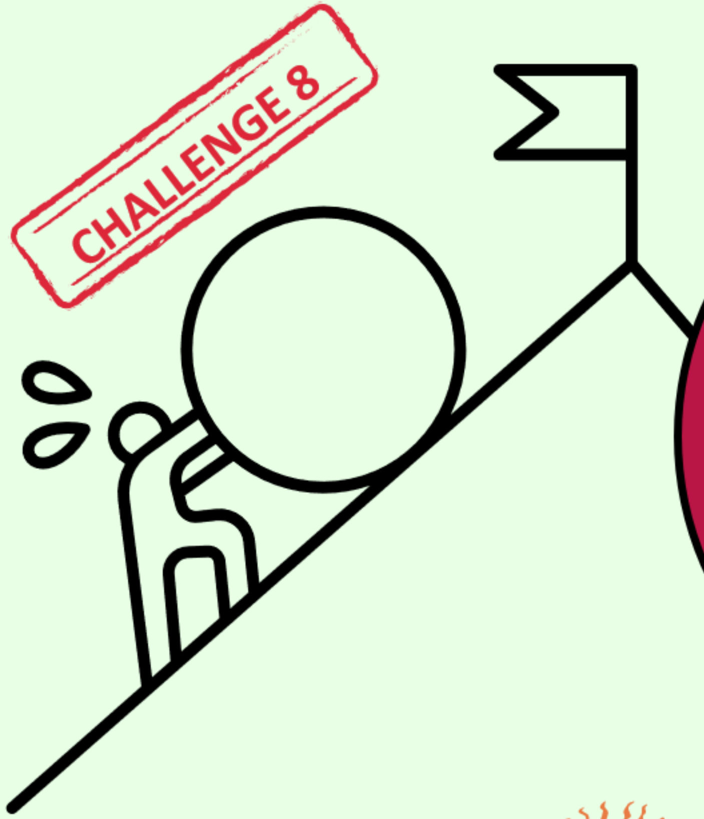
OBSERVABILITY AND MONITORING OF MICROSERVICES

eazy bytes



DEBUGGING A PROBLEM IN MICROSERVICES ?

How do we trace transactions across multiple services, containers and try to find where exactly the problem or bug is?

How do we combine all the logs from multiple services into a central location where they can be indexed, searched, filtered, and grouped to find bugs that are contributing to a problem?

MONITORING PERFORMANCE OF SERVICE CALLS?

How can we track the path of a specific chain service call through our microservices network, and see how long it took to complete at each microservice?

MONITORING SERVICES METRICS & HEALTH ?

How can we easily and efficiently monitor the metrics like CPU usage, JVM metrics, etc. for all the microservices applications in our network?

How can we monitor the status and health of all of our microservices applications in a single place, and create alerts and notifications for any abnormal behavior of the services?

Observability and **monitoring** solve the challenge of identifying and resolving above problems in microservices architectures before they cause outages.

WHAT IS OBSERVABILITY ?

eazy bytes

Observability is the ability to understand the internal state of a system by observing its outputs. In the context of microservices, observability is achieved by collecting and analyzing data from a variety of sources, such as metrics, logs, and traces.

The three pillars of observability are:

Metrics: Metrics are quantitative measurements of the health of a system. They can be used to track things like CPU usage, memory usage, and response times.

Logs: Logs are a record of events that occur in a system. They can be used to track things like errors, exceptions, and other unexpected events.

Traces: Traces are a record of the path that a request takes through a system. They can be used to track the performance of a request and to identify bottlenecks.

By collecting and analyzing data from these three sources, you can gain a comprehensive understanding of the internal state of your microservices architecture. This understanding can be used to identify and troubleshoot problems, improve performance, and ensure the overall health of your system.

WHAT IS MONITORING ?

eazy bytes

Monitoring in microservices involves checking the telemetry data available for the application and defining alerts for known failure states. This process collects and analyzes data from a system to identify and troubleshoot problems, as well as track the health of individual microservices and the overall health of the microservices network.

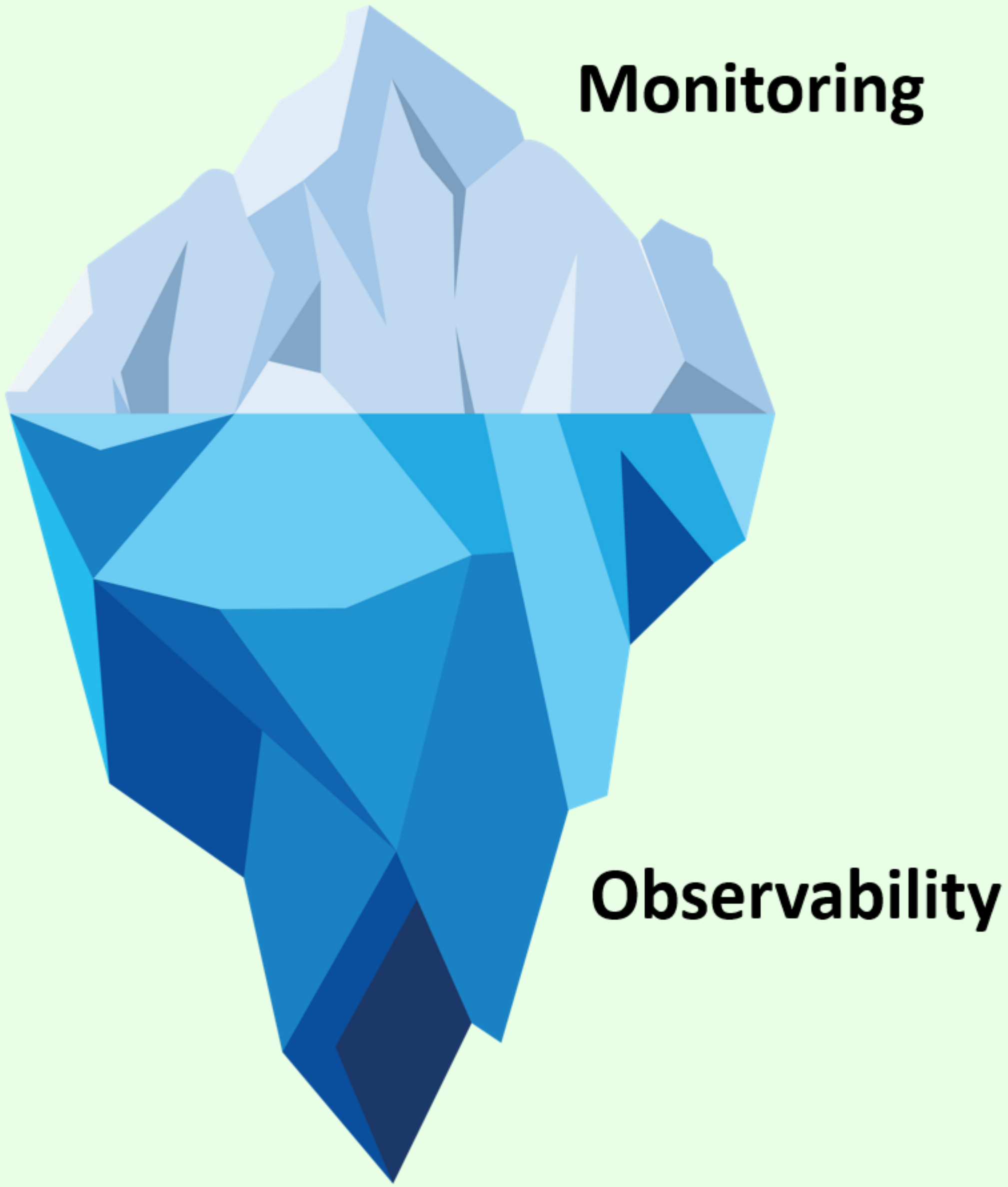
Monitoring in microservices is important because it allows you to:

- **Identify and troubleshoot problems:** **By collecting and analyzing data from your microservices, you can identify problems before they cause outages or other disruptions.**
- Track the health of your microservices: **Monitoring can help you to track the health of your microservices, so you can identify any microservices that are underperforming or that are experiencing problems.**
- **Optimize your microservices:** **By monitoring your microservices, you can identify areas where you can optimize your microservices to improve performance and reliability.**

Monitoring and observability can be considered as two sides of the same coin. Both rely on the same types of telemetry data to enable insight into software distributed systems. Those data types — metrics, traces, and logs — are often referred to as the three pillars of observability.

Observability vs. Monitoring

eazy bytes



Feature	Monitoring	Observability
Purpose	Identify and troubleshoot problems	Understand the internal state of a system
Data	Metrics, traces, and logs	Metrics, traces, logs, and other data sources
Goal	Identify problems	Understand how a system works
Approach	Reactive	Proactive

In other words, monitoring is about collecting data and observability is about understanding data.

Monitory is reacting to problems while observavility is fixing them in real time.

Logging

Logs are discrete records of events that happen in software applications over time. They contain a timestamp that indicates when the event happened, as well as information about the event and its context. This information can be used to answer questions like "What happened at this time?", "Which thread was processing the event?", or "Which user/tenant was in the context?"

Logs are essential tools for troubleshooting and debugging tasks. They can be used to reconstruct what happened at a specific point in time in a single application instance. Logs are typically categorized according to the type or severity of the event, such as trace, debug, info, warn, and error. This allows us to log only the most severe events in production, while still giving us the chance to change the log level temporarily during debugging.

Logging in Monolithic Apps

In monolithic apps, all of the code is in a single codebase. This means that all of the logs are also in a single location. This makes it easy to find and troubleshoot problems, as you only need to look in one place.

Logging in Microservices

Logging in microservices is complex. This is because each service has its own logs. This means that you need to look in multiple places to find all of the logs for a particular request.

To address this challenge, microservices architectures often use centralized logging. Centralized logging collects logs from all of the services in the architecture and stores them in a single location. This makes it easier to find and troubleshoot problems, as you only need to look in one place

Managing logs with Grafana, Loki & Promtail

Grafana is an open-source analytics and interactive visualization web application. It provides charts, graphs, and alerts for the web when connected to supported data sources. It can be easily installed using Docker or Docker Compose.

Grafana is a popular tool for visualizing metrics, logs, and traces from a variety of sources. It is used by organizations of all sizes to monitor their applications and infrastructure.

Grafana Loki is a horizontally scalable, highly available, and cost-effective log aggregation system. It is designed to be easy to use and to scale to meet the needs of even the most demanding applications.

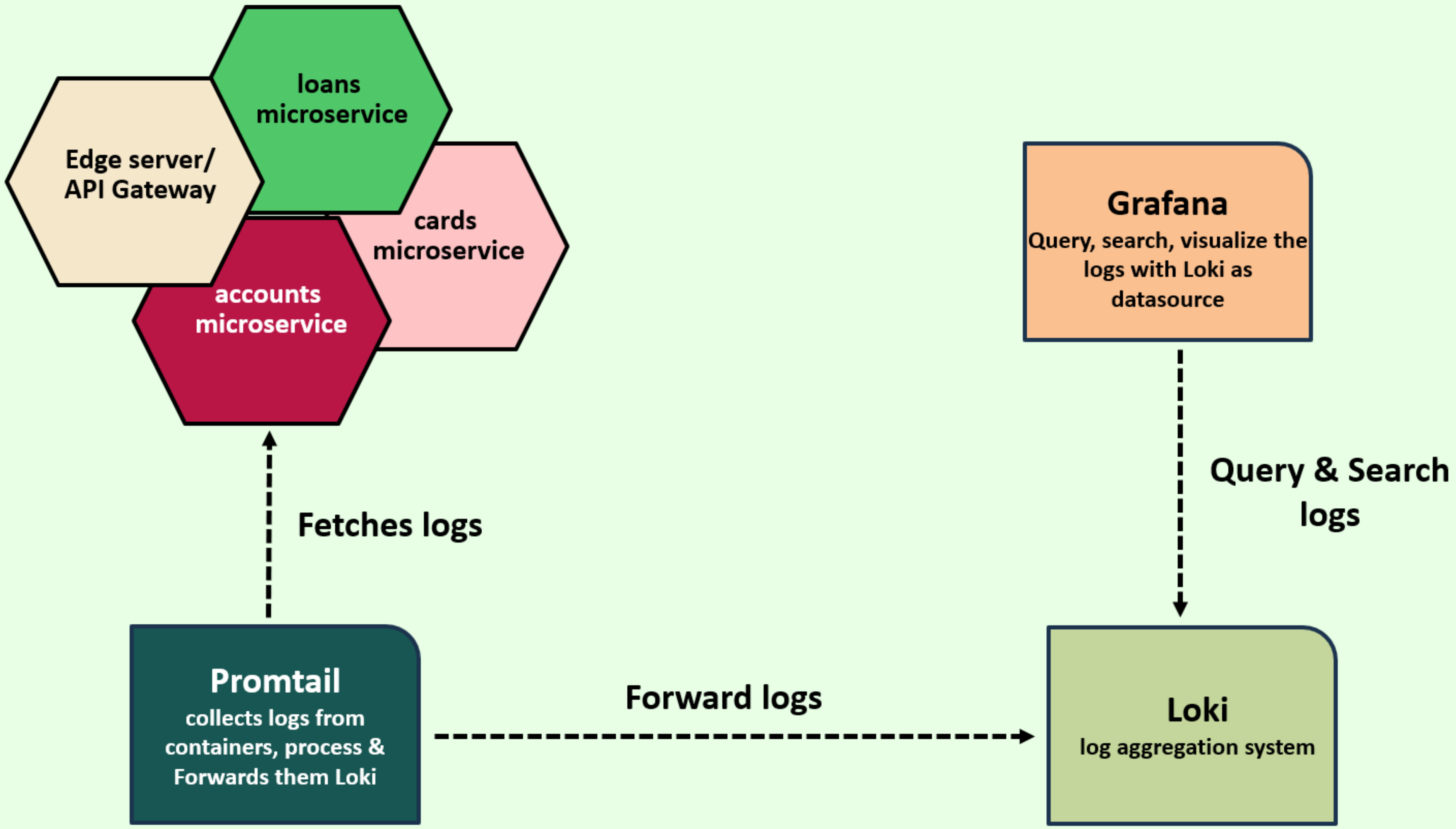
Promtail is a lightweight log agent that ships logs from your containers to Loki. It is easy to configure and can be used to collect logs from a wide variety of sources.

Together, Grafana Loki and Promtail provide a powerful logging solution that can help you to understand and troubleshoot your applications.

Grafana provides visualization of the log lines captured within Loki.

Managing logs with Grafana, Loki & Promtail

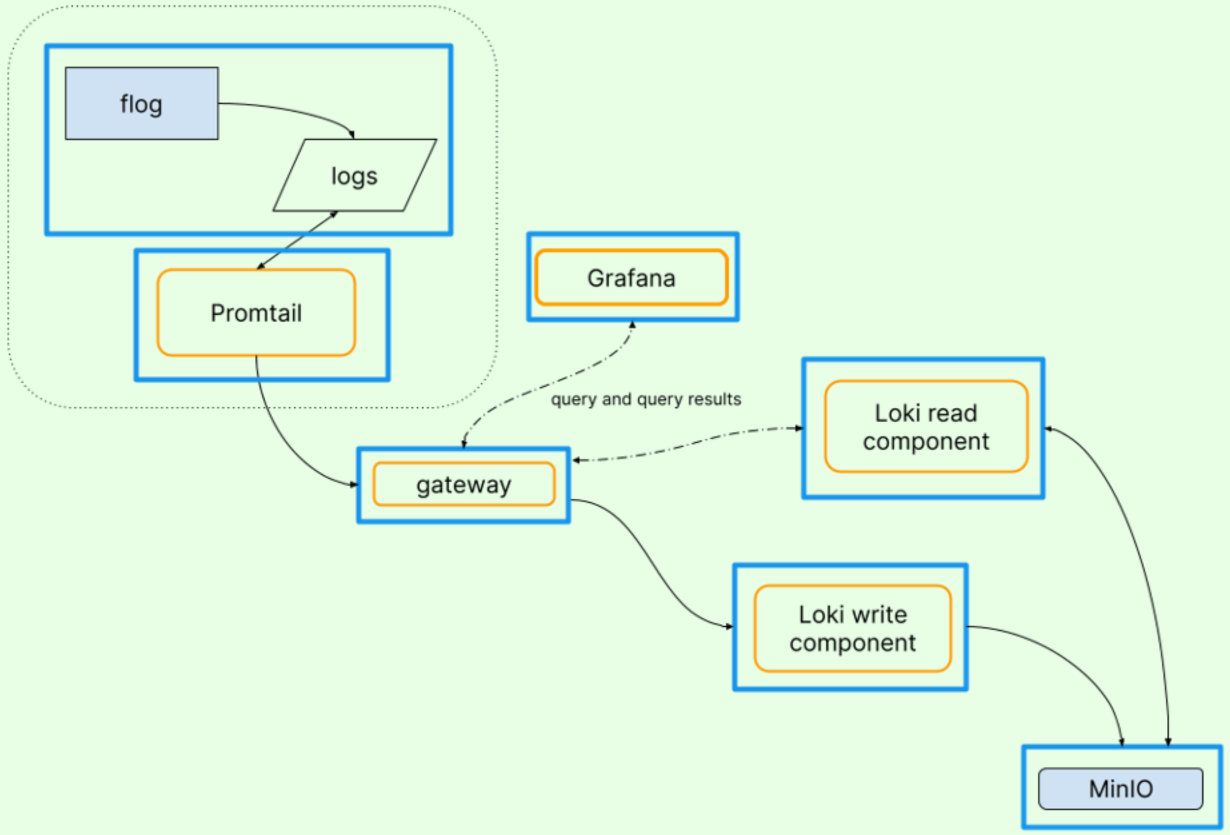
eazy bytes



Sample demo of logging using Grafana, Loki & promotail

eazy bytes

Reference: https://grafana.com/docs/loki/latest/getting-started/



cloud-native applications generate logs as events and send them to the standard output, without being concerned about the processing or storage of those logs.

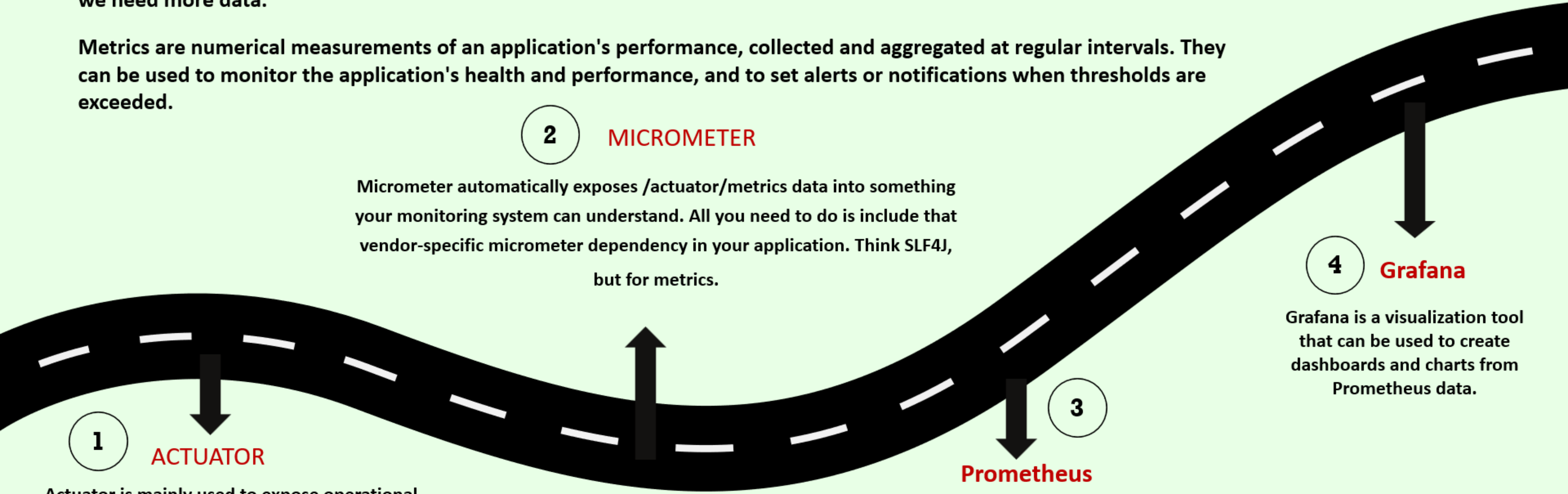
One advantage of treating logs as event streams and emitting them to stdout is that it decouples the application from the log processing infrastructure. The application can focus on its core functionality without being tied to a specific logging implementation or storage solution. The infrastructure, on the other hand, can handle the collection, aggregation, and storage of logs using appropriate tools and services.

15-Factor methodology recommends the same to treat logs as events streamed to the standard output and not concern with how they are processed or stored.

Metrics & monitoring with Spring Boot Actuator, Micrometer, Prometheus & Grafana

Event logs are essential for monitoring applications, but they don't provide enough data to answer all of the questions we need to know. To answer questions like CPU usage, memory usage, threads usage, error requests etc. & properly monitor, manage, and troubleshoot an application in production, we need more data.

Metrics are numerical measurements of an application's performance, collected and aggregated at regular intervals. They can be used to monitor the application's health and performance, and to set alerts or notifications when thresholds are exceeded.



1 ACTUATOR

Actuator is mainly used to expose operational information about the running application — health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.

2 MICROMETER

Micrometer automatically exposes /actuator/metrics data into something your monitoring system can understand. All you need to do is include that vendor-specific micrometer dependency in your application. Think SLF4J, but for metrics.

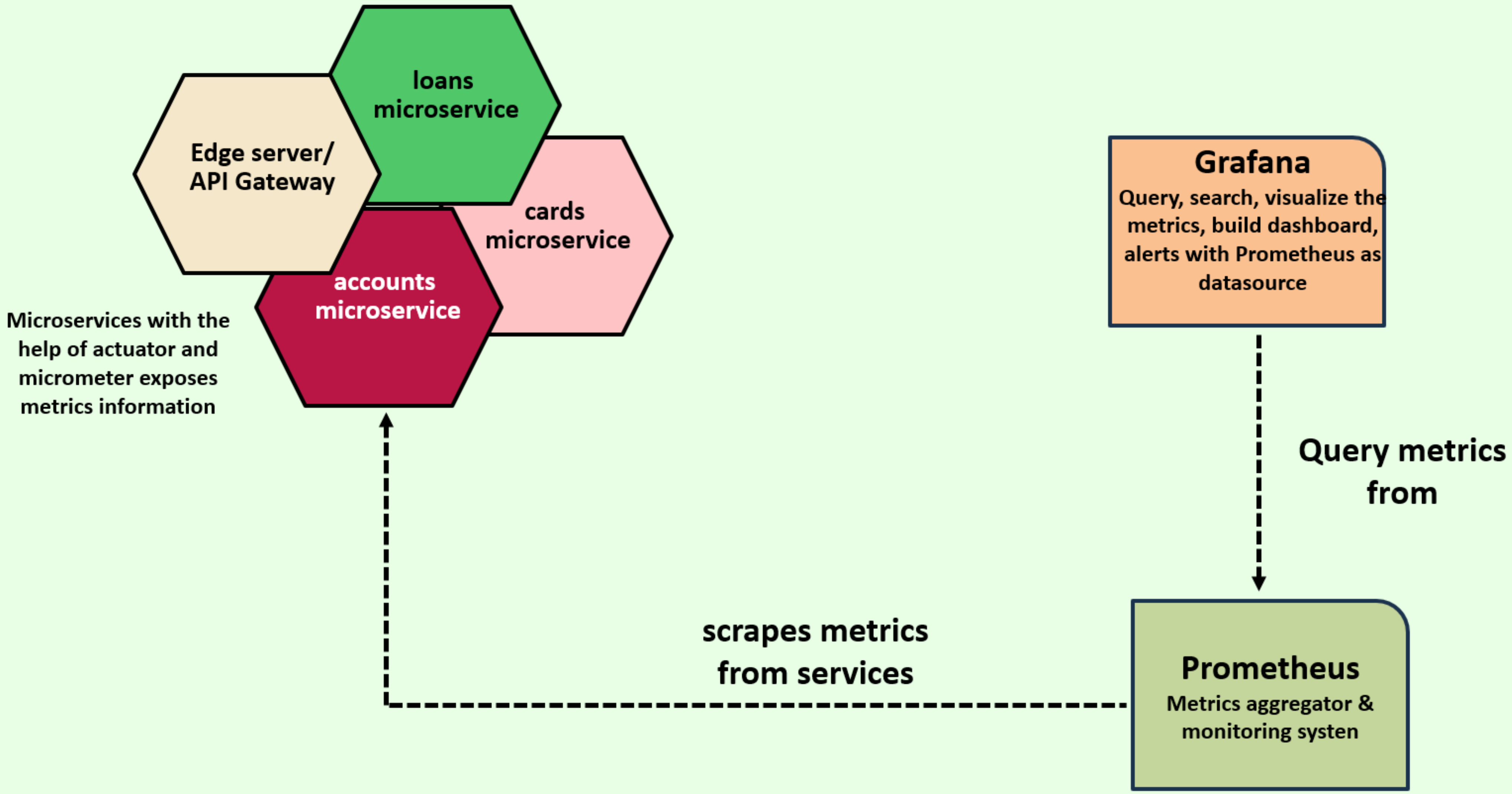
3 Prometheus

The most common format for exporting metrics is the one used by Prometheus, which is "an open-source systems monitoring and alerting toolkit". Just as Loki aggregates and stores event logs, Prometheus does the same with metrics.

4 Grafana

Grafana is a visualization tool that can be used to create dashboards and charts from Prometheus data.

Metrics & monitoring with Spring Boot Actuator, Micrometer, Prometheus & Grafana


loans microservice
Edge server/ API Gateway
cards microservice
accounts microservice
Microservices with the help of actuator and micrometer exposes metrics information
Grafana
Query, search, visualize the metrics, build dashboard, alerts with Prometheus as datasource
Query metrics from
scrapes metrics from services
Prometheus
Metrics aggregator & monitoring systen

Distributed tracing in microservices

Event logs, health probes, and metrics offer a wealth of valuable information for deducing the internal condition of an application. Nevertheless, these sources fail to account for the distributed nature of cloud-native applications. Given that a user request often traverses multiple applications, we currently lack the means to effectively correlate data across application boundaries.

Distributed tracing is a technique used in microservices or cloud-native applications to understand and analyze the flow of requests as they propagate across multiple services and components. It helps in gaining insights into how requests are processed, identifying performance bottlenecks, and diagnosing issues in complex, distributed systems.

One possible solution to address this issue is to implement a straightforward approach where a unique identifier, known as a correlation ID, is generated for each request at the entry point of the system. This correlation ID can then be utilized in event logs and passed along to other relevant services involved in processing the request. By leveraging this correlation ID, we can retrieve all log messages associated with a specific transaction from multiple applications.

Distributed tracing encompasses three primary concepts:

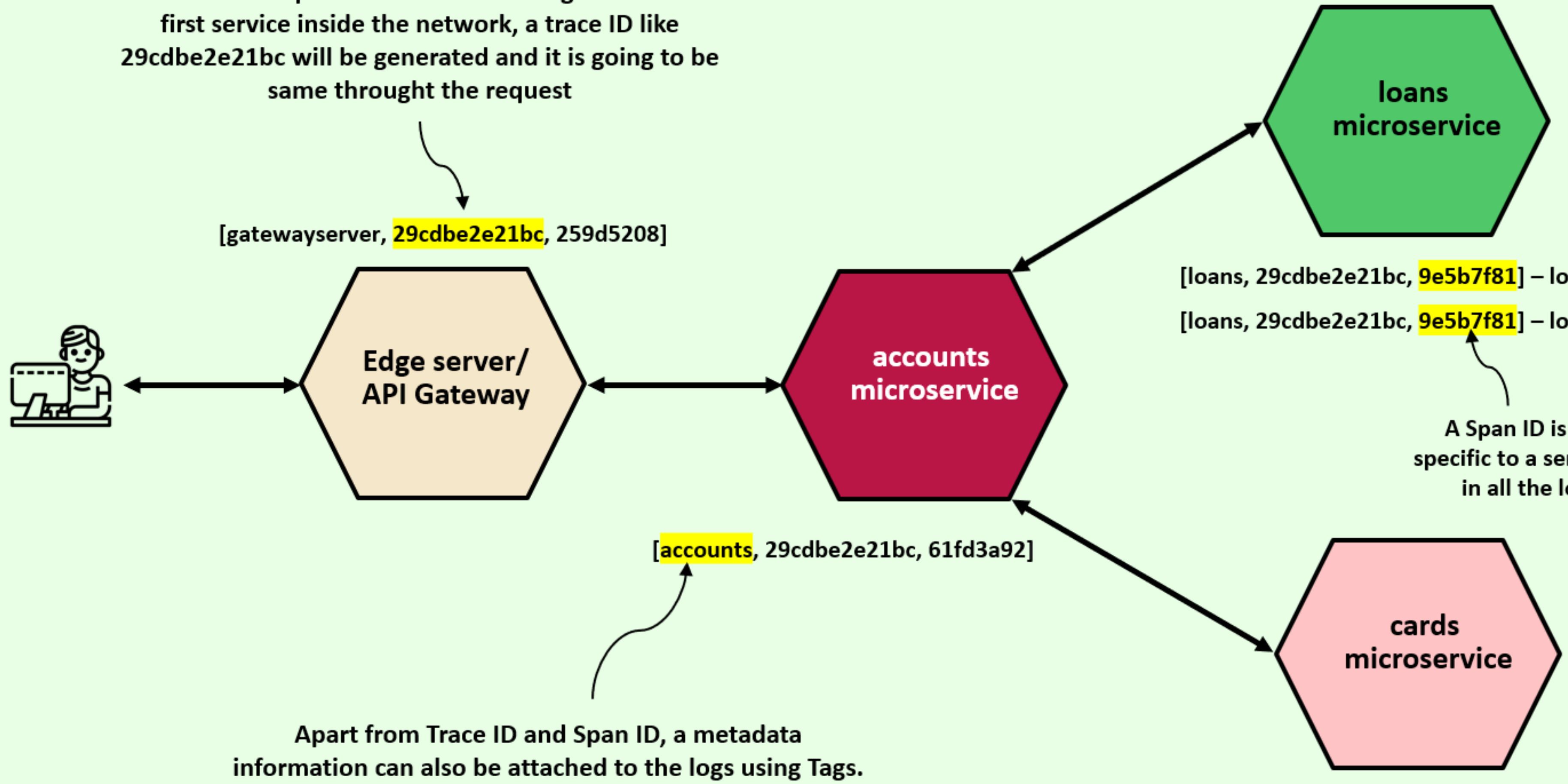
Tags serve as metadata that offer supplementary details about the span context, including the request URI, the username of the authenticated user, or the identifier for a specific tenant.

A trace denotes the collection of actions tied to a request or transaction, distinguished by a **trace ID**. It consists of multiple spans that span across various services.

A span represents each individual stage of request processing, encompassing start and end timestamps, and is uniquely identified by the combination of trace ID and span ID.

Distributed tracing in microservices

eazy bytes



When a client request received at the edge server or the first service inside the network, a trace ID like 29cdbe2e21bc will be generated and it is going to be same throught the request

[gatewayserver, 29cdbe2e21bc, 259d5208]

[loans, 29cdbe2e21bc, 9e5b7f81] – logger statement 1

[loans, 29cdbe2e21bc, 9e5b7f81] – logger statement 2

A Span ID is going to be generated which is specific to a service and the same will be present in all the logs of the service in a request

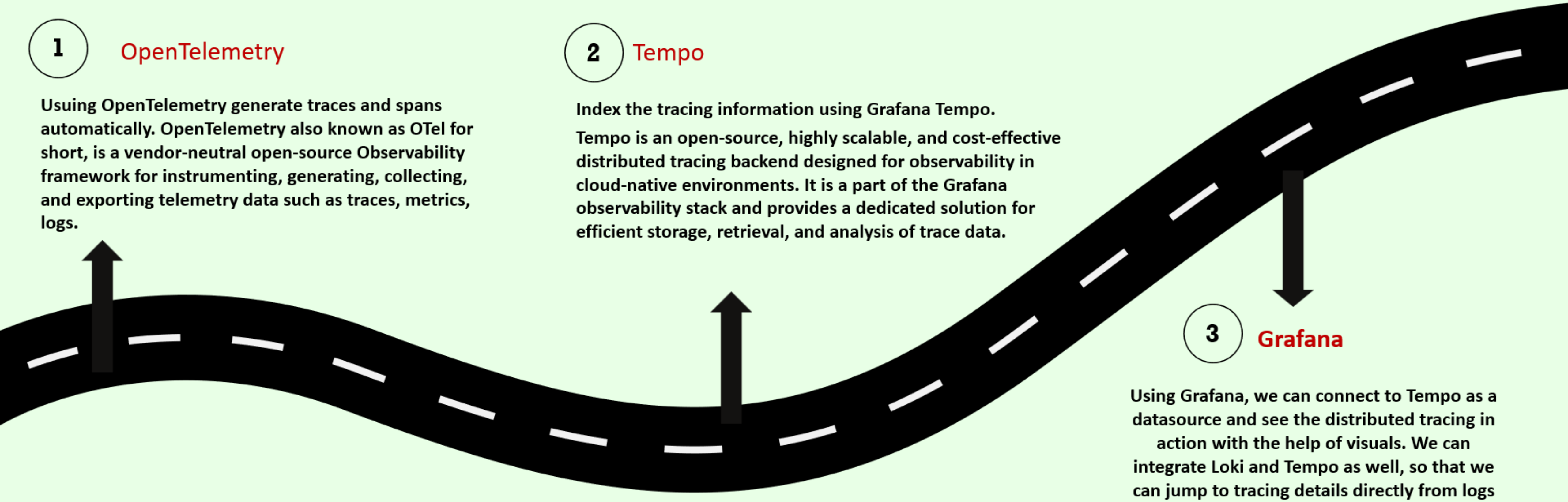
[accounts, 29cdbe2e21bc, 61fd3a92]

Apart from Trace ID and Span ID, a metadata information can also be attached to the logs using Tags. For example, here we added serive name which is accounts

[cards, 29cdbe2e21bc, 8f4b7a76] – logger statement 1

[cards, 29cdbe2e21bc, 8f4b7a76] – logger statement 2

Distributed tracing with OpenTelemetry, Tempo & Grafana



1 OpenTelemetry

Usuing OpenTelemetry generate traces and spans automatically. OpenTelemetry also known as OTel for short, is a vendor-neutral open-source Observability framework for instrumenting, generating, collecting, and exporting telemetry data such as traces, metrics, logs.

2 Tempo

Index the tracing information using Grafana Tempo.

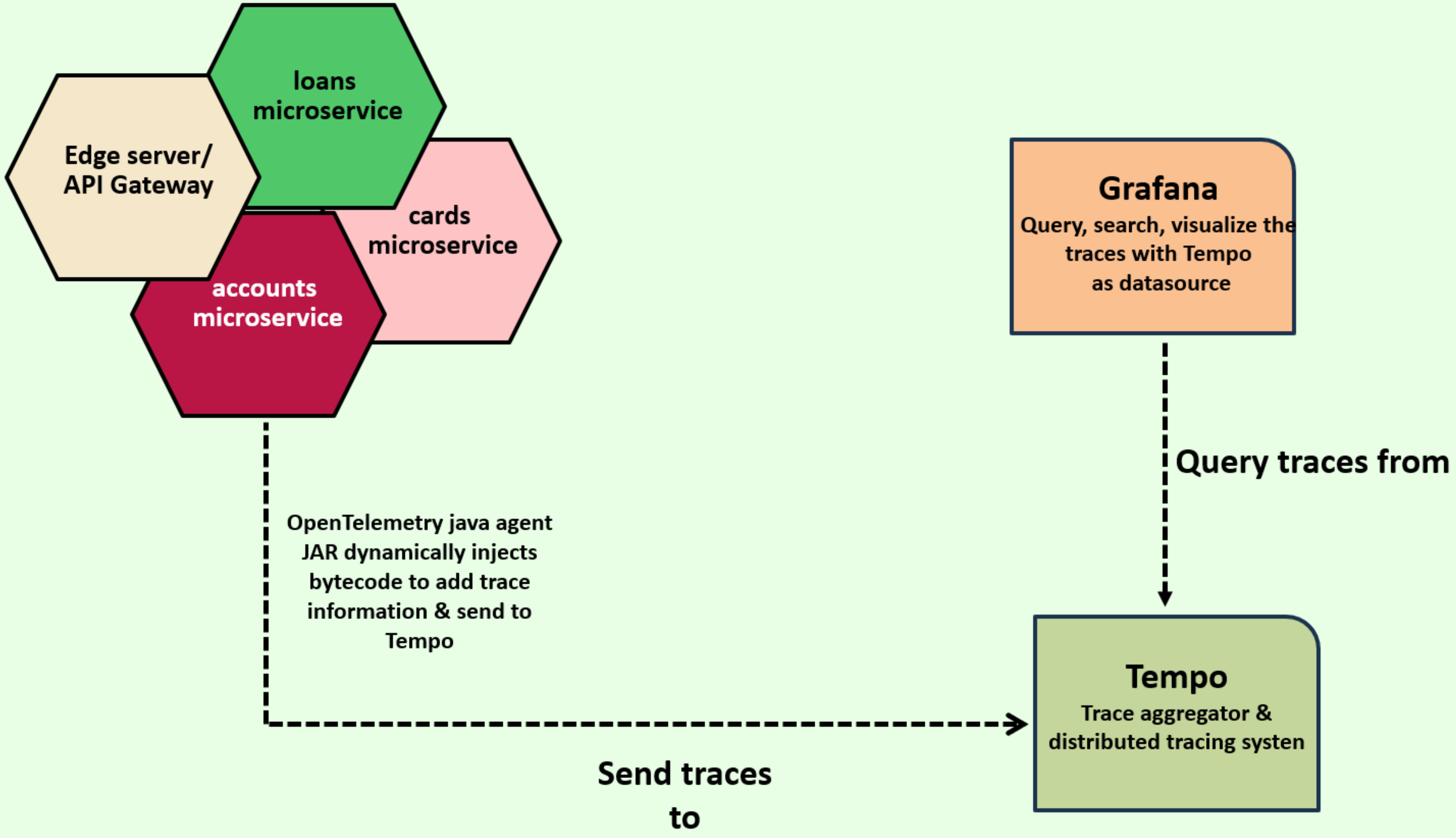
Tempo is an open-source, highly scalable, and cost-effective distributed tracing backend designed for observability in cloud-native environments. It is a part of the Grafana observability stack and provides a dedicated solution for efficient storage, retrieval, and analysis of trace data.

3 Grafana

Using Grafana, we can connect to Tempo as a datasource and see the distributed tracing in action with the help of visuals. We can integrate Loki and Tempo as well, so that we can jump to tracing details directly from logs inside Loki

Distributed tracing with OpenTelemetry, Tempo & Grafana

eazy bytes



Edge server/ API Gateway
loans microservice
cards microservice
accounts microservice
OpenTelemetry java agent JAR dynamically injects bytecode to add trace information & send to Tempo
Send traces to
Grafana
Query, search, visualize the traces with Tempo as datasource
Query traces from
Tempo
Trace aggregator & distributed tracing systen