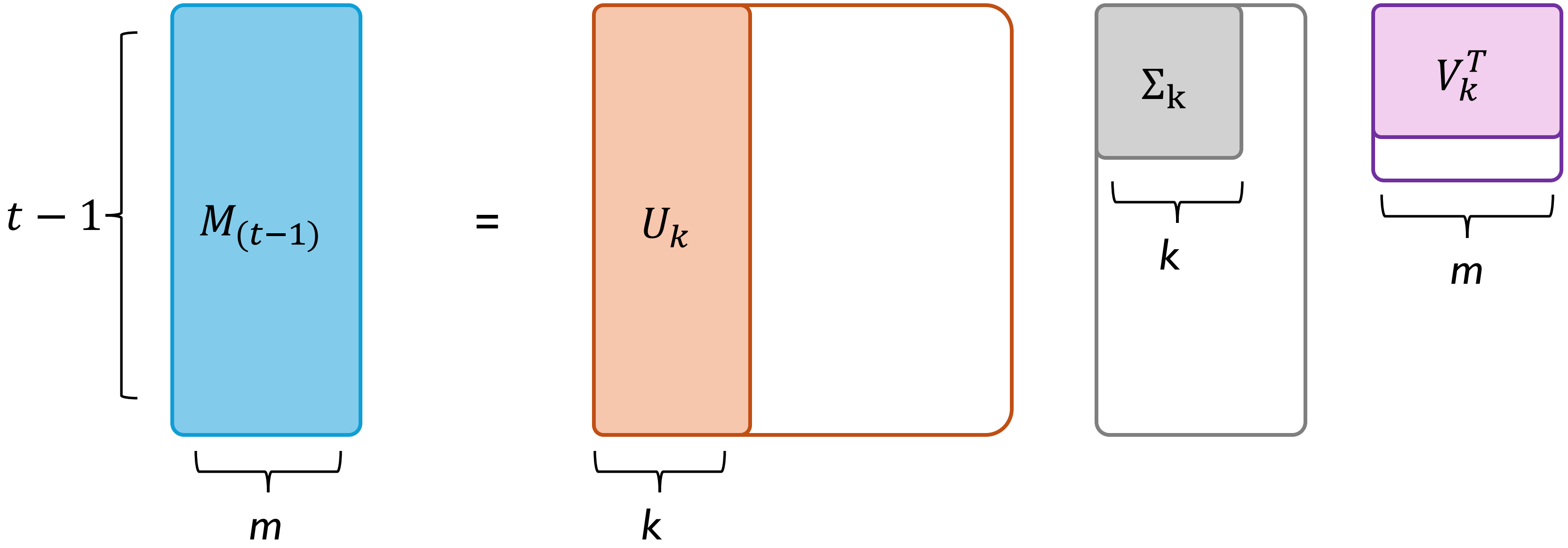
– take data outside the host boundaries, without introducing overhead

Practicality of anomaly detection

– lightweight to co-exist with other offloads to IPUs
– determine critical metrics for each service
– adjust to workload shift with minimal reconfiguration effort

Anomaly detection

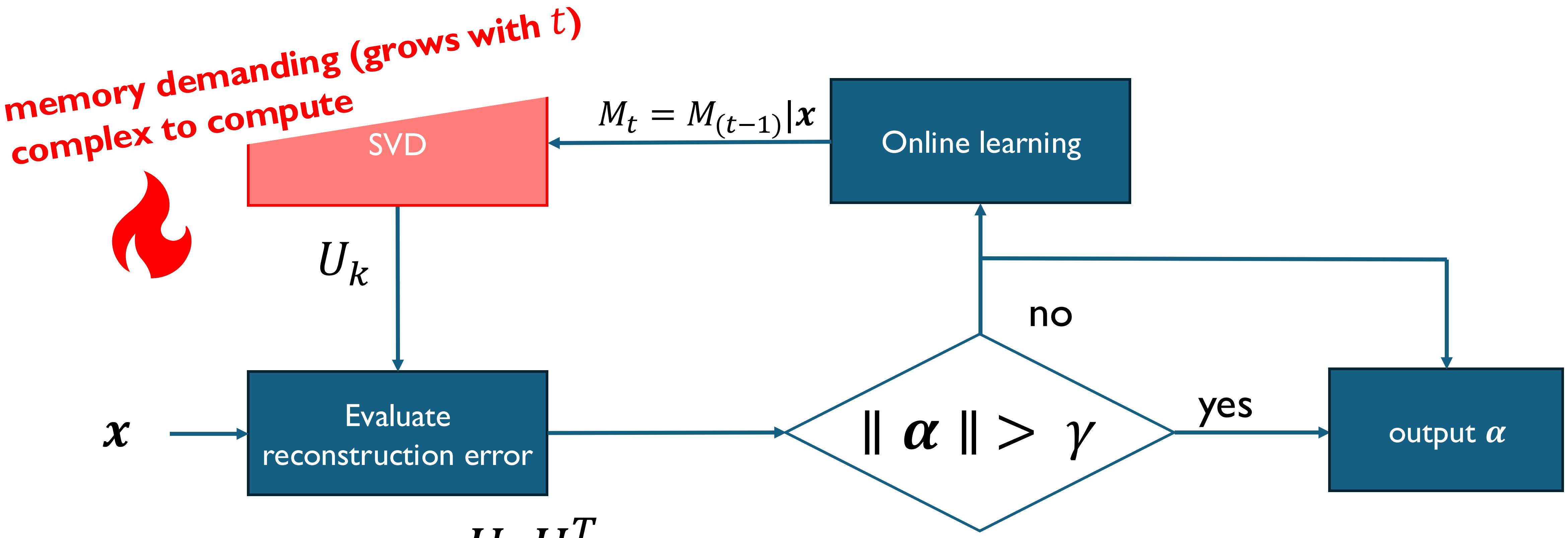
- we borrow from *subspace analysis*
- assume:
  - at time $(t-1)$ we know a non-anomalous metric dataset $M_{(t-1)}$
  - we can compute its *rank-k* $\mathrm{SVD}_k(M_{(t-1)}) = U_k \Sigma_\mathrm{k} V_k^T$



- U_k is a good reconstruction basis for datapoints in $M_{(t-1)}$

Anomaly detection

- at time t, we receive a new vector of metrics $\boldsymbol{x}$



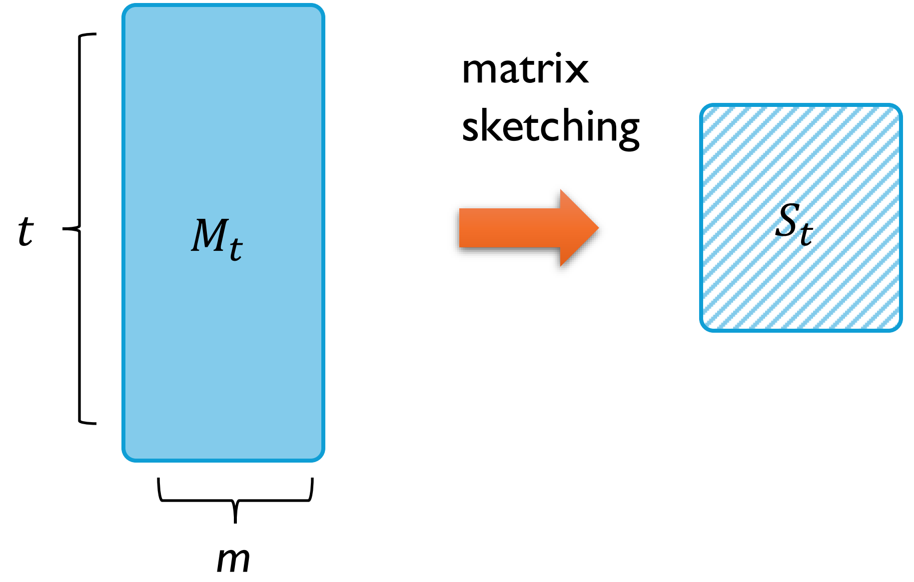
$$\boldsymbol{\alpha} = \boldsymbol{x} - U_k U_k^T \boldsymbol{x}$$

anomaly score vector for each metric

Frequent Direction sketch (practicality!)

Liberty, KDD'13

- matrix sketching: replace M_t with a smaller matrix S_t
  - such that $S_t \approx M_t$



- run SVD on S_t
- streaming operations
  - we can compute S_t using only S_{t-1} and new datapoint $\boldsymbol{x}$
  - never need of storing M_t during runtime

Design challenges

Host to SmartNIC data movement

– take data outside the host boundaries, without introducing overhead

Practicality of anomaly detection

– lightweight to co-exist with other offloads to IPUs
– determine critical metrics for each service
– adjust to workload shift with minimal reconfiguration effort

Evaluation setup

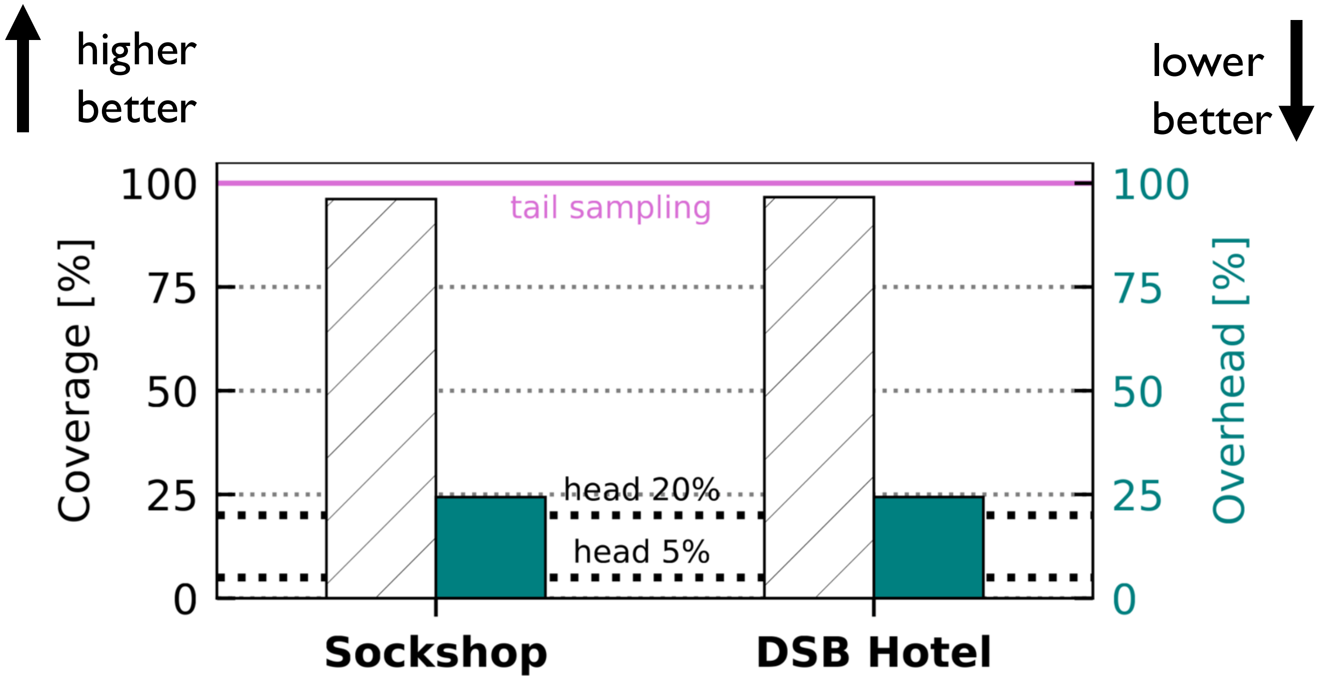
- 4 nodes Kubernetes cluster with Istio service mesh
  - NVIDIA BlueField-2 IPU
- application workloads
  - DeathStarBench (DBS) HotelReservation and Google's SockShop benchmarks
  - synthetic load generation of user requests
- metrics collection
  - container system resource usage (CPU, memory, I/O, network, ..) via cAdvisor
  - service-level e.g., Envoy proxies, Redis key-value stores
  - 1 second local streaming interval host → IPU
- anomaly injection via chaos-engineering

Anomaly type	Injection tools
Memory pressure	ChaosMesh [2], `stress-ng` [30], `pmbw` [88]
LLC pressure	FIRM's `llc.c` [76]
I/O pressure	ChaosMesh [2], `stress-ng` [30]
CPU usage	ChaosMesh [2], `stress-ng` [30]
L7 failure	`redis-cli` [78], ChaosMesh [2]

Table 4: Anomaly injection setup.

μView high fault coverage & low overhead

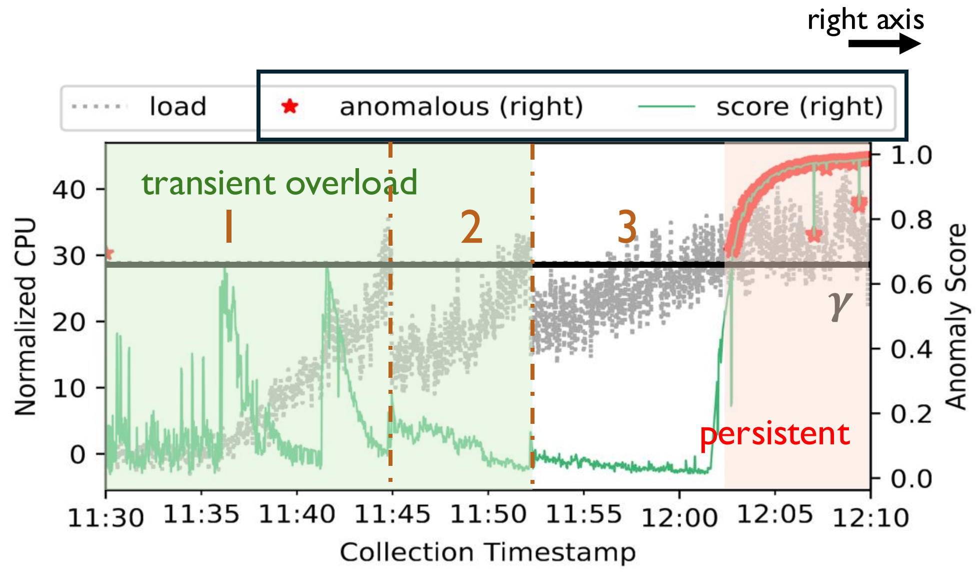
- Trace violate SLOs when:
  - latency above threshold *or* HTTP/gRPC errors
  - threshold : tail latency percentile computed on healthy requests
- Baselines
  - **tail sampling**: always keeps relevant traces, but the collector needs to ingest all traces
  - random **head sampling**: industry de-facto approach

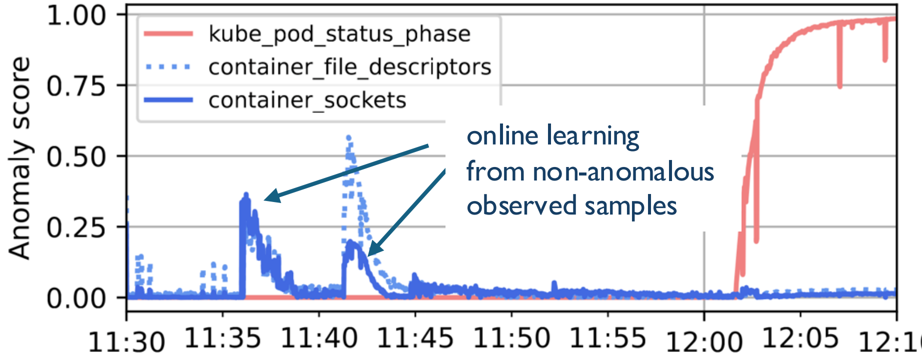


μView adaptation to dynamic workloads

- frontend service + kubernetes HPA autoscaler
  - rescaling rule: service CPU usage above 30%
  - maximum capacity 3 replicas

- goal: distinguish two *overload* conditions
  - *transient*, before rescale [non-anomalous]
  - *persistent*, saturated maximum capacity [anomalous]





Summary

- Observability overheads at cloud-scale
  - remedy in production: coarse-grained sampling ☹
- *ingestion* cost dominates overheads, not *generation* **!**
  - **local processing** at fine temporal granularity ☺

- μView: zooming into microservice state in real-time
  - **informative data**, at **low overhead** (leverage IPUs to offload analysis)
  - practicality
    - lightweight streaming anomaly detection → fits IPUs resource constraints
    - one-catch-all anomaly threshold
  - adaptive to the dynamicity of cloud-native environments

- near-optimal fault coverage for distributed tracing
  - more use-cases in our paper (soon 🤞)