Aalto University
School of Science

Observability, Vulnerability Diagnostics, and Explainability

CS-E4660 Advanced Topics in Software Systems

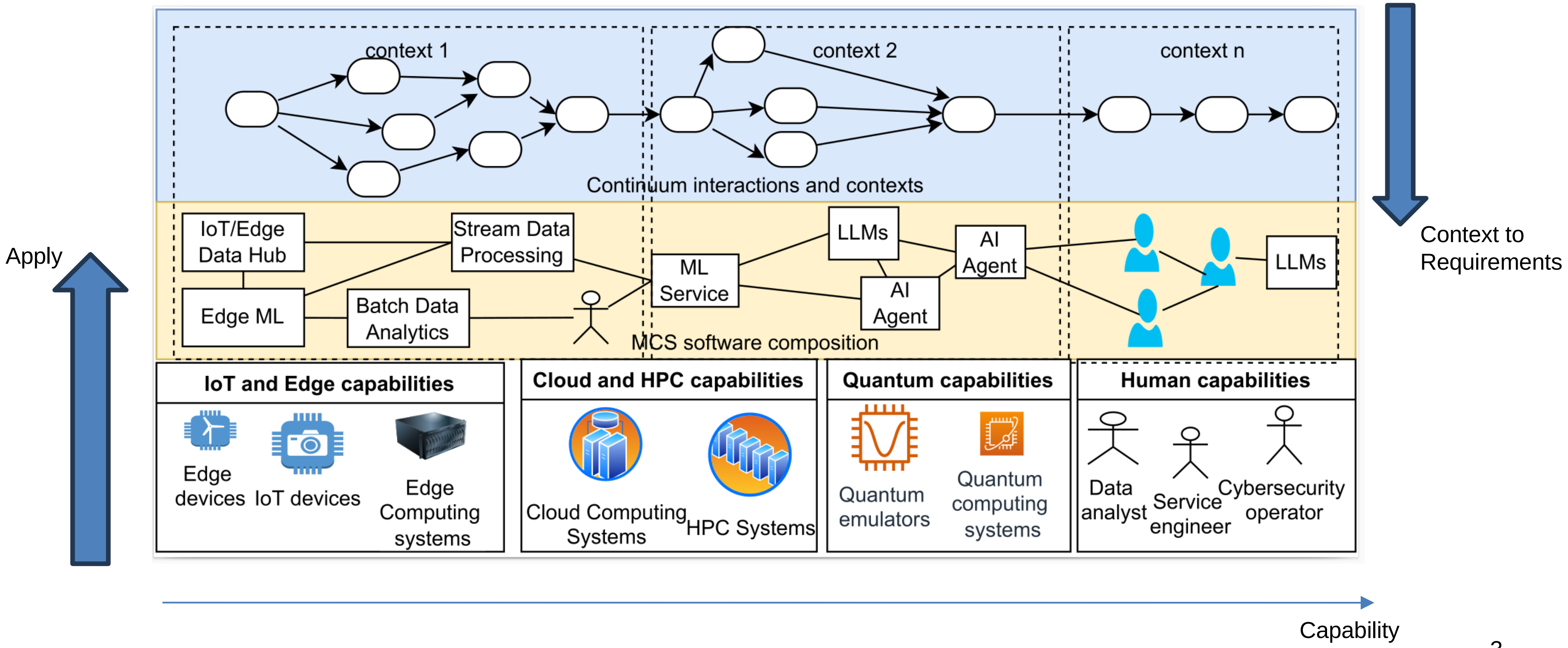
Hong-Tri Nguyen

hong-tri.nguyen@aalto.fi

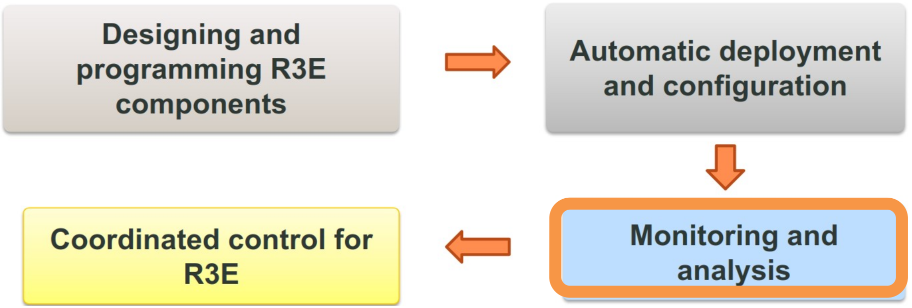
Sep 17, 2025

From previous lectures

Context, composition, and capabilities in multi-continuum computing



Recall: R3E for multi-continuum



- R3E engineering
  - Design and program components
  - Deploy and configure
  - **Monitor** and analyze
  - Coordinate controls
- Coordination for R3E
  - Support: computing, time, intelligence continuum
  - Needs **monitoring/observability**
  - Orchestrate or reactiveness/choreography
  - Multiple systems/infrastructures
    - Computing
    - Time
    - Intelligence

Outline

1. Overview of observability
2. Observability components and steps
3. Root cause analytics and explainability
4. Emerging AI agent observability
5. Take-home message
    - Hand-on preparation

Observability

Observability - optimization and operation

- Proactive issue detection: utilizing observability data to identify potential issues
- Informed decision-making: leveraging insights from observability data to make data-driven decisions

- Support objectives of end-to-end system engineering:
  - Profiling and analytics
    - understand the baseline performance (peak usage patterns and idle times)
  - Predictive scaling up:
    - to feed a predictive model that triggers scaling actions
  - Workload reallocations:
    - shift underutilized to other workloads to free nodes
    - scale down during low need

Observability - the first line of resilience

- Detect anomalies from **behaviors** (ML models, input) to spot irregularities as attacks
  - Data drift detection and feature distribution monitoring reveal shifts in input data to compromise the model
- Trace root causes if existing suspicious predictions
  - **Linking** suspicious predictions to datasets or model versions to identify the source of the problem
- Monitor API usage patterns (way the model is being queried/used)
  - Watching **unusual query patterns**, **degraded performance**, or **usage anomalies** can highlight potential model inversion, or theft

https://owasp.org/www-project-machine-learning-security-top-10/

Observability - trustworthiness

Trust from:

- **Belief**: an expectation that benevolence exists in others
- **Knowledge**: trust develops over time with the accumulation of relevant knowledge
- **System**: a system's ability to meet a set of requirements which will lead that individual to believe that the system can be trusted to perform specific tasks

R3E as key factors of trustworthiness, especially AI usage:

- Valid and reliable: are based on accuracy and robustness
- Secure and resilient: relates to robustness and beyond the data provenance to encompass unexpected or adversarial use
- Accountable and transparent: require training data provenance and AI system decisions -> WHAT happened
- Explainable and interpretable:
  - Explainable: a representation of the mechanisms underlying AI systems' operation, HOW a decision was made in the system
  - Interpretable: the meaning of AI systems' output in the context, WHY a decision was made

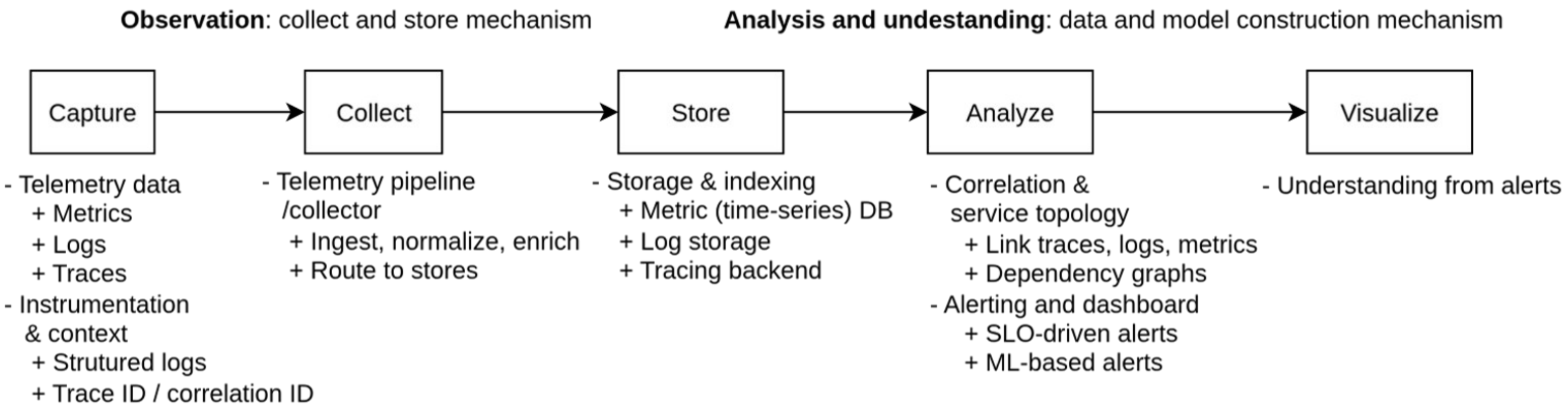
Samonas, Spyridon, and David Coss. "The CIA strikes back: Redefining confidentiality, integrity and availability in security." *Journal of Information System Security* 10.3 (2014).
https://airc.nist.gov/airmf-resources/airmf/3-sec-characteristics/

Observation and analysis

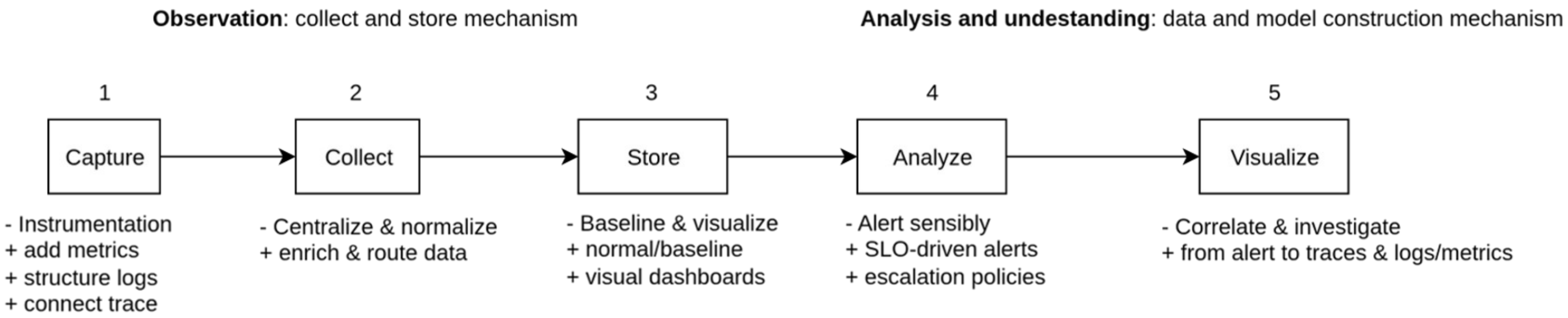
Observation: collect and store mechanism

Analysis and undestanding: data and model construction mechanism

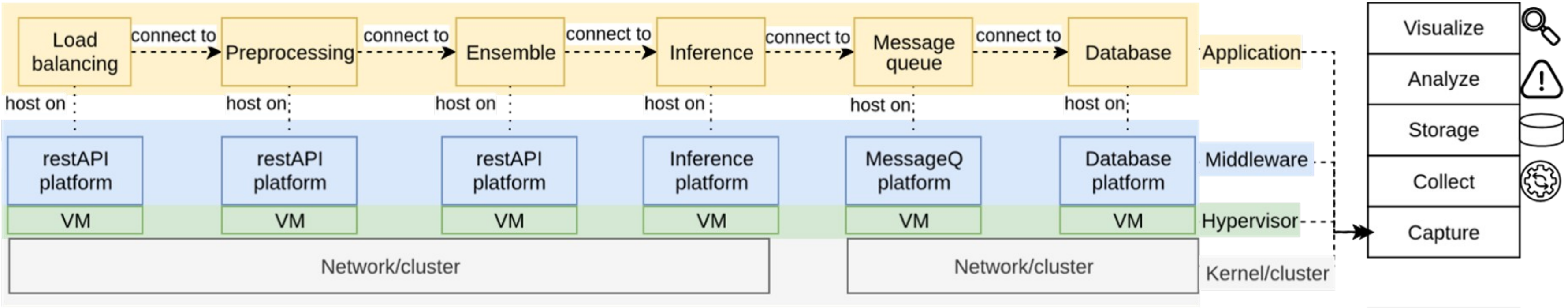
Observability components



Observability steps

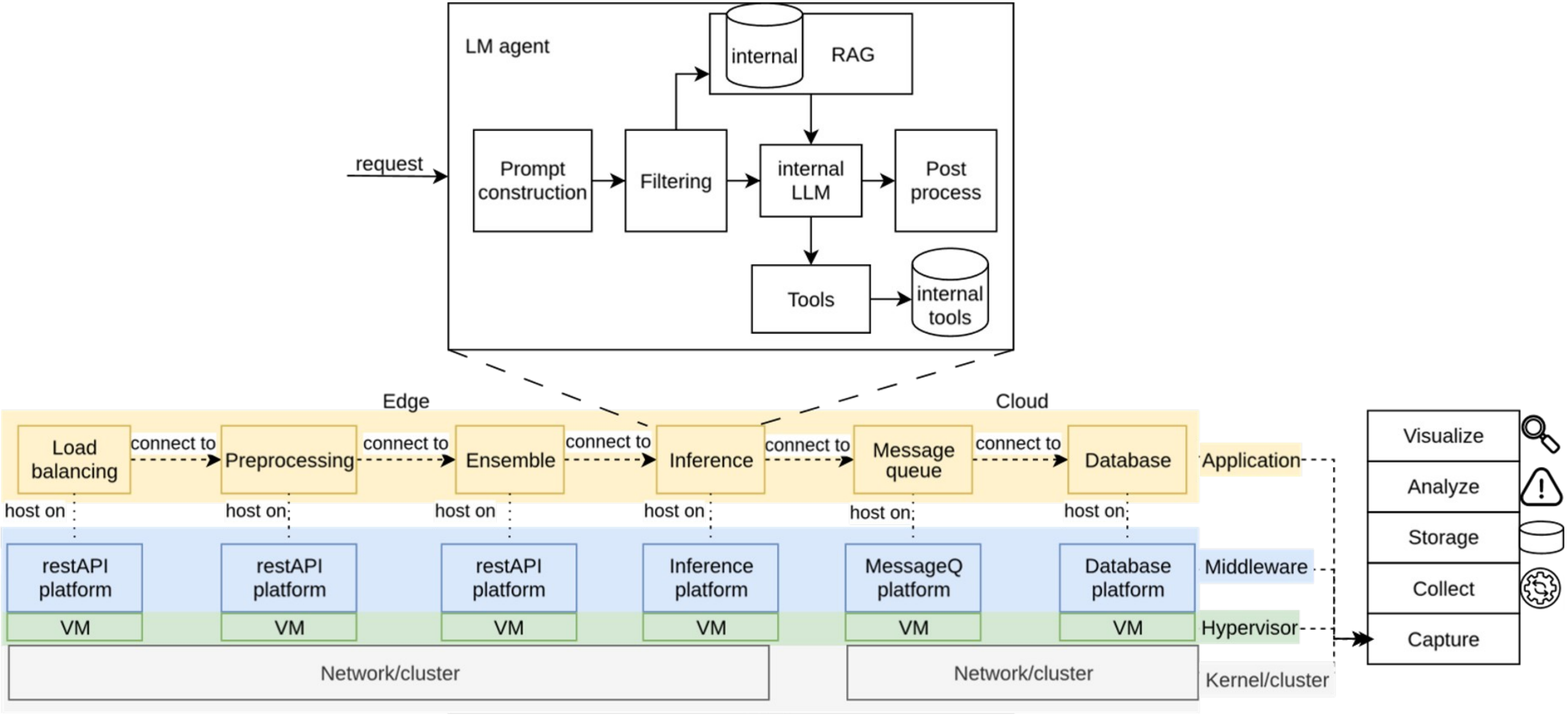


Design observability for an example



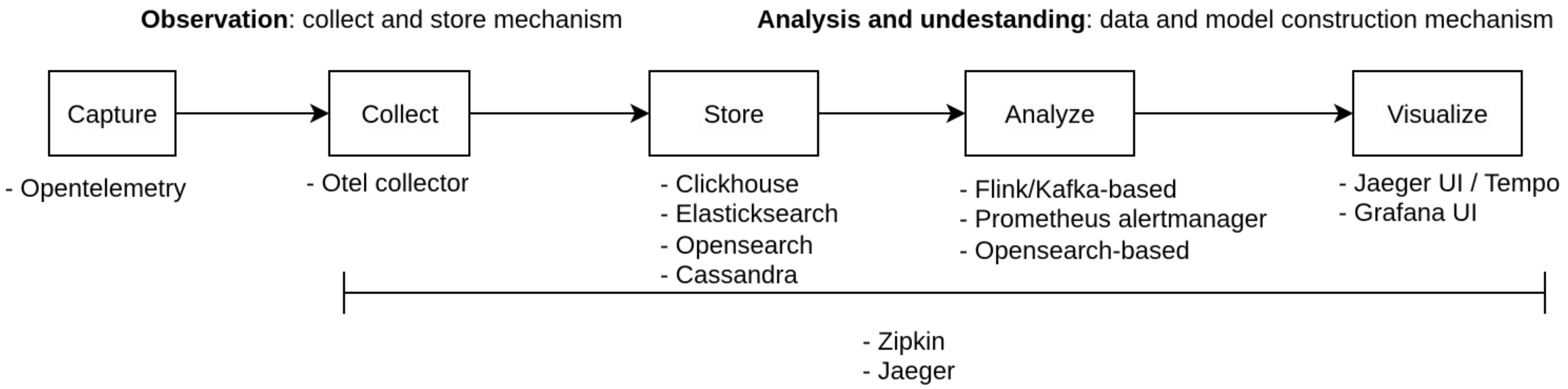
Example for edge-cloud setting for the observability-analysis pipeline

With the AI-based service like LLM agent



Example for edge-cloud setting with an LLM agent for the observability-analysis pipeline

Tools


Observation: collect and store mechanism
Analysis and undestanding: data and model construction mechanism
Capture
Collect
Store
Analyze
Visualize
- Opentelemetry
- Otel collector
- Clickhouse
- Elasticksearch
- Opensearch
- Cassandra
- Flink/Kafka-based
- Prometheus alertmanager
- Opensearch-based
- Jaeger UI / Tempo
- Grafana UI
- Zipkin
- Jaeger

Observability techniques

Telemetry – data

- Log: are detailed chronological records of **specific events** that occur within a system. Logs offer a granular view of what happened within the system
- Metric: provide a broad view of **system health**, consisting of quantitative data that measures various aspects of system performance and resource utilization
- Trace: track end-to-end insight **the flow of a request** as it travels through multiple components of a system

Sridharan, Cindy. Distributed systems observability: a guide to building robust systems. O'Reilly Media, 2018.

Purpose of each telemetry

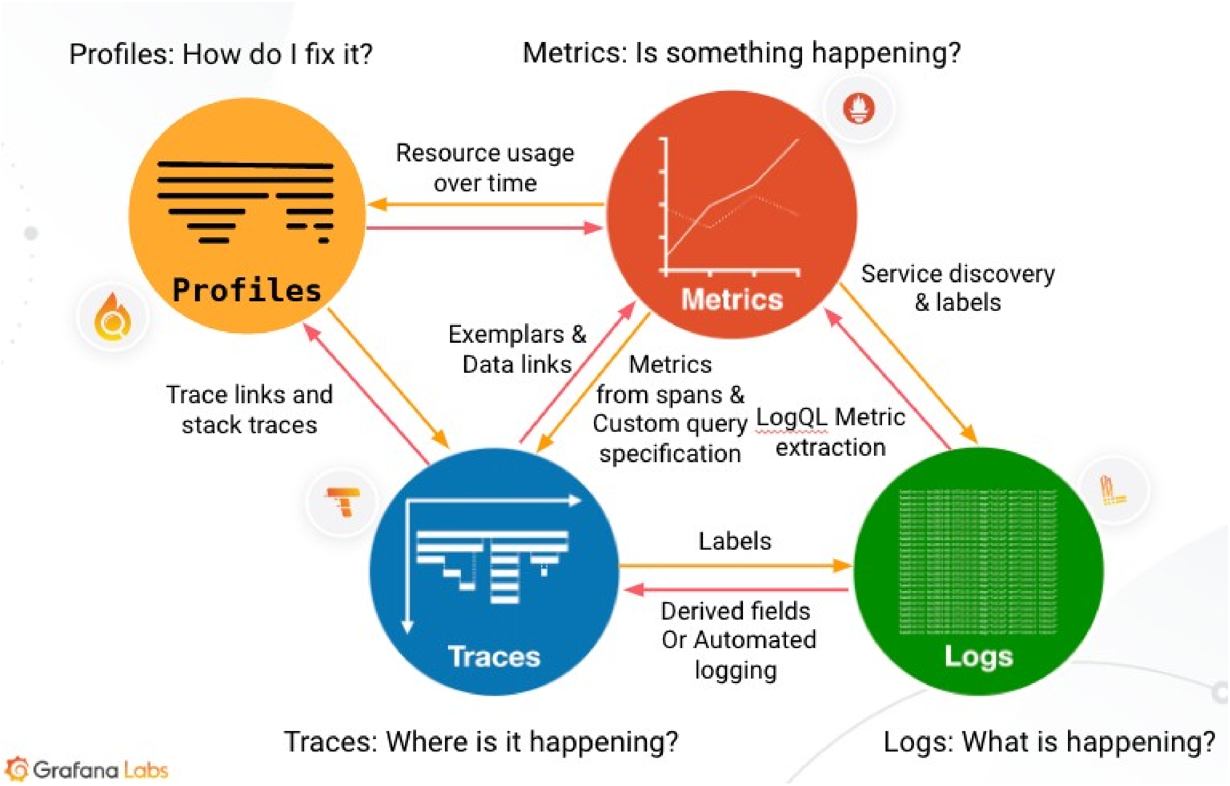
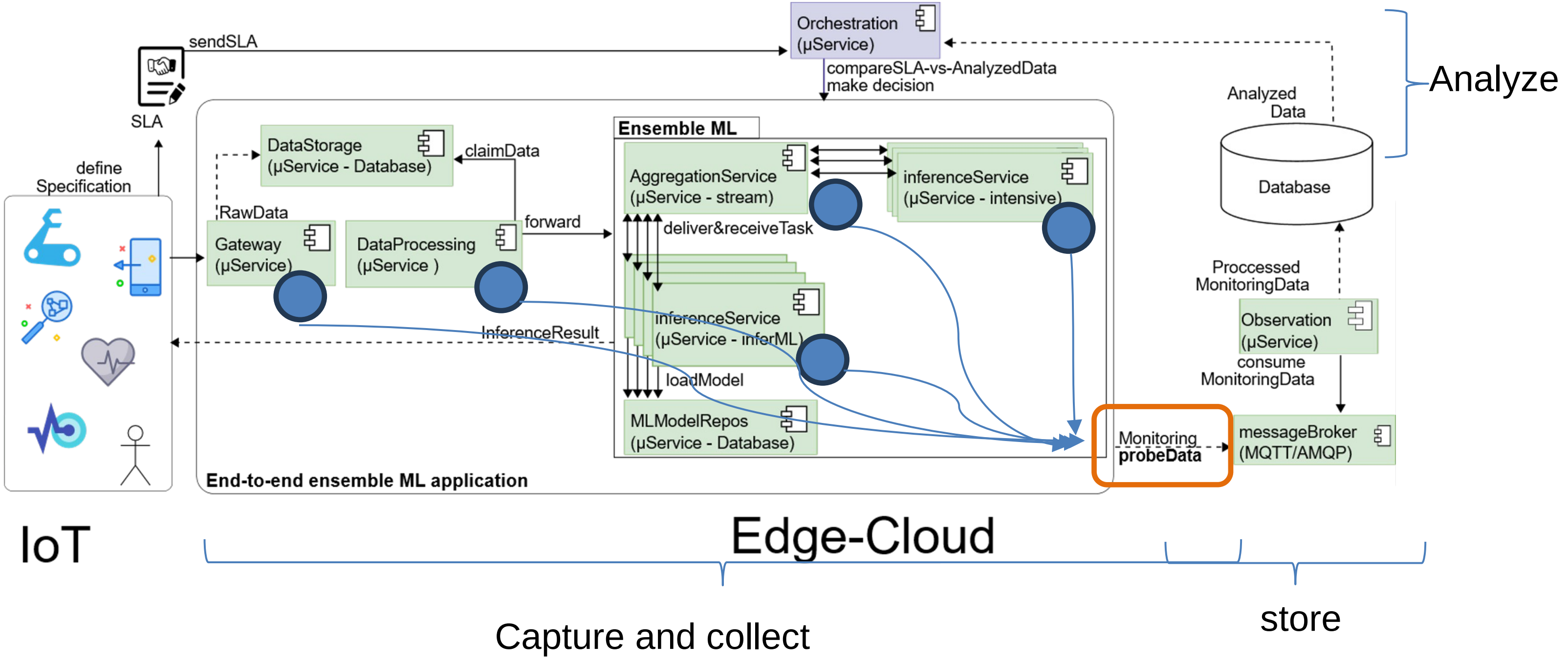


Figure source: https://grafana.com/docs/tempo/latest/introduction/telemetry/

Example: Collect data for observability



Truong, Hong-Linh, and Tri-Minh Nguyen. "QoA4ML-A framework for supporting contracts in machine learning services." *2021 IEEE International Conference on Web Services (ICWS)*. IEEE, 2021.

Distributed tracing

Definition

- Distributed tracing is a method to monitor applications, by recording and tracking or logging end-to-end requests when they flow through various services or components.

Trace

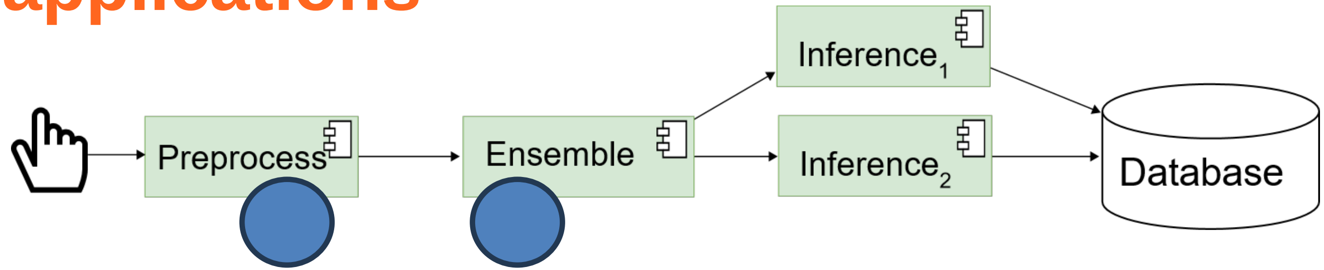
- A trace is a record of a request, capturing a set of spans, events, and annotations with timestamps and ordering from every machine that the request traverses. As a Directed Acyclic Graph (DAG) formed by spans

Span

- A span, constituting a unit of work in a trace tree. Each edge in the trace tree represents the causal relationship among spans. A span model follows (timestamp, operation) pairs

Y. Shkuro, Mastering Distributed Tracing: Analyzing performance in microservices and complex systems. Packt Publishing Ltd, 2019.

Service-based applications




```
with tracer.start_as_current_span("preprocess"):
    with tracer.start_as_current_span("preprocess-ensemble"):
        requested = get(
            "http://ensemble:8082/server_request",
            params={"param": param_value},
            headers=headers,
        )
```

```
@app.route("/server_request")
def server_request():
    with tracer.start_as_current_span(
        "server_request",
        context=extract(request.headers),
        kind=SpanKind.SERVER,
        attributes=collect_request_attributes(request.environ),
    ):
```

Tracing Instrumentation: auto and manual

- Hooks HTTP/gRPC, DB calls, messaging, popular frameworks automatically
- Use
  - Bootstrapping tracing, many services portfolios, uniform stacks.
- Pros
  - Fast coverage, **zero/low code**
  - Consistent naming
  - Good baseline
- Cons
  - Generic spans (little domain context)
  - Fragile across framework/library versions
  - Potential overhead if everything is traced

- Add spans around applications and critical dependencies

Use

- Key flows (checkout, payment), async/long-running work, known dependencies

Pros

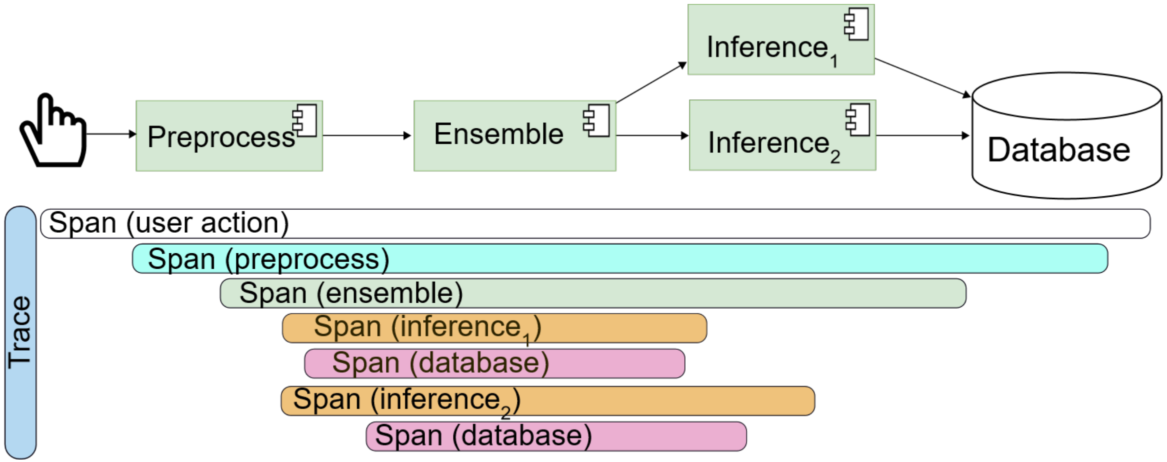
- Precise boundaries & domain-rich attributes
- Aligns traces to SLOs/user journeys
- Control sampling & naming

Cons

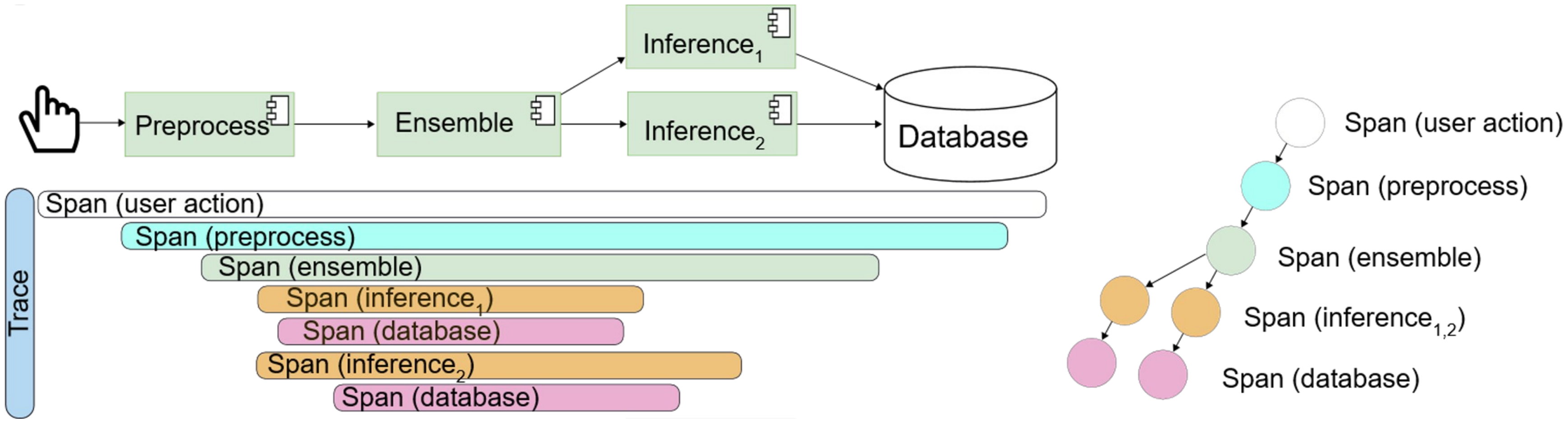
- Engineering time; code noise
- Risk of drift without conventions/reviews

https://opentelemetry.io/docs/collector/management/

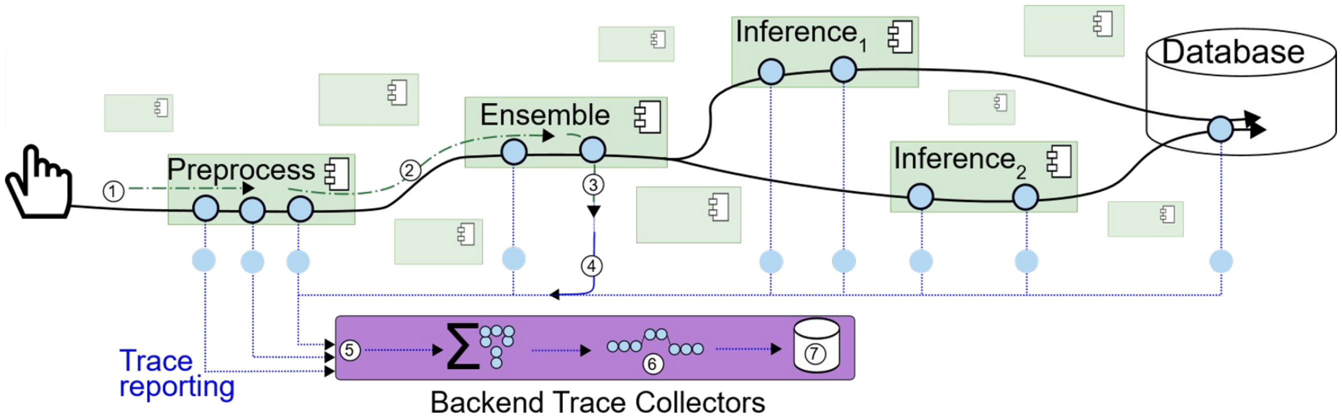
Service-based applications



Service-based applications



Architecture for distributed tracing



A request traverses system processes: **(1)** assigning a unique traceID, **(2)** propagating traceID and sampled flag, **(3)** annotating with traceId, **(4)** transmitting trace data, **(5)** backend receives, **(6)** processing, **(7)** storing

Example: Canopy: an end-to-end tracing

(1) Instrumentation

(2) Request traverses instrumented components

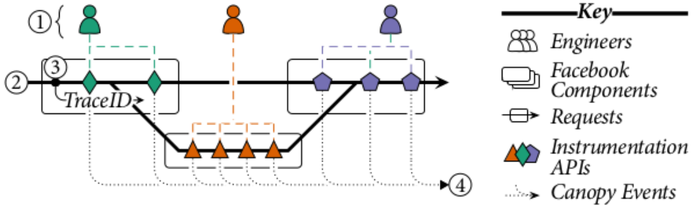
(3) Requests has a TraceID along with the path

(4) Generate and edit events

(5) agrreagate events

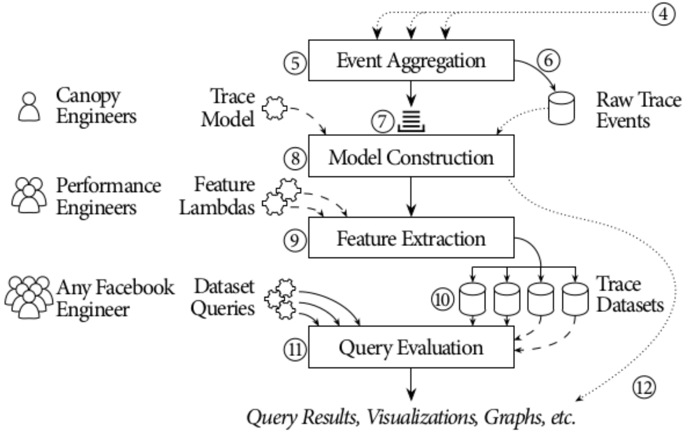
(6) store for (7) and (8) construct trace

Capture and collect



(a) Engineers instrument Facebook components using a range of different Canopy instrumentation APIs (①). At runtime, requests traverse components (②) and propagate a TraceID (③); when requests trigger instrumentation, Canopy generates and emits events (④).

Storage

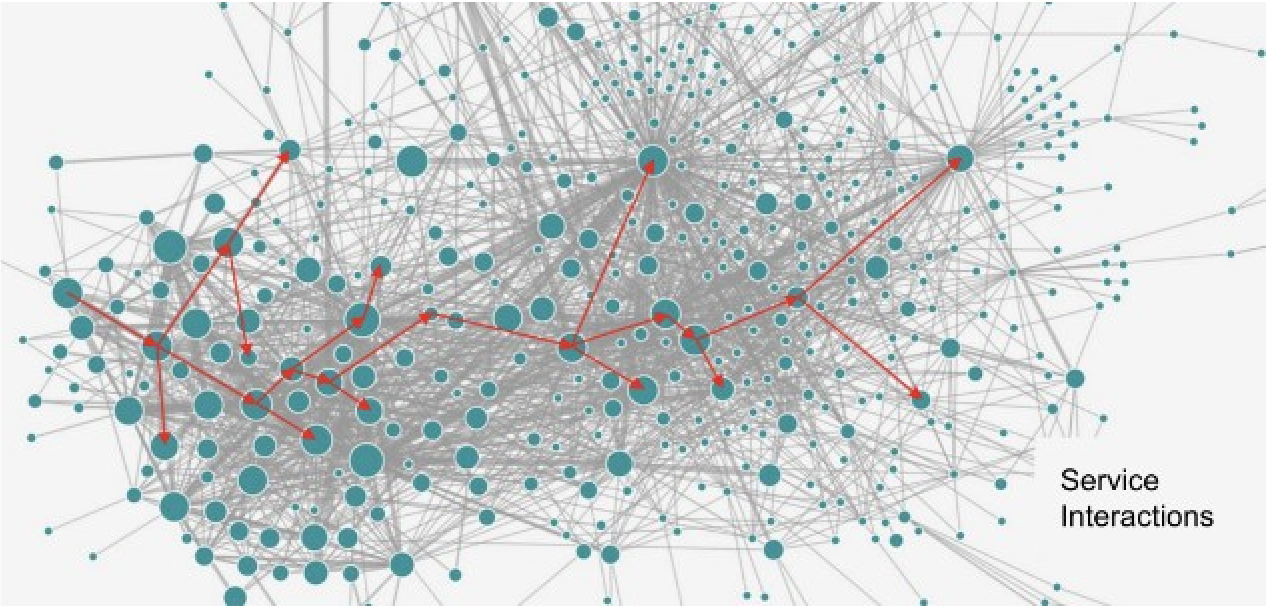


(b) Canopy's tailer aggregates events (⑤), constructs model-based traces (⑧), evaluates user-supplied feature extraction functions (⑨), and pipes output to user-defined datasets (⑩). Users subsequently run queries, view dashboards and explore datasets (⑪,⑫).

Figure 2: Overview of how (a) developers instrument systems to generate events and (b) Canopy processes trace events (cf. §3.1).

Kaldor, Jonathan, et al. "Canopy: An end-to-end performance tracing and analysis system." *Proceedings of the 26th symposium on operating systems principles*. 2017.

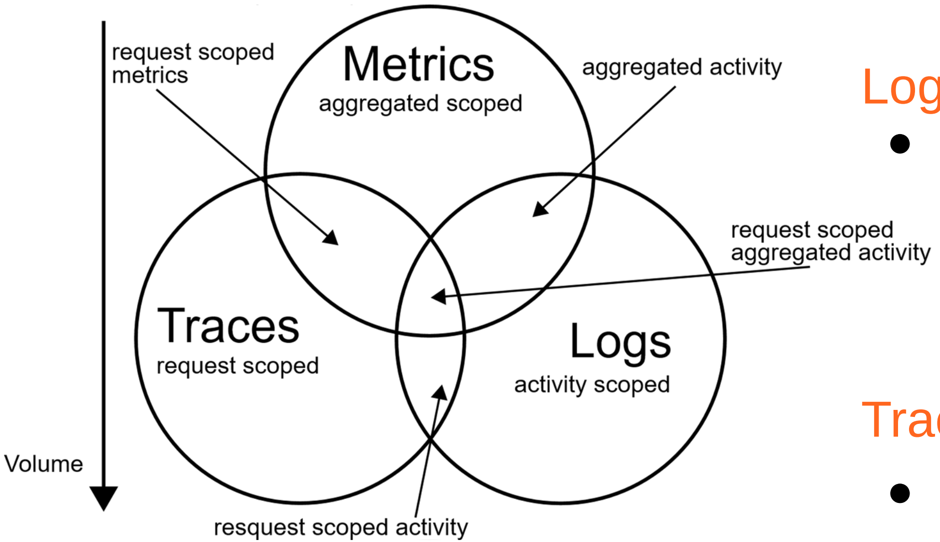
Example of whole view



RPC graph at Uber [5]

Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, “CRISP: Critical path analysis of Large-Scale microservice architectures,” in 2022 USENIX Annual Technical Conference (USENIX ATC 22)

Comparison



T. Wang and G. Qi, "A comprehensive survey on root cause analysis in (micro) services: Methodologies, challenges, and trends," 2024.

Metrics (CPU, I/O activity, network traffic)

- For precise fault localization via considering performance indicator **at specific point**
  - Lack: relationships among metric data, sequential triggering of alerts

Logs (application, system, and network)

- A lot of **operational status** information, error/warning messages for forensic evidence a single process or transaction
  - Challenges: unstructured nature, diverse formats, vast volume

Traces

- Valuable for documenting and analyzing request path to show **critical insights:**
  - Interconnected relationships
  - fine-grained information invaluable for graph-based RCA

Analytics

Root cause analysis

Create fault-free patterns/features

- Presentation of observed data, such as correlation of metrics, error ratios, service graph, throughput or span duration

Anomaly detection

- Time-series technique for alerting abnormal events via the dissimilarity between fault-free features and the run-time operations/behaviors
  - SLO thresholds
  - Statical methods or multivariated ML detectors (autoencoders, LSTM)
  - Graph/Event-based detectors

Ranking fault candidates

- Ranking the list of candidates via various scores, mostly based on probability

Examples: RCA

- TraceRCA [1] uses (1) multi-modal observe data to present features for fault invocation detection, (2) microservice anomaly localization based on metrics from percentages correlation between normal and abnormal traces, (3) ranking microservices via in/out invocation
- MRCA [2] (1) feature learning based on auto encoder (log parsing, latency from trace); (2) anomaly detection (3) root cause localization – causal analysis via the metric data
- Nezha [3] (1) construction phase is about data integration and pattern mining; (2) production phase
  - Anomaly detector from the performance
  - Data integrator unifies the multi-modal data into event graphs
  - Pattern miner extracts patterns and calculates supports
  - Ranker ranks a list of candidate

1.Li, Zeyan, et al. "Practical root cause localization for microservice systems via trace analysis." *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*. IEEE, 2021.
2. Wang, Yidan, et al. "MRCA: Metric-level root cause analysis for microservices via multi-modal data." *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2024.
3. Yu, Guangba, et al. "Nezha: Interpretable fine-grained root causes analysis for microservices on multi-modal observability data." *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2023.

Explainability

What is explanation?

- Explain the execution flow from input data to inference results (underlying AI systems' operation) **HOW** a decision was made in the system

Capture meaningful data for report construction

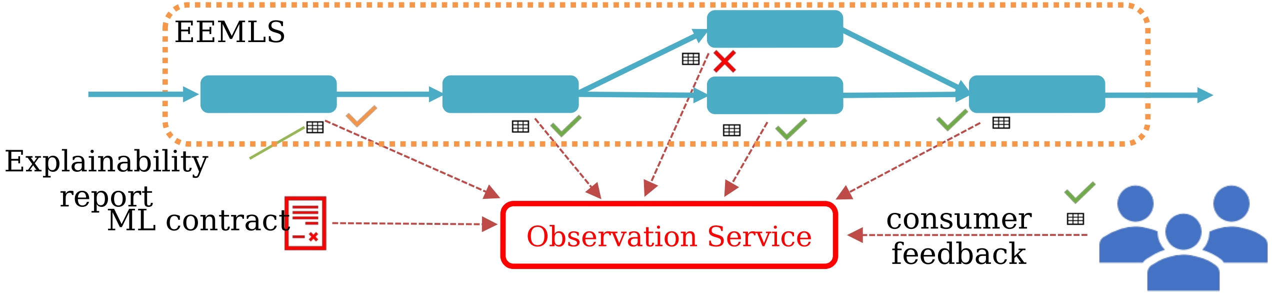
- Determine specific data, usually ML-based data like accuracy and confident with false negative rate
- Construct reports for further evaluation

Evaluate the violation

- Dissimilarity between the reports and predefinition of contracts

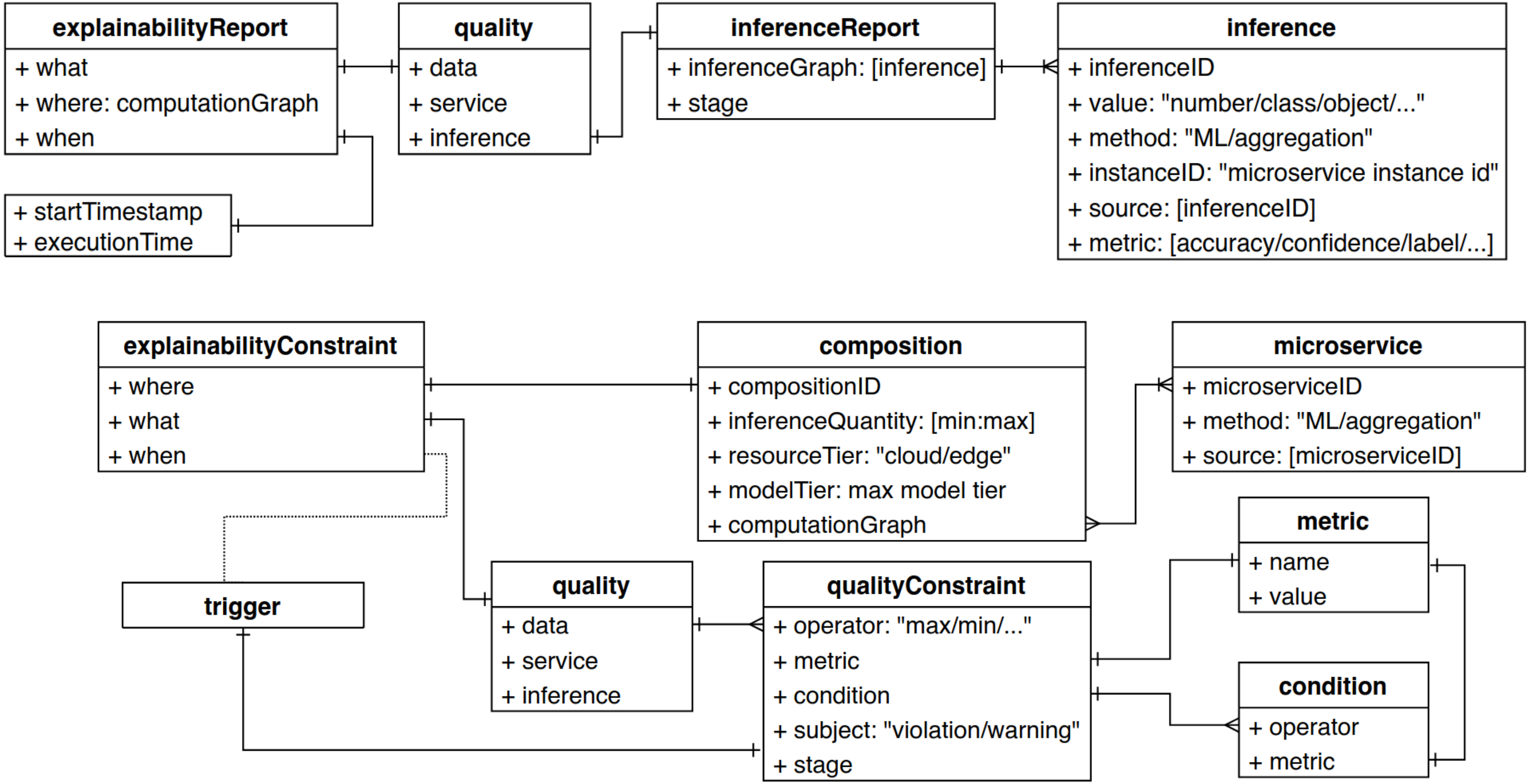
Example

- Explainability for ML results ensemble [1] (1) monitoring probes collect ensemble service operations (ML accuracy and false negative rate) and consumer (2) observation agent compares those requirements



1. Nguyen, Minh-Tri, et al. "Novel contract-based runtime explainability framework for end-to-end ensemble machine learning serving." CAIN 2024.

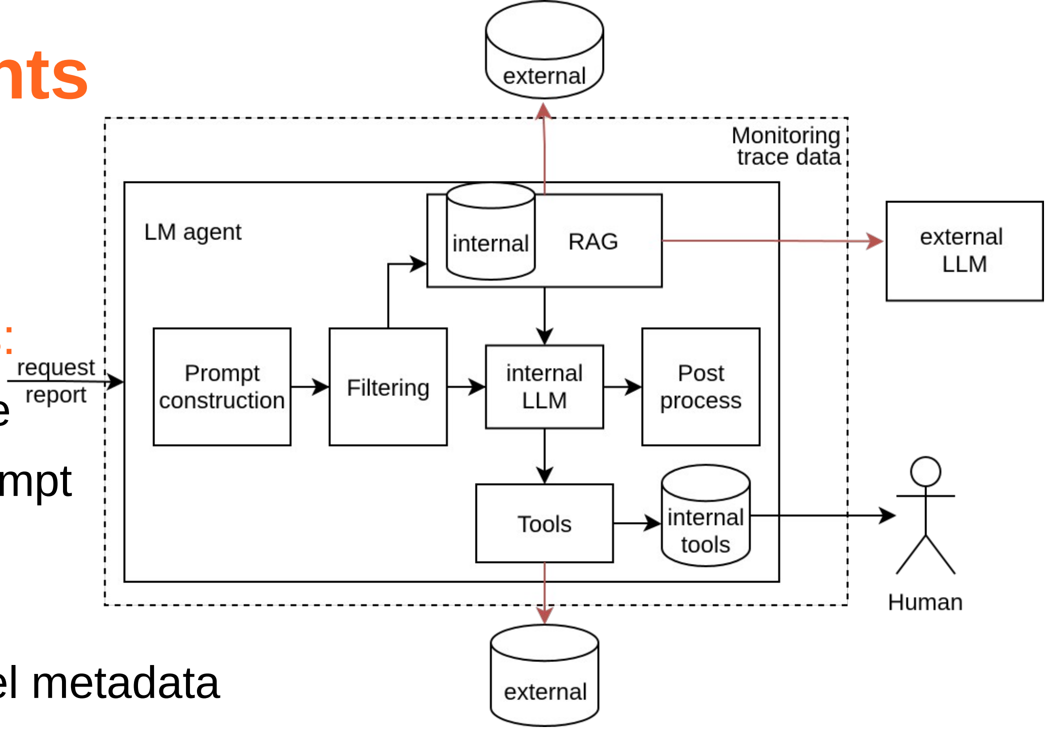
Example



1. Nguyen, Minh-Tri, et al. "Novel contract-based runtime explainability framework for end-to-end ensemble machine learning serving." CAIN 2024.

Emerging AI agent observability

Observability for AI agents



Why need observability for AI agents:

- Non-deteministic harder to root-cause
- New failure modes: hallucination, prompt injection, retrieval failure

What to be captured

- Prompt / model response along model metadata
- Confidence/calibration/hallucination scores
- Similarity score of documents (RAG provenance)

Evaluate the violation for trustworthiness

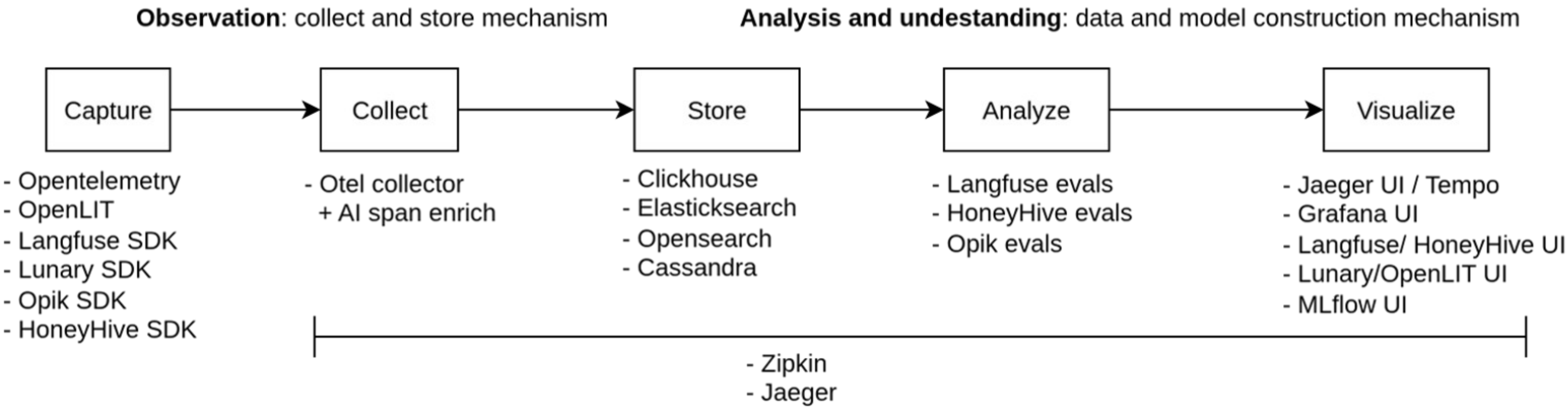
- R3E as the key factor for trustworthiness

Example: AI observability framework

- Single agent: **AgentSight** intercepts LLM traffic instead of source for extracting semantic intent (Gap between Intent and Action) and kernel events (dynamic in-kernel eBPF filter)
  - Uprobes: decrypt LLM communication and monitor syscalls: openat2, connection, exceve
  - Detect: prompt injection attack, identifies resource-wasting reasoning, reveals hidden coordination bottlenecks
- Multi-agent system: **LumiMAS** works to detect hallucination and bias
  - Monitoring: Application start/end, Agent start/end,LLM calls, tool usage, token usage, semantic interaction
  - Anomaly detection: an LSTM-based autoencoder architecture
  - Investigation: root cause analysis LLM-based agent

Zheng, Yusheng, et al. "AgentSight: System-Level Observability for AI Agents Using eBPF." *arXiv preprint arXiv:2508.02736* (2025).
Solomon, Ron, et al. "LumiMAS: A Comprehensive Framework for Real-Time Monitoring and Enhanced Observability in Multi-Agent Systems." *arXiv preprint arXiv:2508.12412* (2025)

Observability tools for AI agent



https://github.com/ollama/ollama

Jose, Edwin, and Prasad Prabhakaran. "Harnessing Large Language Models (LLMs) Optimizing Performance, Monitoring, and Compliance." Authorea Preprints (2024).

Take-home messages

Take-home messages

Observability is the foundation for many purposes: optimization, defense, trust
Observability along with infrastructure to build a dataset for analytics
Components along steps: instrumentation, collectors, and storage
Techniques for an end-to-end service-based application

- Metrics: localizing fault, but lack of causal links
- Logs: information, but unstructure and vast volume
- Traces: relationship among requests, but scalability and vast volume
- Multi-modal: to understand more WHERE, WHAT are happenings and Is sth happening

Need more data? -- Did you check hypervisor and kernel layers?

Too much data? -- reducing observability data via sampling

- Head-based solution randomly select traces (usually 1%)
- Tail-based solution ML/AI-based to filter

Hands-on preparation

Tools:

- Application: **python-based**/go-based apps
- Container: **Docker**/container-d
- Cluster: **Minikube**/k8s or k3s
  - **Kubectl**
  - **Helm**
  - (options) Istio/envoy/istioctl
- Observability: Opentelemetry/jaeger -- Documentation
  - Concepts
  - Architecture/components
- LLM tools:
  - Langfuse/OpenLIT

Study logs

- What does it mean about the observation and analysis?
  - Trade-off or side effect from observation

- Which components or steps in observation are the most important from your perspective? Why?

- Write short your thought on that and send to the Mycourse using at least 1-2 references

Thank you

Q&A