Groundbreakers

12th Oracle Developer Meetup

Distributed Tracing : Jaeger를 활용한 분산 Tracing

강인호
inho.kang@oracle.com

2019.06.16

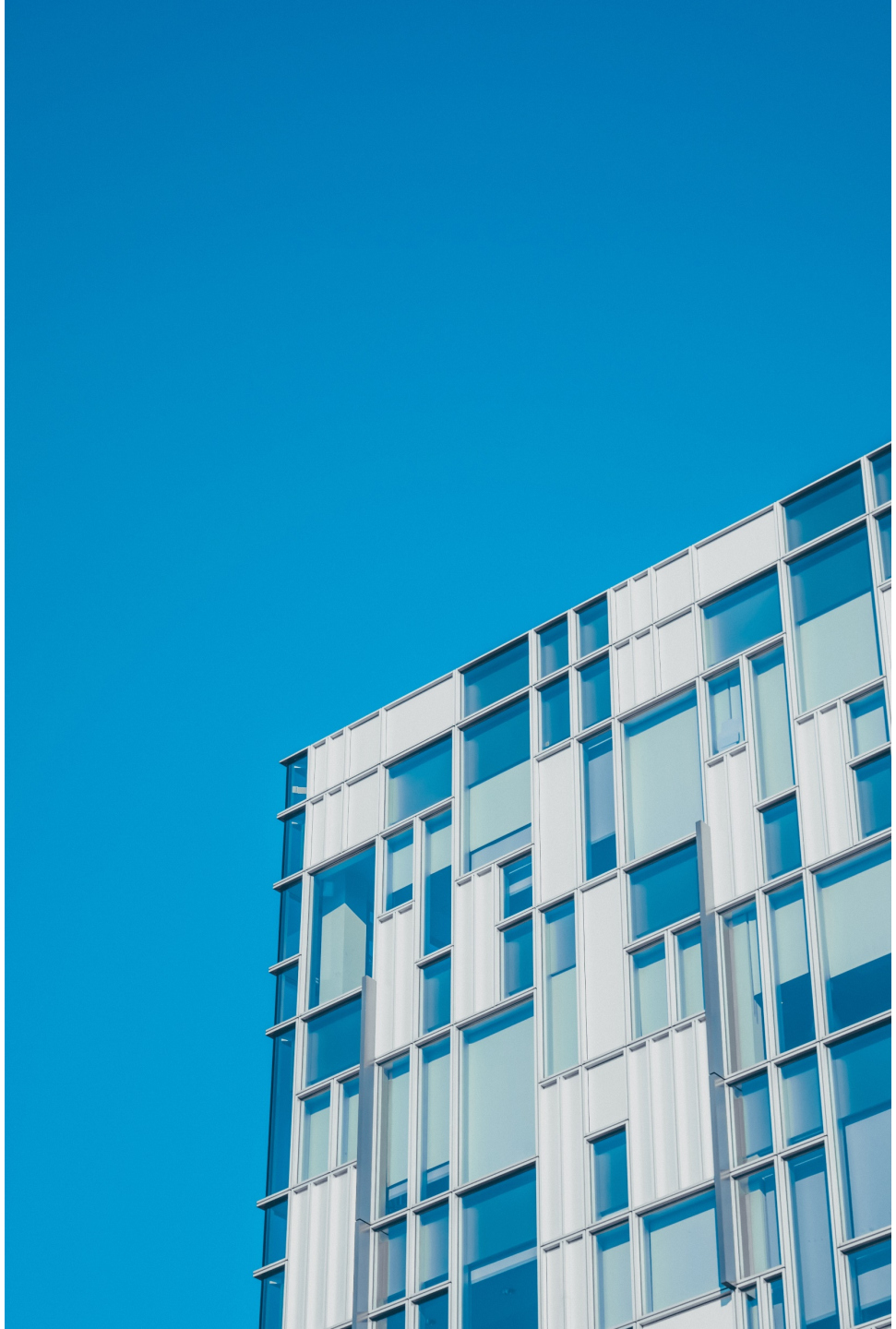
ORACLE

Kang In Ho

- .Net Developer
- CBD, SOA Methodology Consulting
- ITA/EA, ISP Consulting
- Oracle Corp.
  - Middleware
  - Cloud Native Application, Container Native
  - Emerging Technology Team
- k8s korea user group

innoshom@gamil.com

A Table of Contents

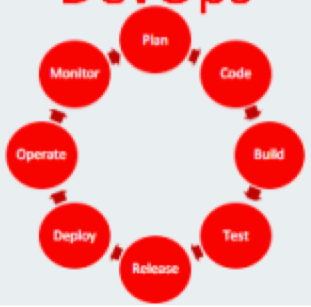
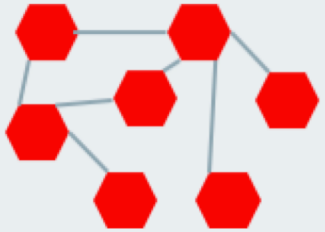
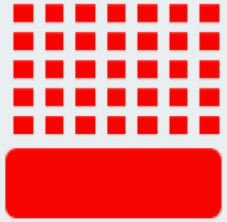


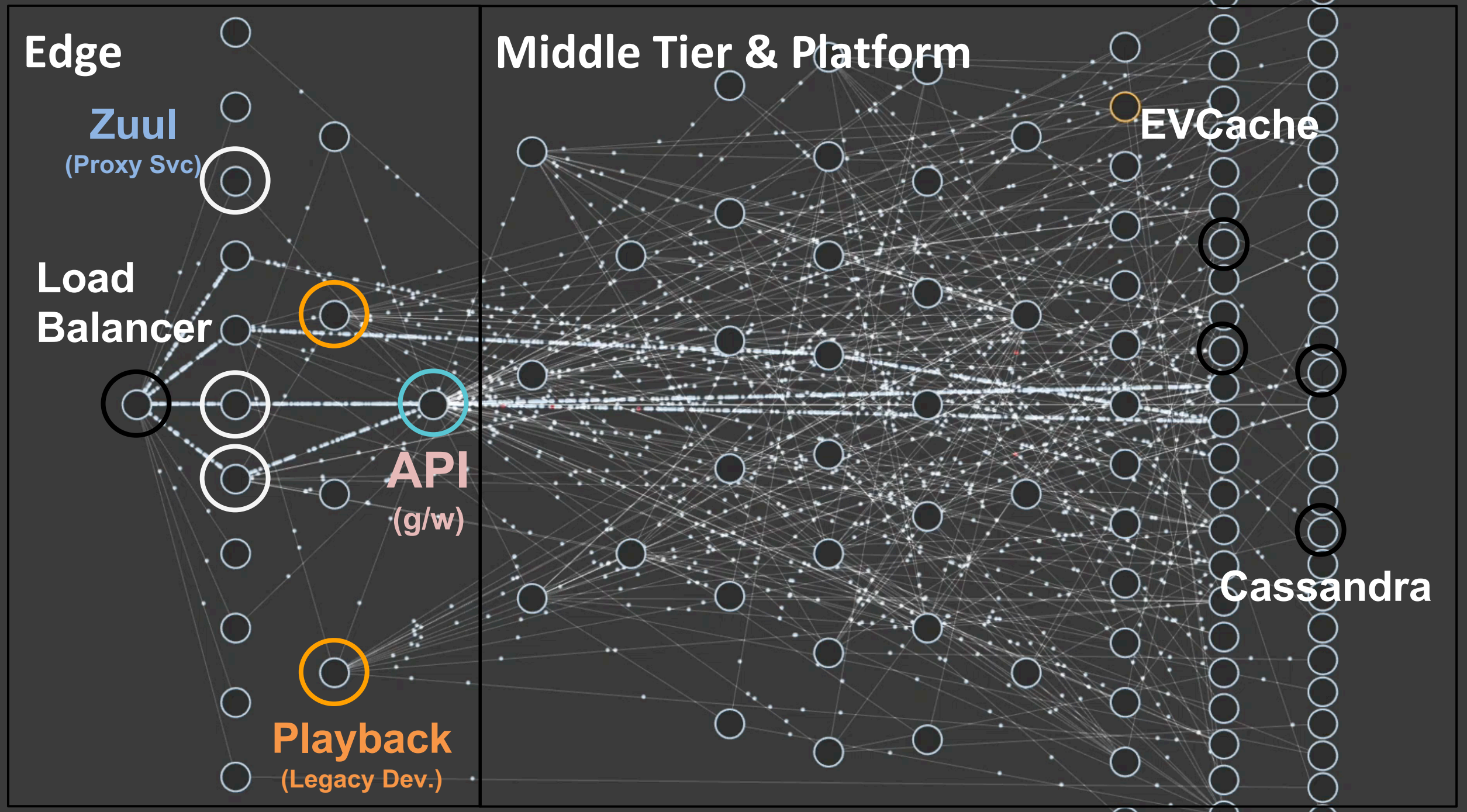
PART 1

Why Distributed Tracing

History and Multi-Dimensional Evolution of Computing

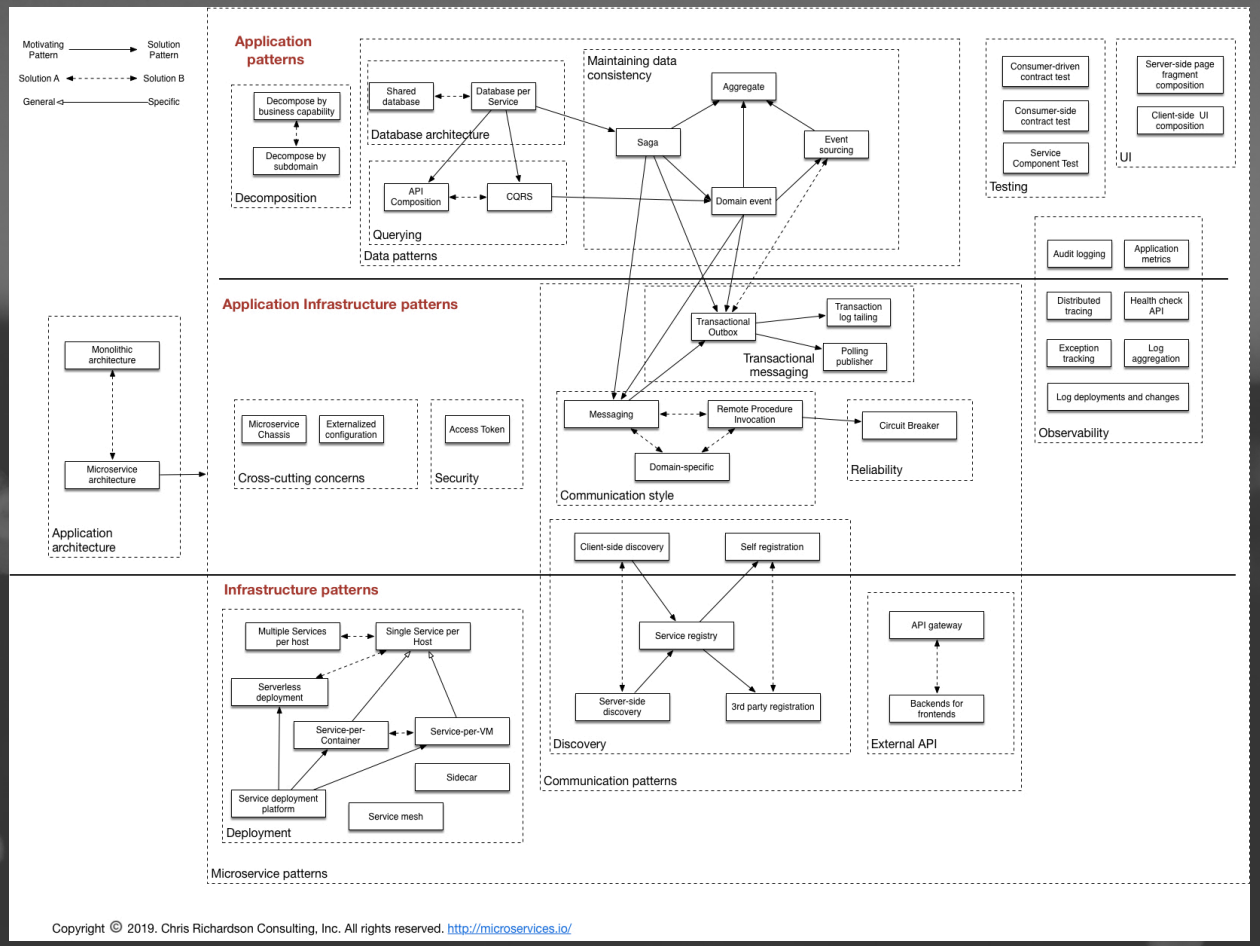
아키텍처의 변화 방향

Development Process	Application Architecture	Deployment and Packaging	Application Infrastructure
Waterfall 	Monolithic 	Physical Server 	Datacenter 
Agile 	N-Tier 	Virtual Servers 	Hosted 
DevOps 	Microservices 	Containers 	Cloud 


Edge
Middle Tier & Platform
Zuul
(Proxy Svc)
EVCache
Load
Balancer
API
(g/w)
Cassandra
Playback
(Legacy Dev.)

4 Architecture Patterns relationship

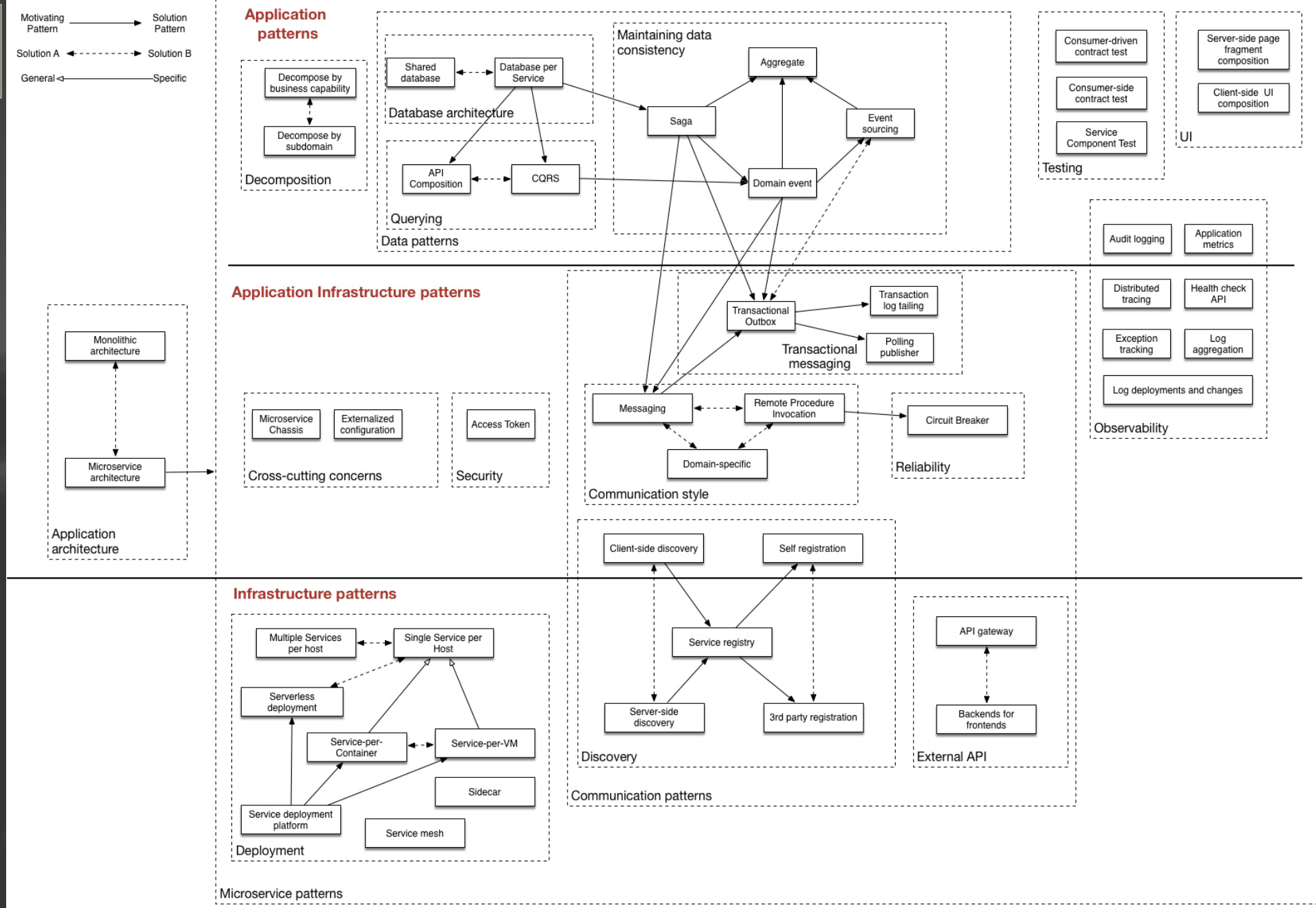
- Patterns of Microserivce



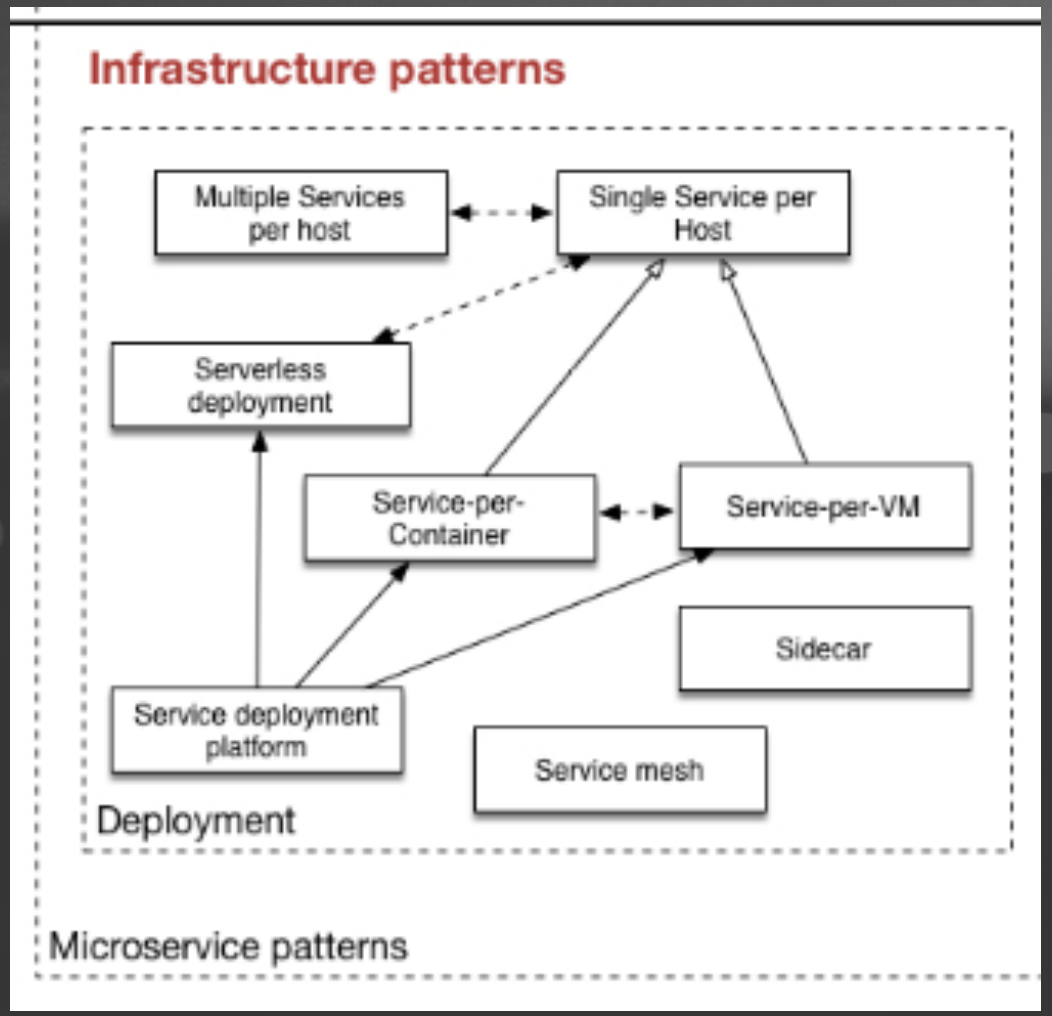
http://microservices.io

Reference: http://microservices.io

1

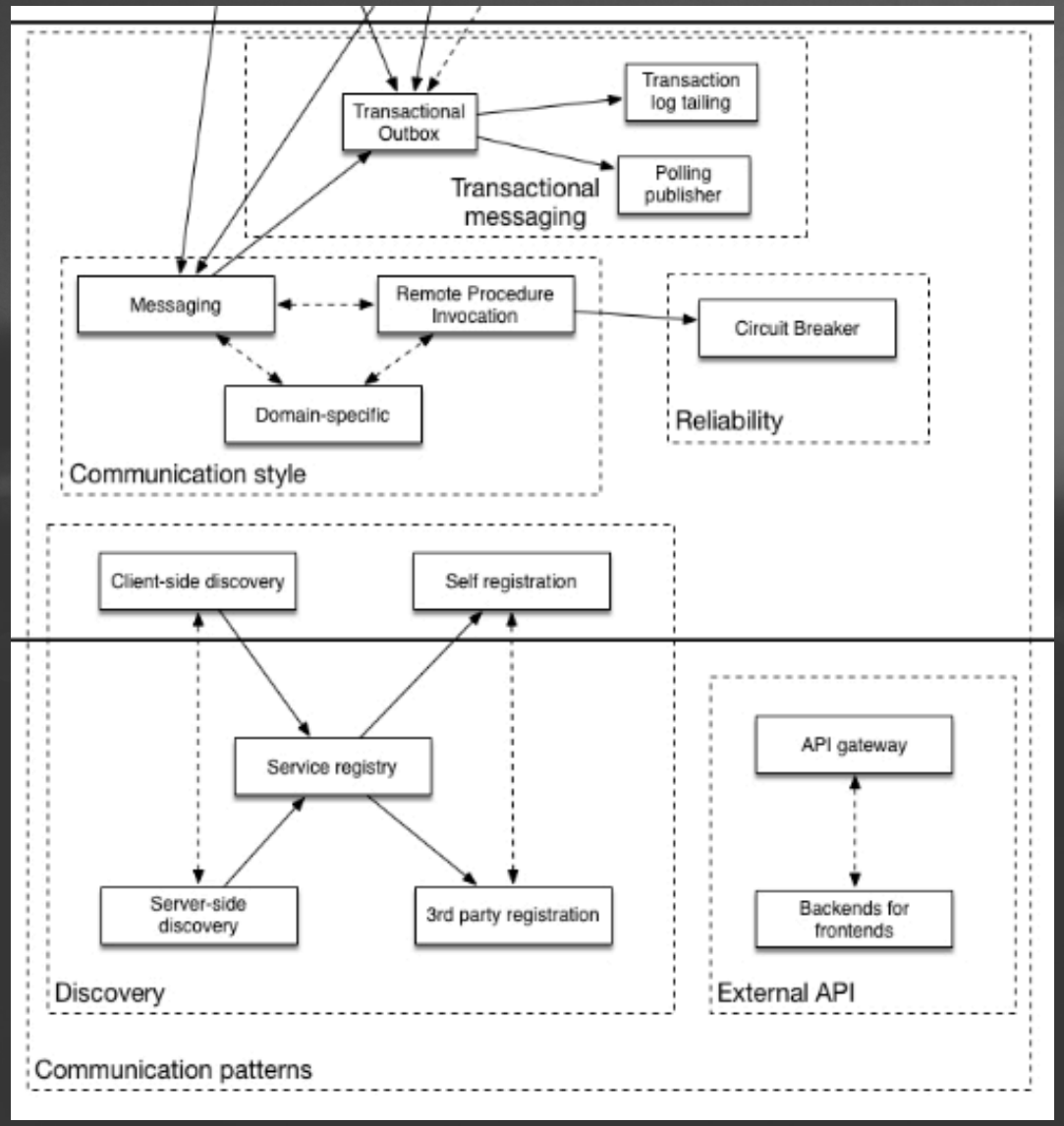


2 Infrastructure Patterns - Deployment



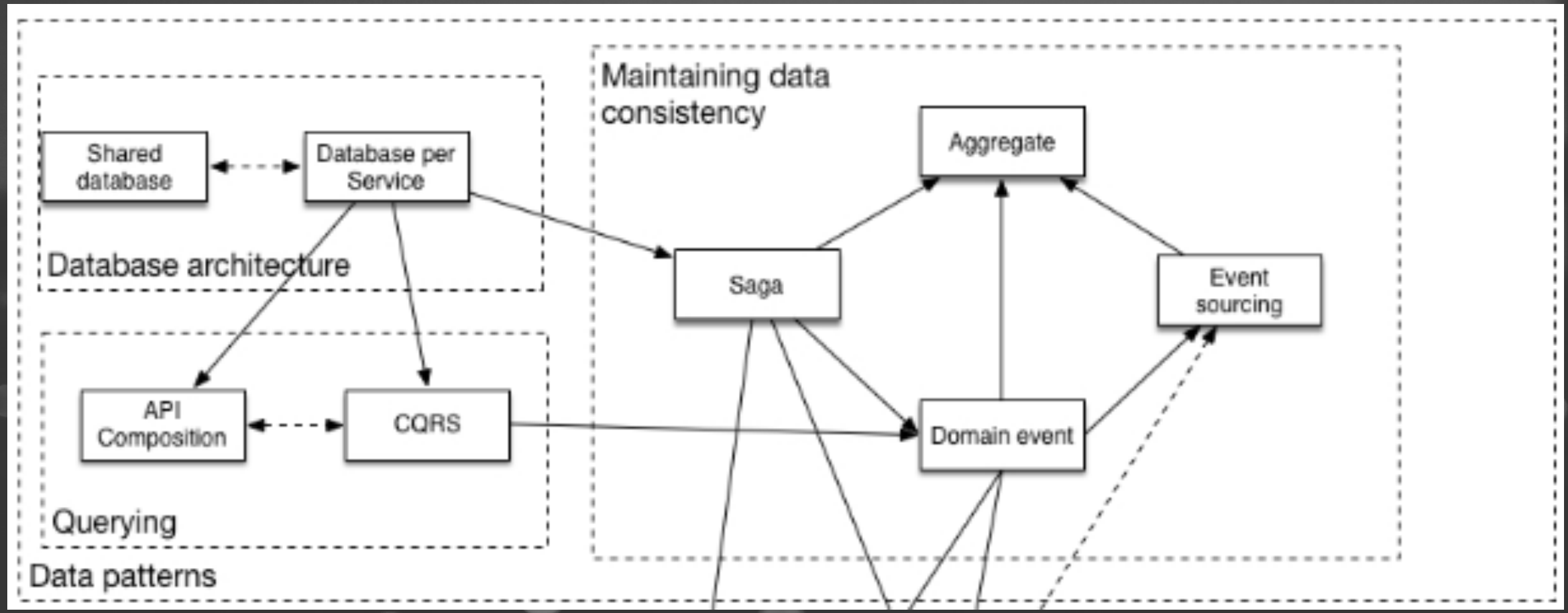
- Single Serviece per Host
- Multiple Services per Host
- Serverless deployment
  - Oracle Fx
  - AWS Lamda
- Service-per-Container
- Service-per-VM
- Service Deployment Platform
  - Docker orchestration
    (ex: k8s, Docker Swarm mode)

3 Communication Patterns

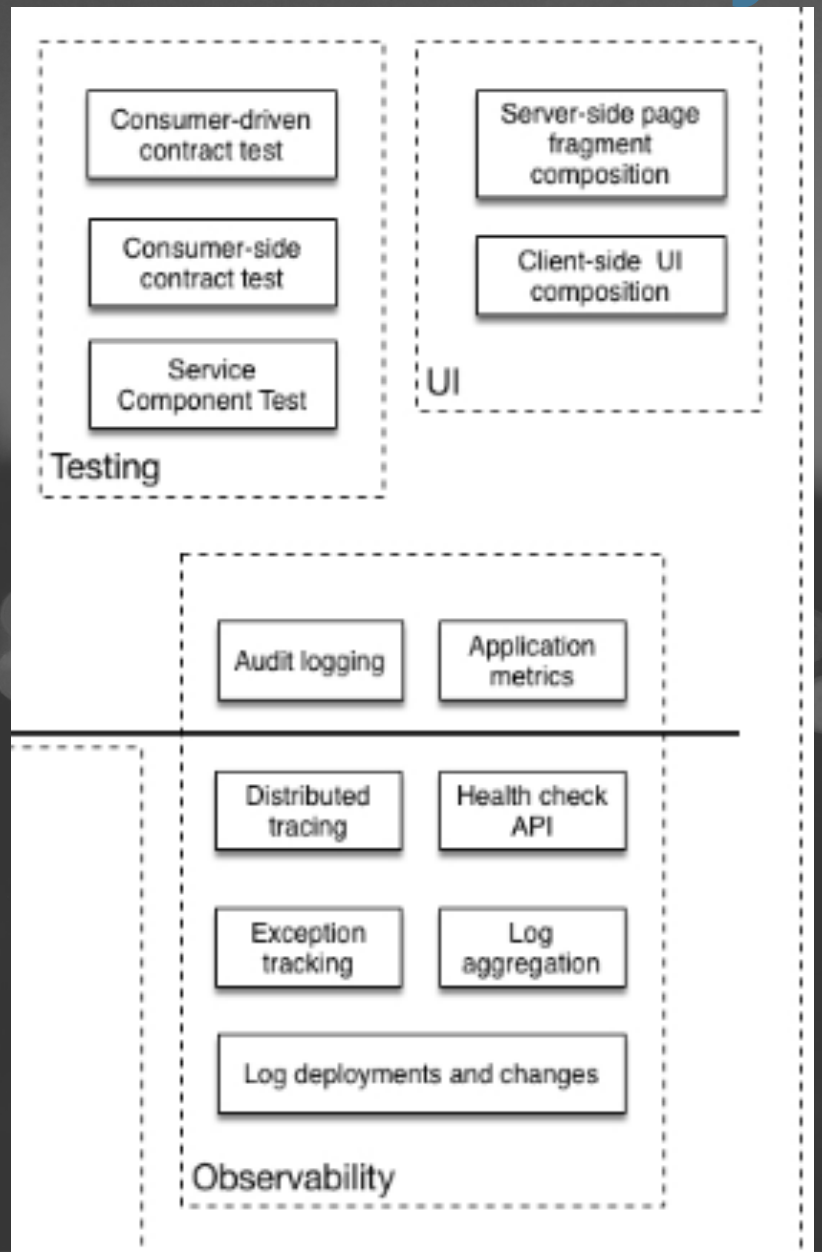


- Discovery
  - Service Registry
  - Server-side discovery
  - Client-side discovery
  - Self Registration
  - 3rd Party Registration
- External API
  - API Gateway
  - Backend for front end
- Communication Style
- Reliability

8 Data Patterns



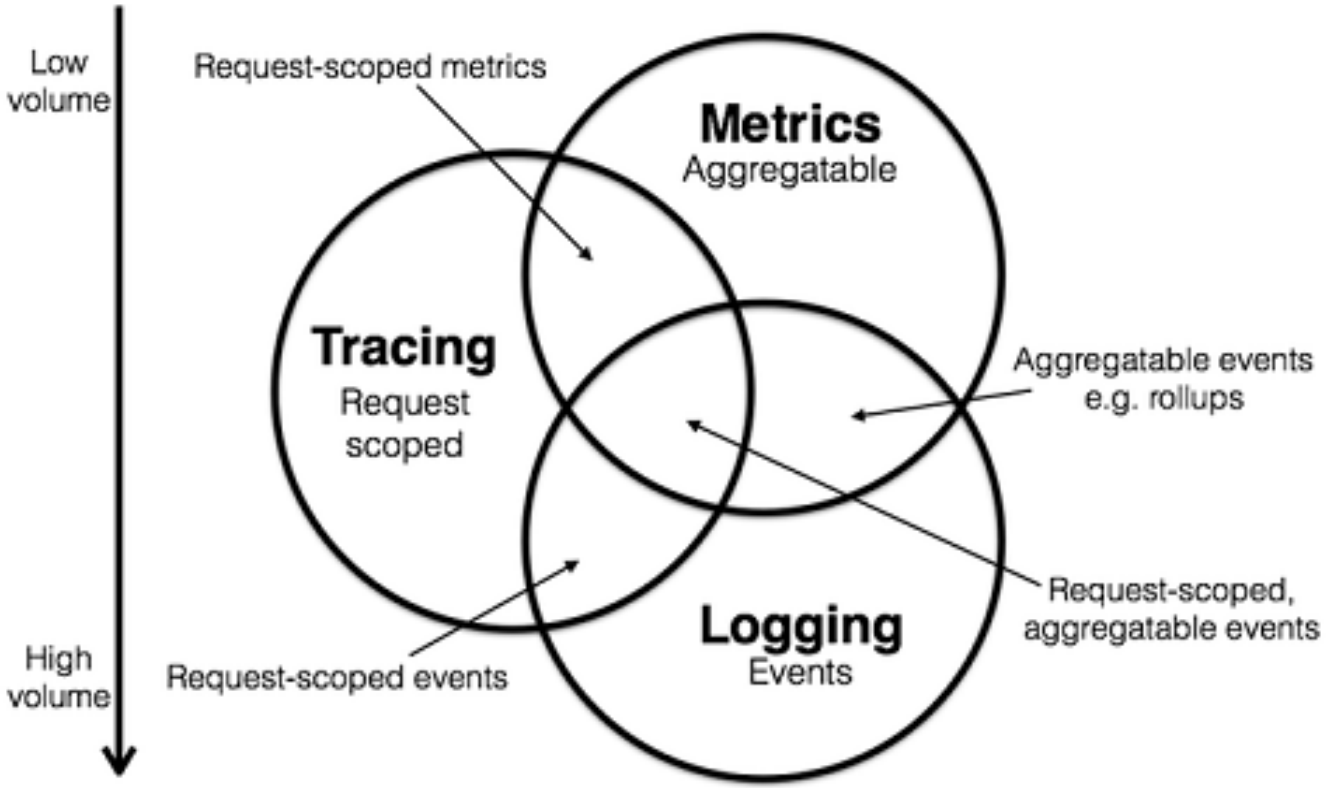
Testing & Observability




How do we know
What’s going on?

Observability

3가지 구성요소



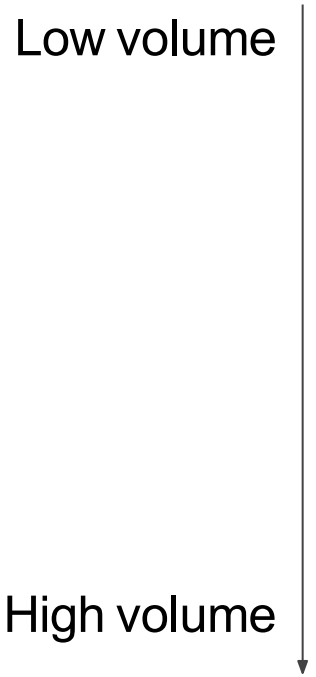
- **Logging is used to record discrete events.**
  For example, the debugging or error information of an application, which is the basis of problem diagnosis.

- **Metrics is used to record data that can be aggregated.**
  For example, the current depth of a queue can be defined as a metric and updated when an element is added to or removed from the queue. The number of HTTP requests can be defined as a counter that accumulates the number when new requests are received.

- **Tracing is used to record information within the request scope.**
  For example, the process and consumed time for a remote method call, This is the tool we use to investigate system performance issues. Logging, metrics, and tracing have overlapping parts as follows.

https://www.alibabacloud.com/help/doc-detail/68035.htm

Monitoring == Observing Events



Metrics - Record events as **aggregates** (e.g. counters)

Tracing - Record **transaction-scoped** events

Logging - Record **unique** events

Observability

3가지 구성요소

The term "observability" in control theory states that the system is observable if the internal states of the system and, accordingly, its behavior, can be determined by only looking at its inputs and outputs

The pillars of observability

Joyent

- Much has been made of the so-called “pillars of observability”: monitoring, logging and instrumentation
- Each of these is important, for each has within it the capacity to answer questions about the system
- But each also has limitations!
- Their shared limitation: each can only be as effective as the observer — they cannot answer questions not asked!
- Observability seeks to answer questions asked and prompt new ones: **the human is the foundation of observability**!

2018 Observability Practitioners Summit, Bryan Cantrill, the CTO of Joyent

전통적인 모니터링 방법들

기존의 Metrics나 Logs 방법으로는 분산환경에 대응 불가

Metrics and Logs are

- Per Instance
- Lack of context

Metrics : 메트릭은 숫자 정보(count, usage)만 모으는 것이므로 자원을 많이 잡아 먹지도 않는다.

Logs : 로그는 메트릭 보다는 더 Observability가 좋은 툴이다. 하지만 debugging 툴로는 한계가 있다.

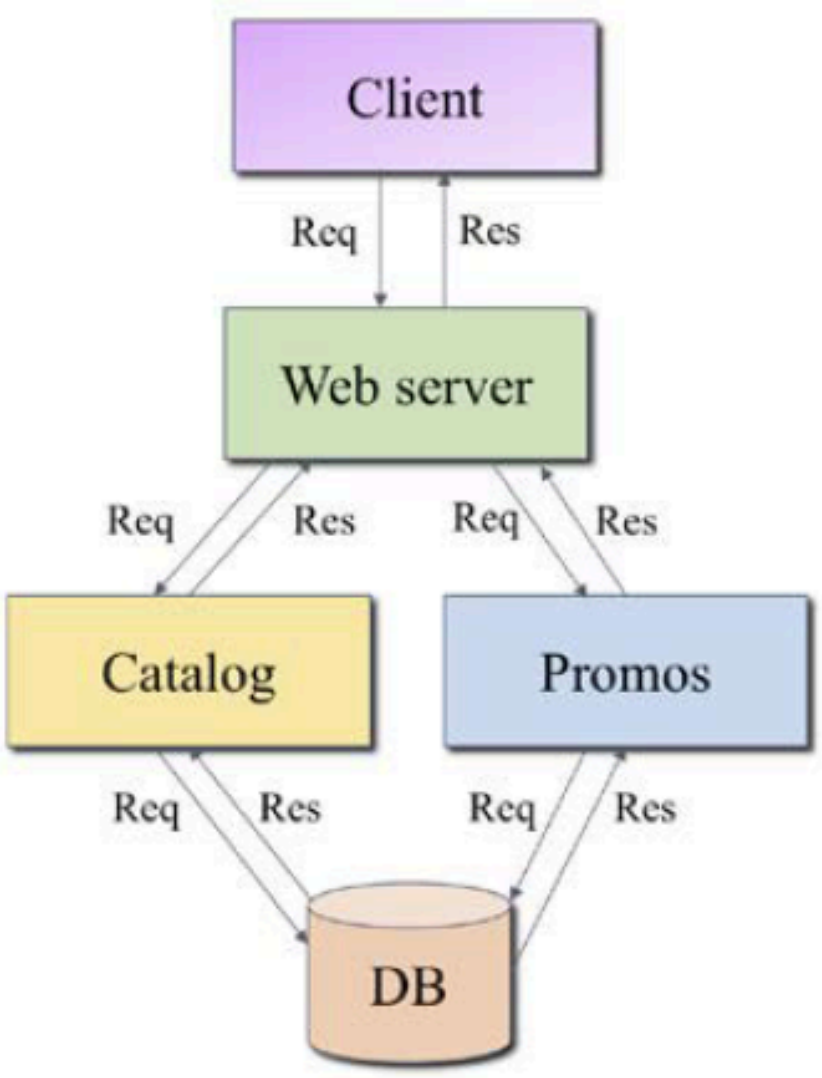
분산환경에서 Tracing을 한다는 것은?

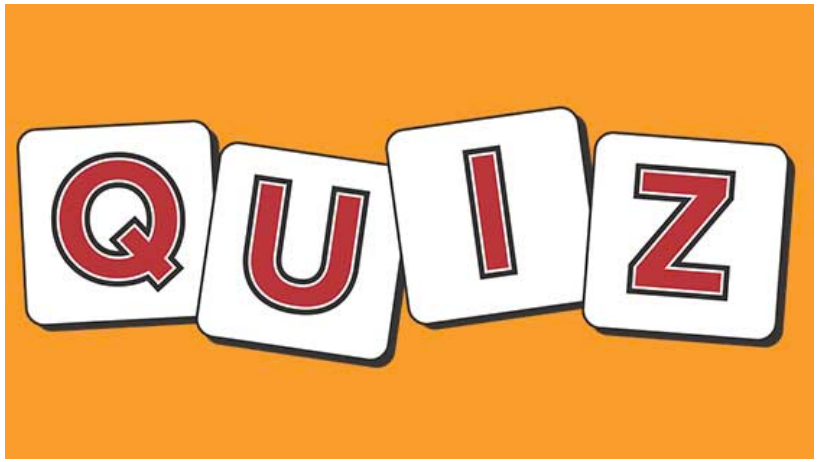
폴리그랏 환경에서 Request Path를 추적해서

- 어느 구간에서 병목이 발생하고
- Request가 실패하면 어디에서 Error가 발생했는지?
- Critical Pah가 뭔지를 밝히는 작업

그러나 전통적인 모니터링 툴들은 아직 이것에 적합하지 못하다.

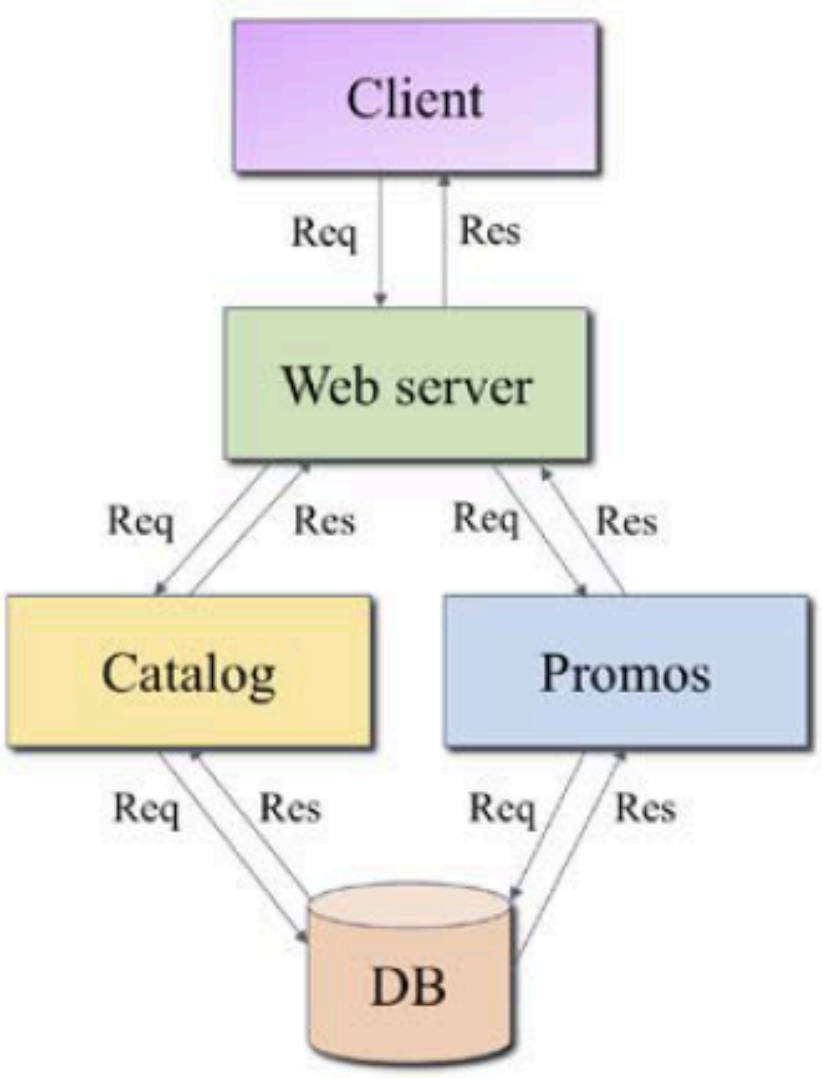
Basic Idea of Tracing

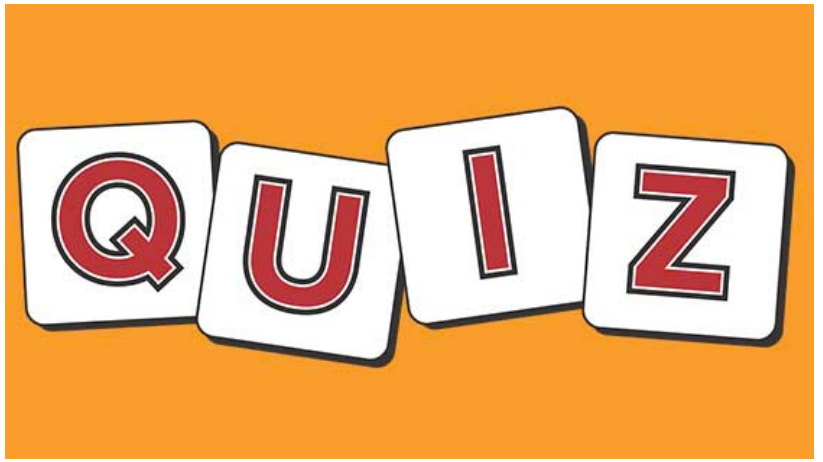




어떻게 하면 분산환경에서
Request Path를 추적할 수 있을까?

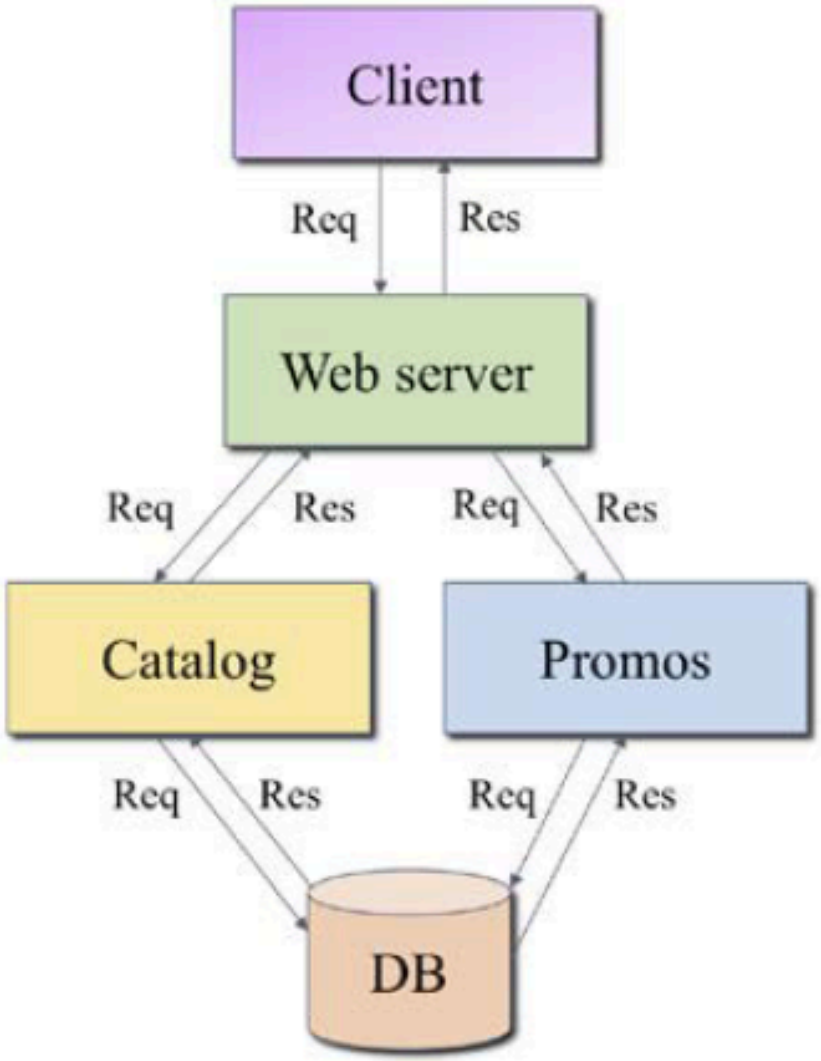
Basic Idea of Tracing

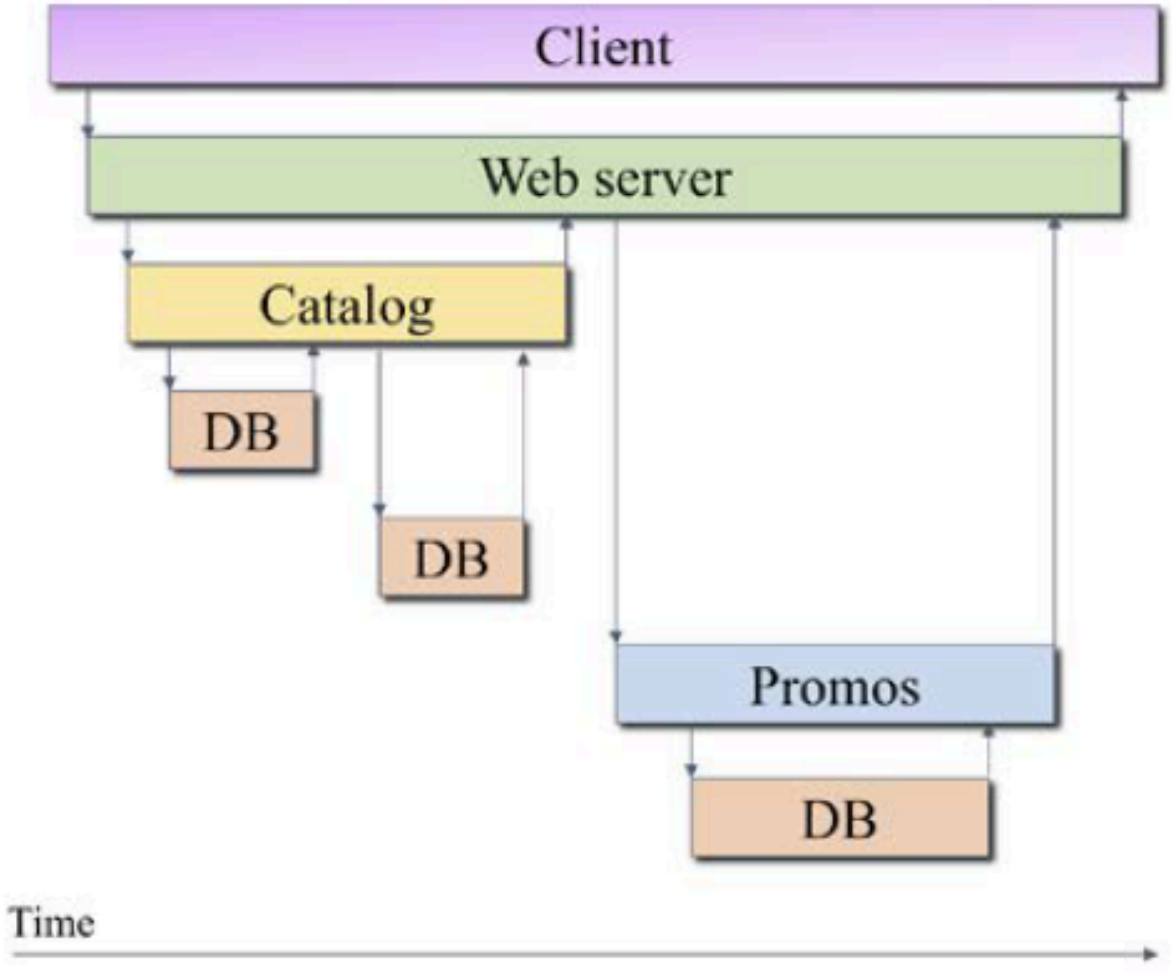




1. 각 단위, 단위의 프로세스 마다의 실행 시간과 정보들을 Profiling하고
2. 중앙에서 프로파일 정보를 모아서, Request 별로 연관관계를 재조합해서
3. Visualization Tool 또는 분석 하도록 한다.

Basic Idea of Tracing





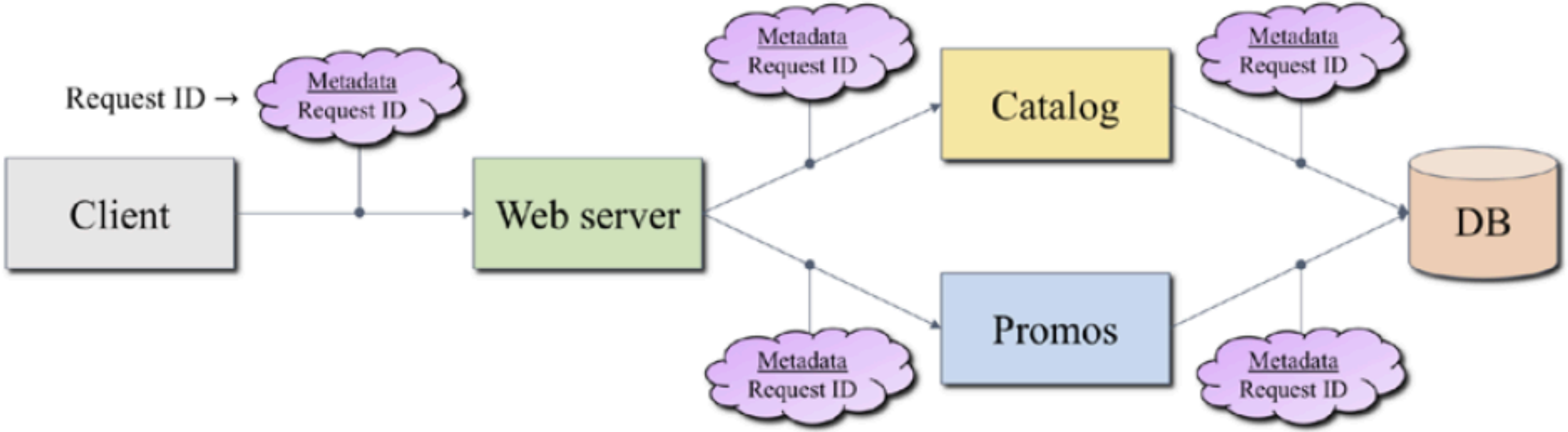
Request correlation

Schema Based

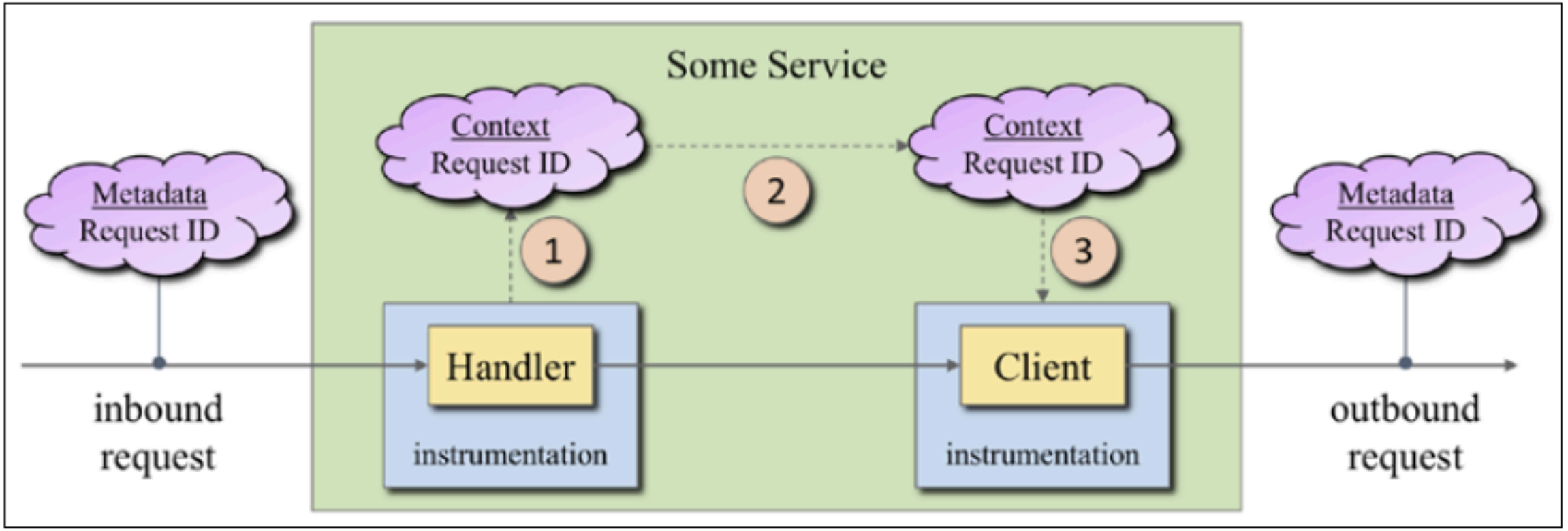
- Application 별로 Event Schema를 Manual 하게 정의해서 Log을 기반으로 연관성을 찾아내는 방법
- Modern Microservice 기반에는 한계

MetaData Propagation

- Global ID(Execution ID or Request ID)를 가지고 메타데이터를 전파하는 방식

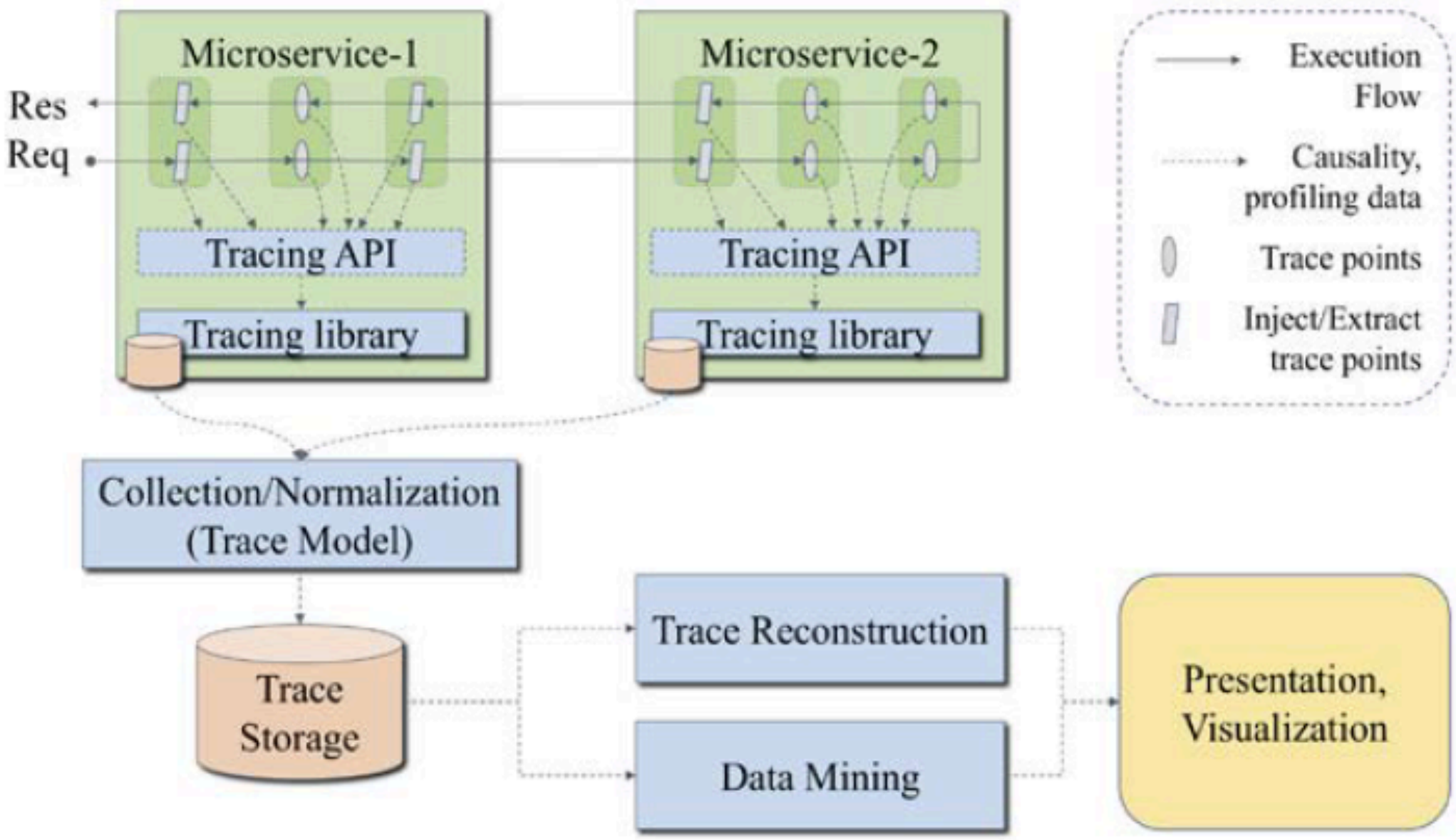


Inter-process Propagation



(1) The Handler that processes the inbound request is wrapped into instrumentation that **extracts metadata** from the request and stores it in a Context object in memory.
(2) Some in-process propagation mechanism, for example, based on thread-local variables.
(3) Instrumentation wraps an RPC client and **injects metadata** into outbound (downstream) requests.

Anatomy of Distributed Tracing



Technical background of tracing

Tracing은 1990년도 부터 있었으나, Google의 논문"Dapper, a Large-Scale Distributed System Tracing Infrastructure " 발표 이후 주류로 자리 매김하게 됨

- Dapper (Google): Foundation for all tracers
- Zipkin (Twitter)
- Jaeger (Uber)
- StackDriver Trace (Google)
- Appdash (golang)
- X-ray (AWS)

Tracing Systems

Zipkin and OpenZipkin

- 최초의 Open Source Tracing system
- 2012년, Twitter 공개
- 많은 지원 생태계 구성
- 자체 Tracer인 Brave API 기반으로 다양한 fraework가 이를 지원 (ex: Spring, Spark, Kafka, gRPC등)
- 다른 tracing System과는 data-format level에서 연동

Jaeger

- 2015년 Uber에서 만들어졌고, 2017년 Open Source Project 로 공개
- CNCF산하의 프로젝트로 OpenTracing에 집중하면서 많은 주목을 받고 있음
- Zipkin과의 호환 (B3 metadata format)
- 직접적인 instrumentation을 제공하지 않으며 OpenTracing-compatible tracer를 이용

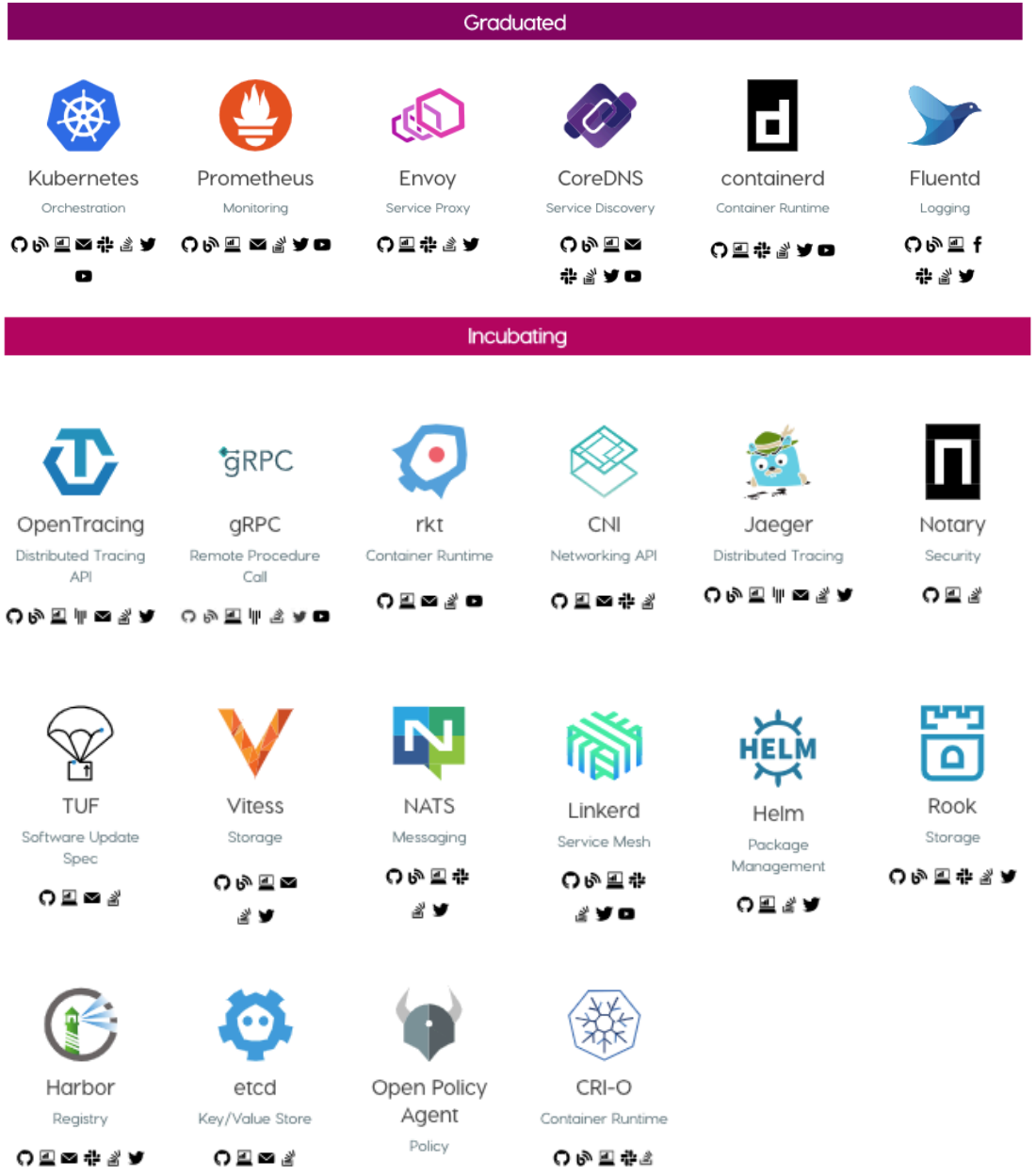
CNCF에서 Observability 비중

점점 비중이 강조 되고 있음

약 20%

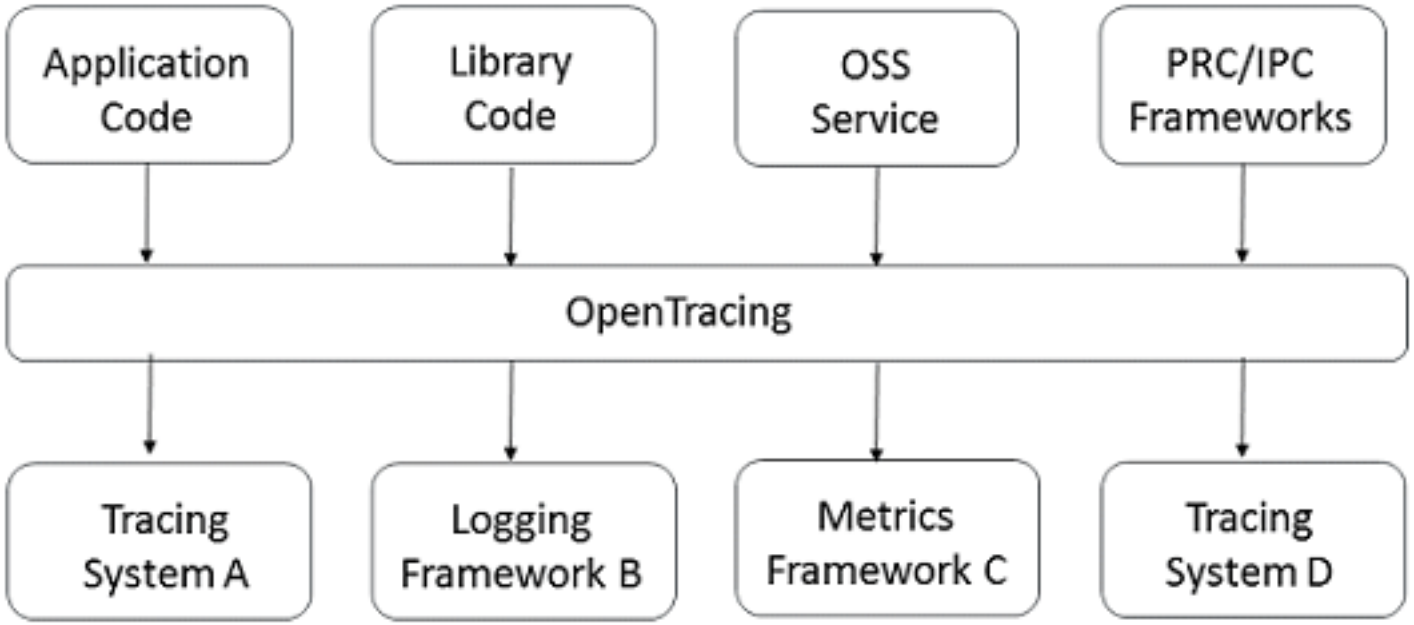
- Graduated : Prometehus, Fluentd (Log collection layer)
- Incubating : OpenTracing, Jaeger,
- Sandbox : OpenMetrics and Cortex (Monitoring)

- Prometheus: A monitoring and alerting platform
- Fluentd: A logging data collection layer
- OpenTracing: A vendor-neutral APIs and instrumentation for distributed tracing
- Jaeger: A distributed tracing platform



OpenTracing

The OpenTracing specification is created to address the problem of API incompatibility between different distributed tracing systems. OpenTracing is a lightweight standardization layer and is located between applications/class libraries and tracing or log analysis programs.

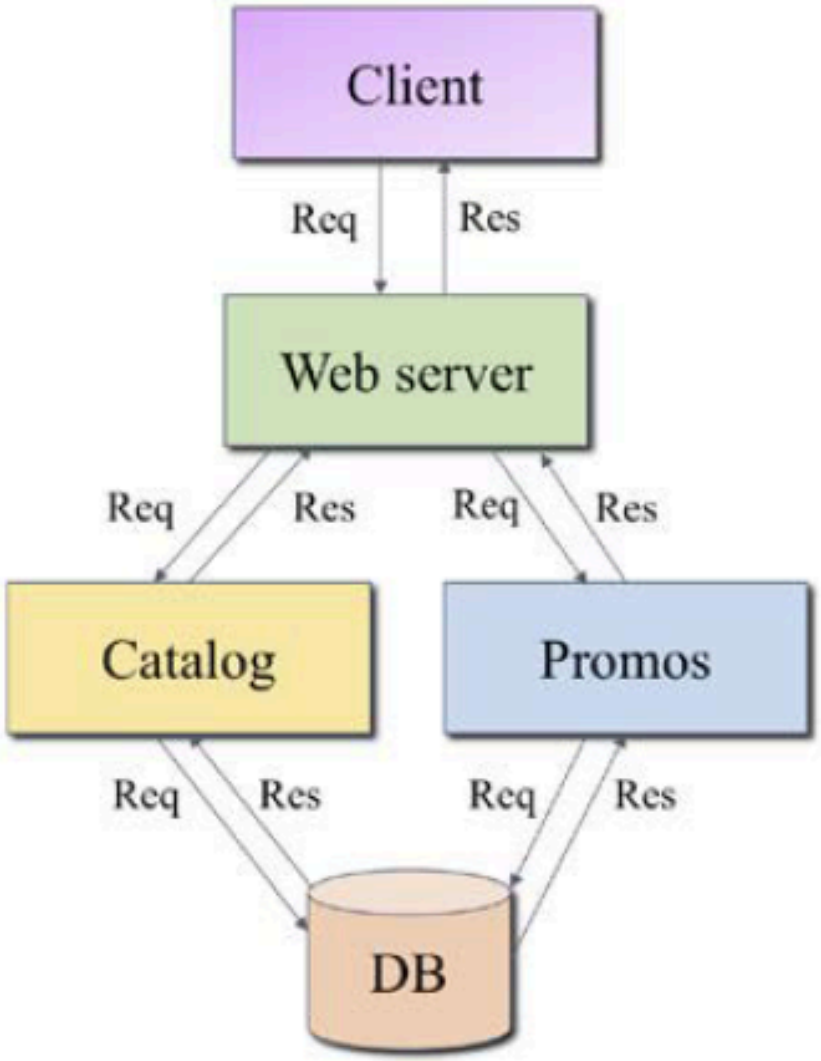


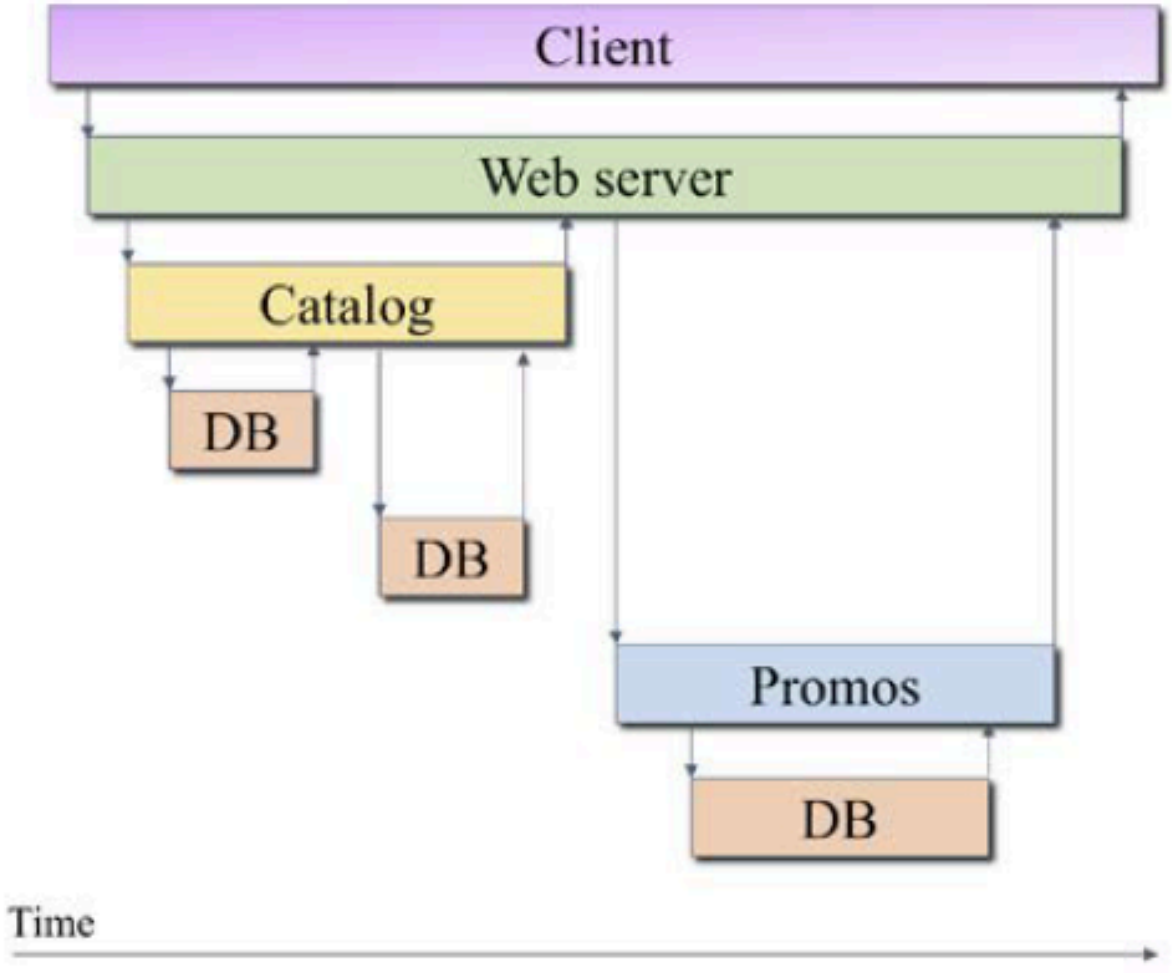
- **Advantages**

OpenTracing already enters CNCF and provides unified concept and data standards for global distributed tracing systems.

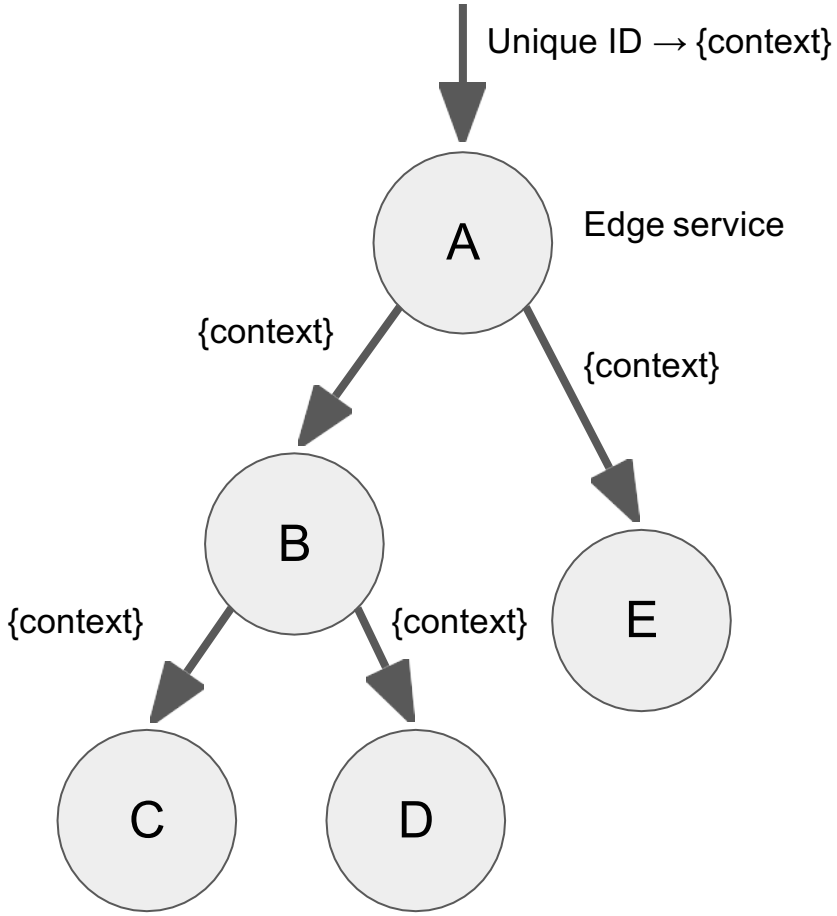
- OpenTracing provides APIs with no relation to platforms or vendors, which allows developers to conveniently add (or change) the implementation of tracing system.

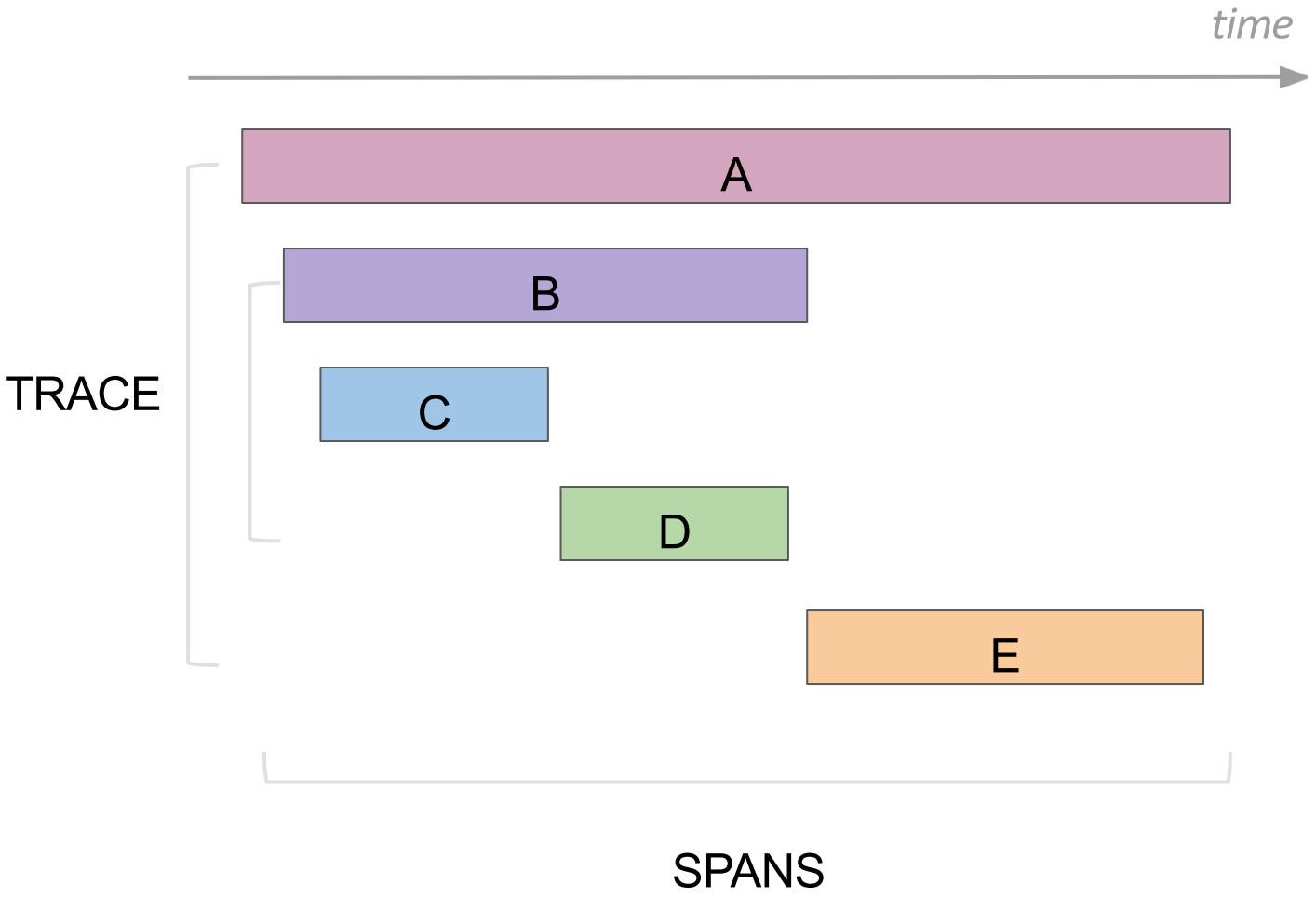
Basic Idea of Tracing





Distributed Tracing In A Nutshell

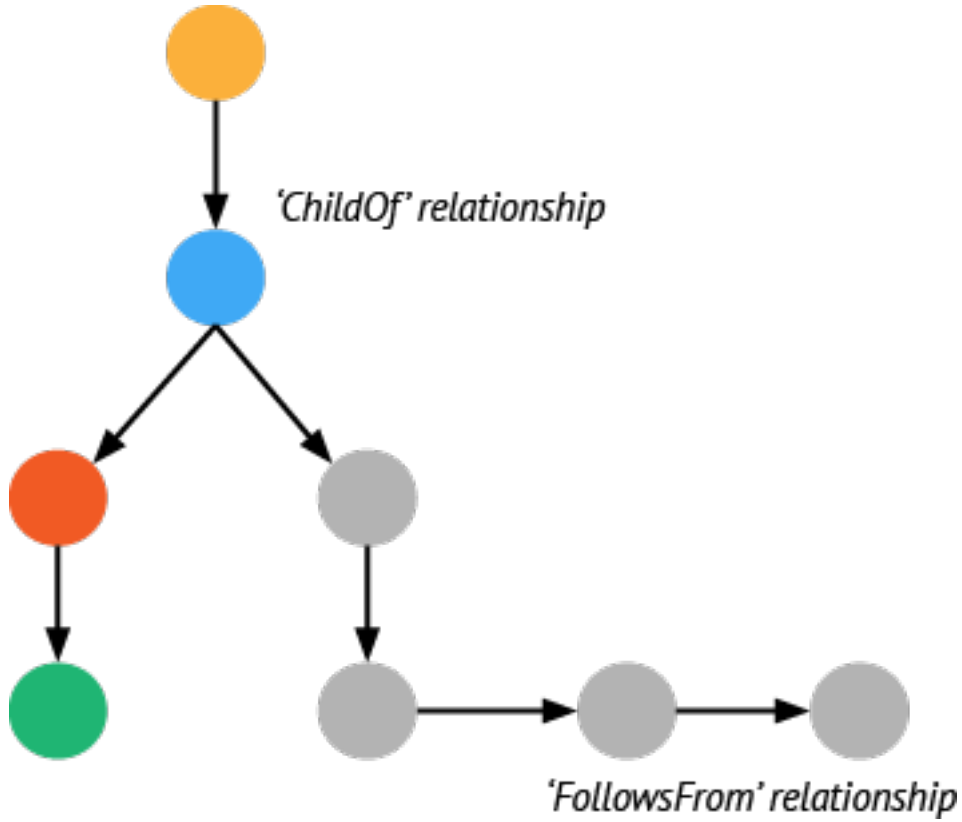




Span Relationship

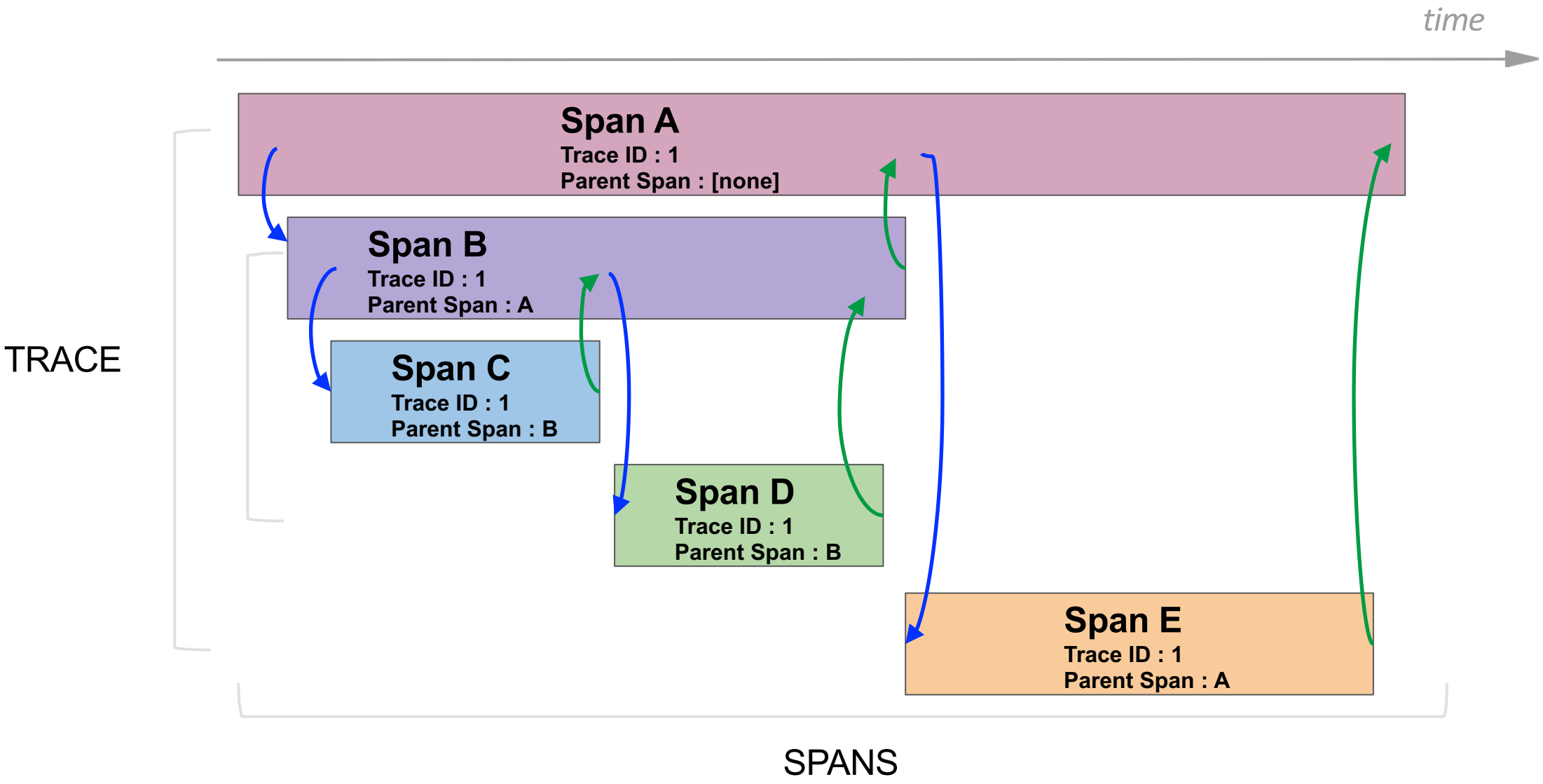
Traces in OpenTracing are defined implicitly by their **Spans**. In particular, a **Trace** can be thought of as a directed acyclic graph (DAG) of **Spans**, where the edges between **Spans** are called **References**.
For example, the following is an example **Trace** made up of 8 **Spans**:
Causal relationships between Spans in a single Trace

```
        [Span A]  ←←←(the root span)
            |
     +------+------+
     |             |
[Span B]       [Span C] ←←←(Span C is a `ChildOf` Span A)
     |             |
[Span D]       +---+-------+
               |           |
          [Span E]    [Span F] >>> [Span G] >>> [Span H]
                                          ↑
                                          ↑
                                          ↑
                                 (Span G `FollowsFrom` Span F)
```



https://opentracing.io/specification/

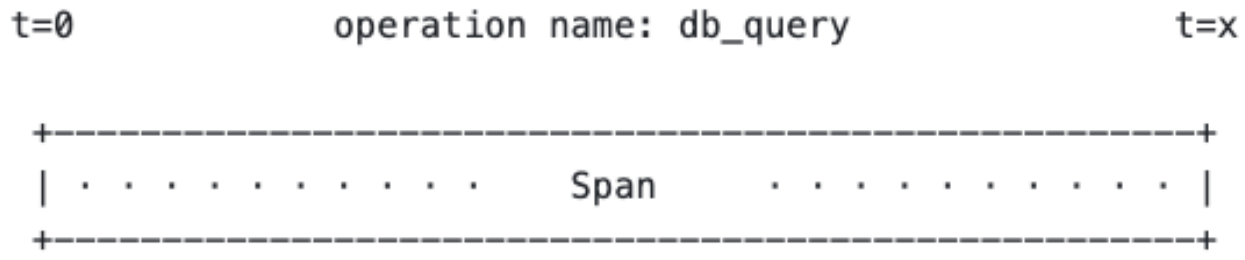
Distributed Tracing In A Nutshell



What is a Span

The “**span**” is the primary building block of a distributed trace, representing an individual unit of work done in a distributed system.
Each component of the distributed system contributes a span - a named, timed operation representing a piece of the workflow.
Spans can (and generally do) contain “References” to other spans, which allows multiple Spans to be assembled into one complete **Trace** - a visualization of the life of a request as it moves through a distributed system.

Example Span:



Each span encapsulates the following state according to the OpenTracing specification:

- An **operation name**
- A **start timestamp and finish timestamp**
- A set of key:value span **Tags**
- A set of key:value span **Logs**
- A **SpanContext**

What is a Span – Tags

Tags are key:value pairs that enable user-defined annotation of spans in order to query, filter, and comprehend trace data.
Span tags should apply to the *whole* span. There is a list available at semantic_conventions.md listing conventional span tags for common scenarios. Examples may include tag keys like db.instance to identify a databse host, http.status_code to represent the HTTP response code, or error which can be set to True if the operation represented by the Span fails.

- Example
  - http.url = "http://google.com"
  - http.status_code = 200
  - peer.service = "mysql"
  - db.statement = "select * from users"

What is a Span – Logs

Logs are key:value pairs that are useful for capturing *span-specific* logging messages and other debugging or informational output from the application itself. Logs may be useful for documenting a specific moment or event within the span (in contrast to tags which should apply to the span as a whole).

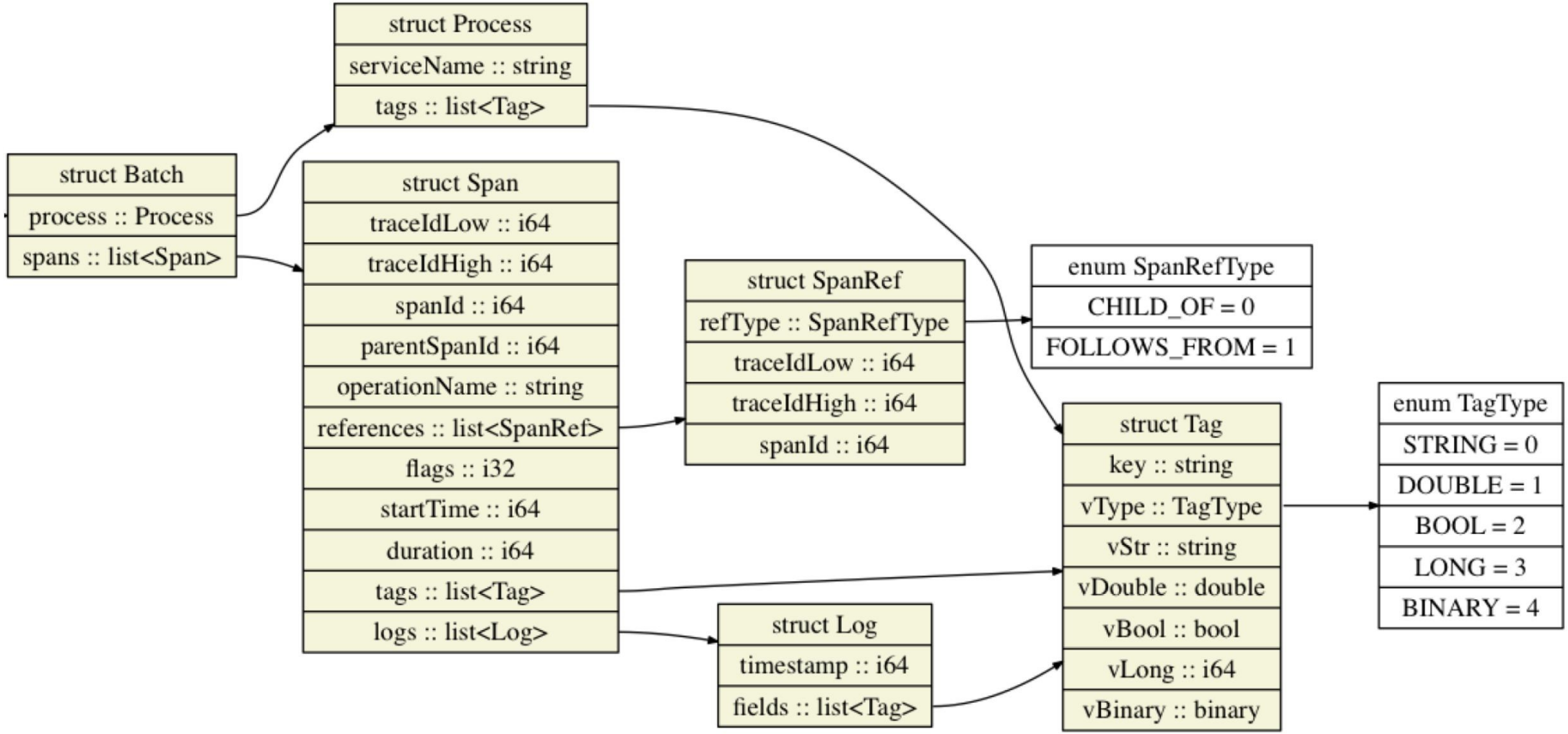
Describes an event at a point in time during the span lifetime.

- OpenTracing supports structured logging
- Contains a timestamp and a set of fields

- Example

```
span.log_kv(
        {'event': 'open_conn', 'port': 433}
)
```

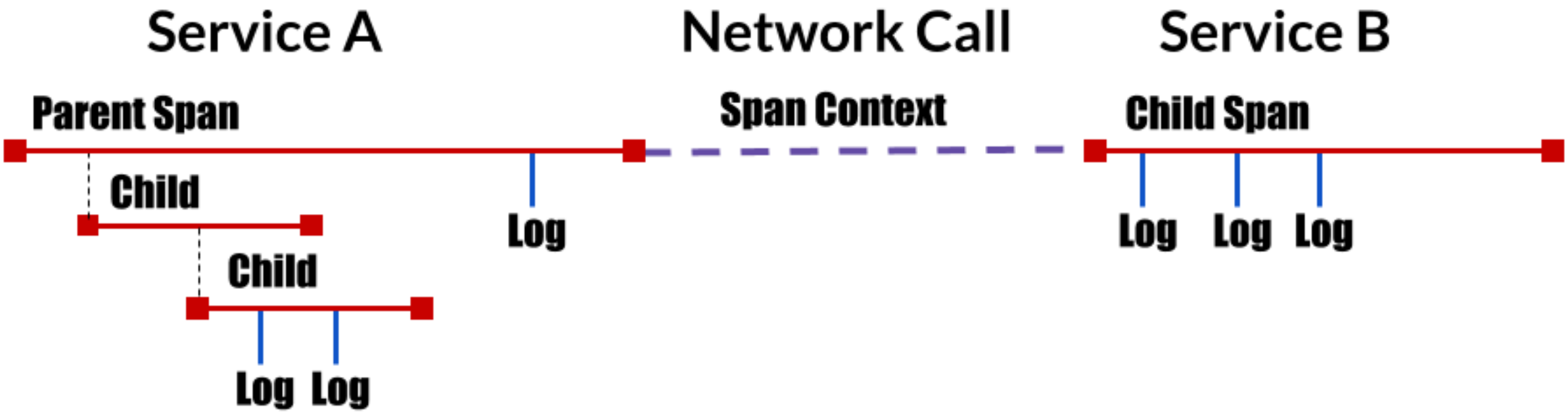
OpenTracing Data Model



https://opentracing.io/specification/

Distributed Tracing In A Nutshell

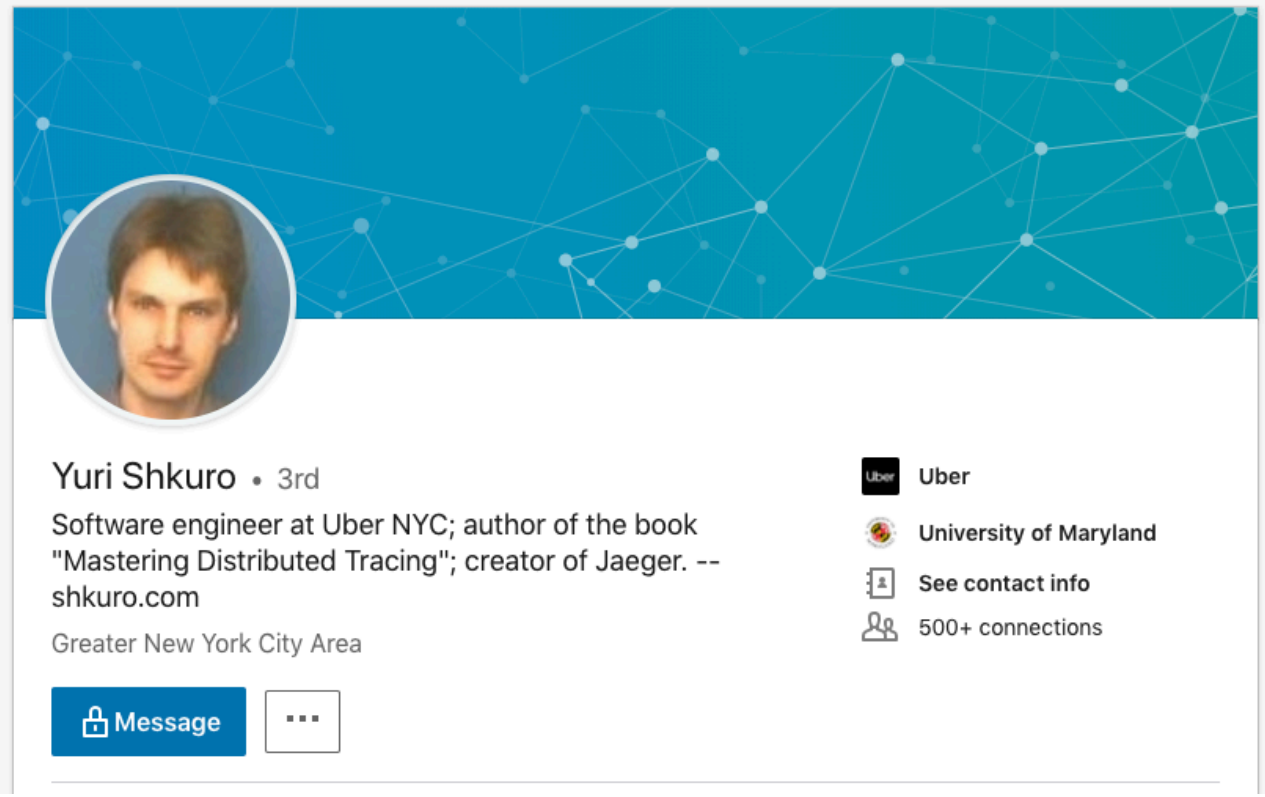
Remote Procedure Call 환경



Jaeger

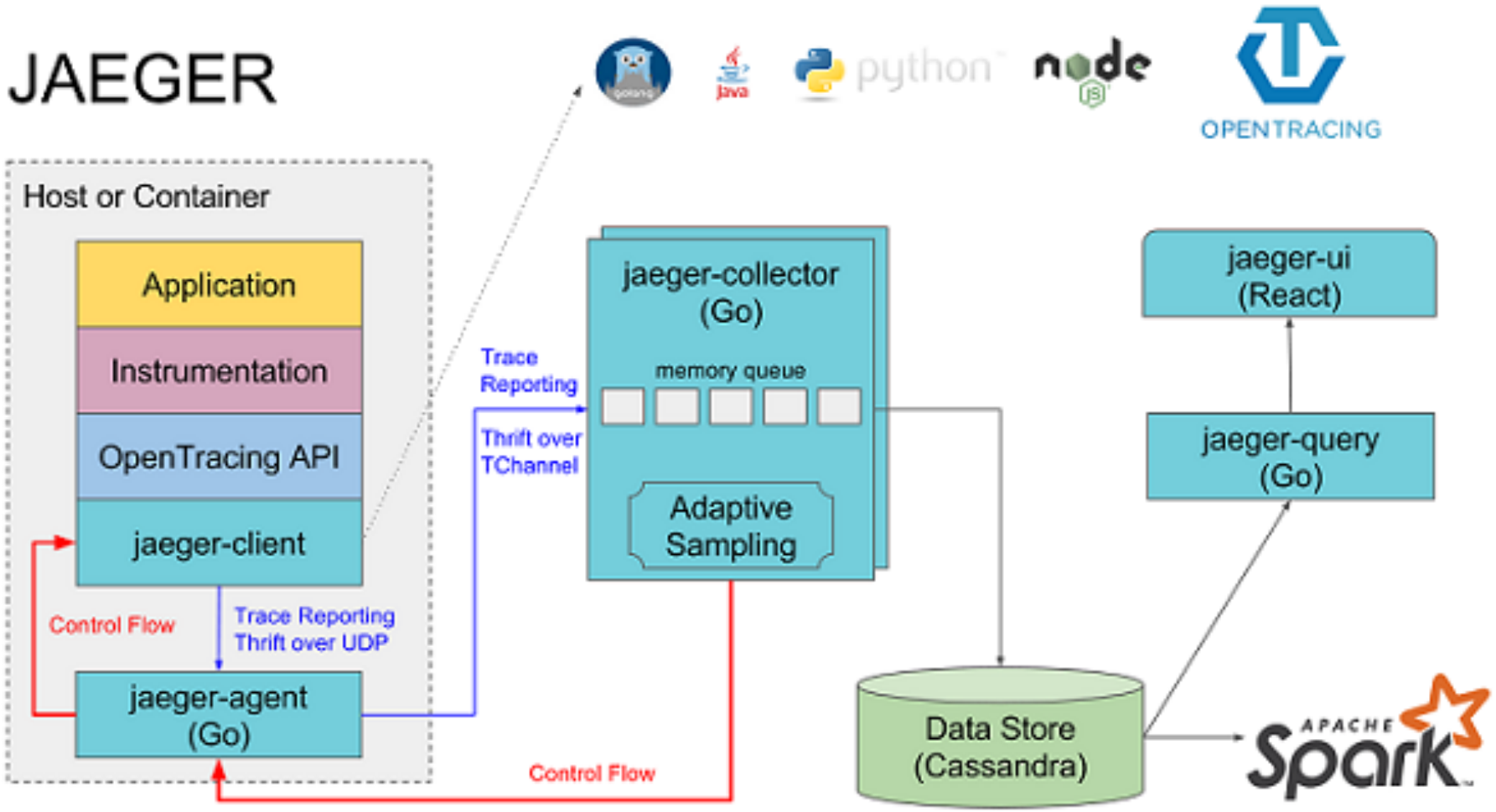
Jaeger

- 2015년 Uber에서 만들어졌고, 2017년 Open Source Project 로 공개
- CNCF산하의 프로젝트로 OpenTracing에 집중하면서 많은 주목을 받고 있음
- Zipkin과의 호환 (B3 metadata format)
- 직접적인 instrumentation을 제공하지 않으며 OpenTracing-compatible tracer를 이용



Jaeger

Jaeger is an open-source distributed tracing system released by Uber. It is compatible with OpenTracing APIs.



Jaeger Architecture

- **Jaeger client:** Implements SDKs that conform to OpenTracing standards for different languages. Applications use the API to write data. The client library transmits trace information to the jaeger-agent according to the sampling policy specified by the application.
- **Agent**: A network daemon that monitors span data received by the UDP port and then sends the data to the collector in batches. It is designed as a basic component and deployed to all hosts. The agent decouples the client library and collector, shielding the client library from collector routing and discovery details.
- **Collector**: Receives the data sent by the jaeger-agent and then writes the data to backend storage. The collector is designed as a stateless component. Therefore, you can simultaneously run an arbitrary number of jaeger-collectors.
- **Data store**: The backend storage is designed as a pluggable component that supports writing data to Cassandra and Elasticsearch.
- **Query**: Receives query requests, retrieves trace information from the backend storage system, and displays it in the UI. Query is stateless and you can start multiple instances. You can deploy the instances behind load balancers such as Nginx.

Understanding Sampling

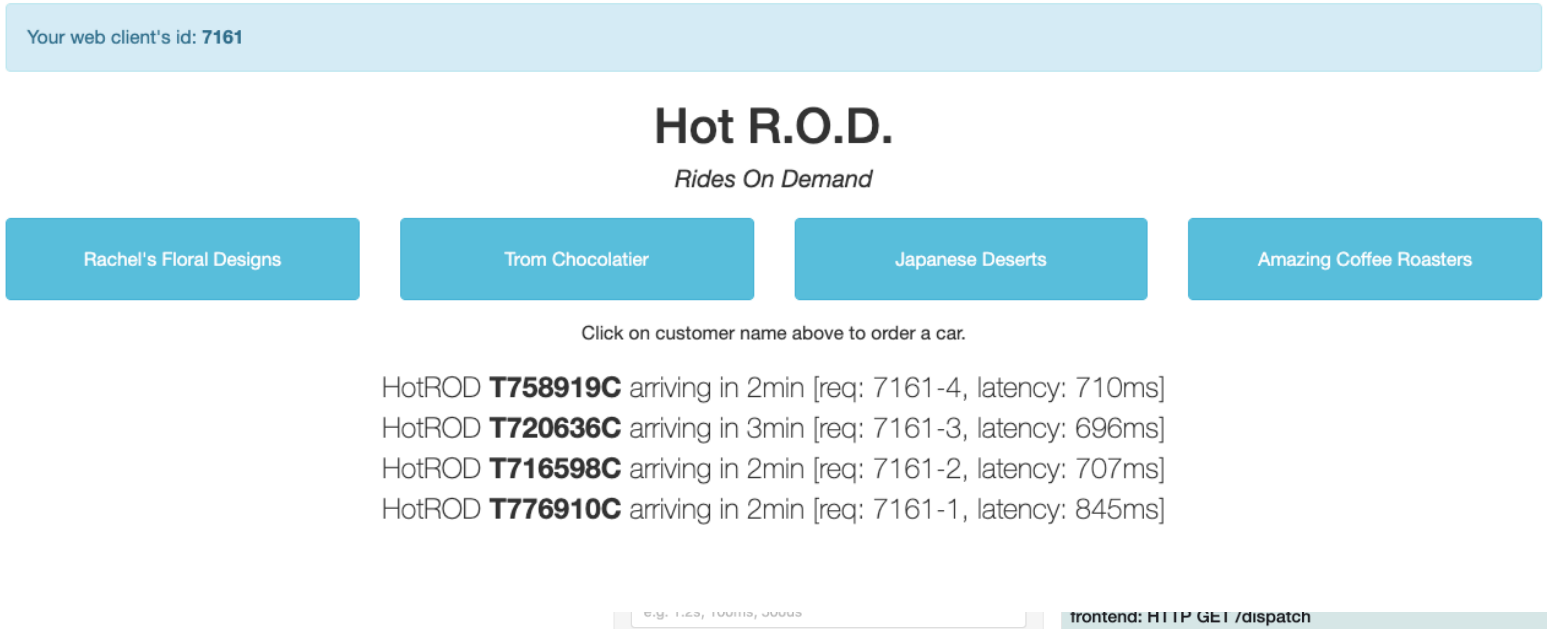
- Tracing data > than business traffic
- Most tracing systems sample transactions
- **Head-based sampling**: the sampling decision is made just before the trace is started, and it is respected by all nodes in the graph
- **Tail-based sampling**: the sampling decision is made after the trace is completed / collected
  - Head-based sampling don't catch 0.001 probability anomalous behavior.

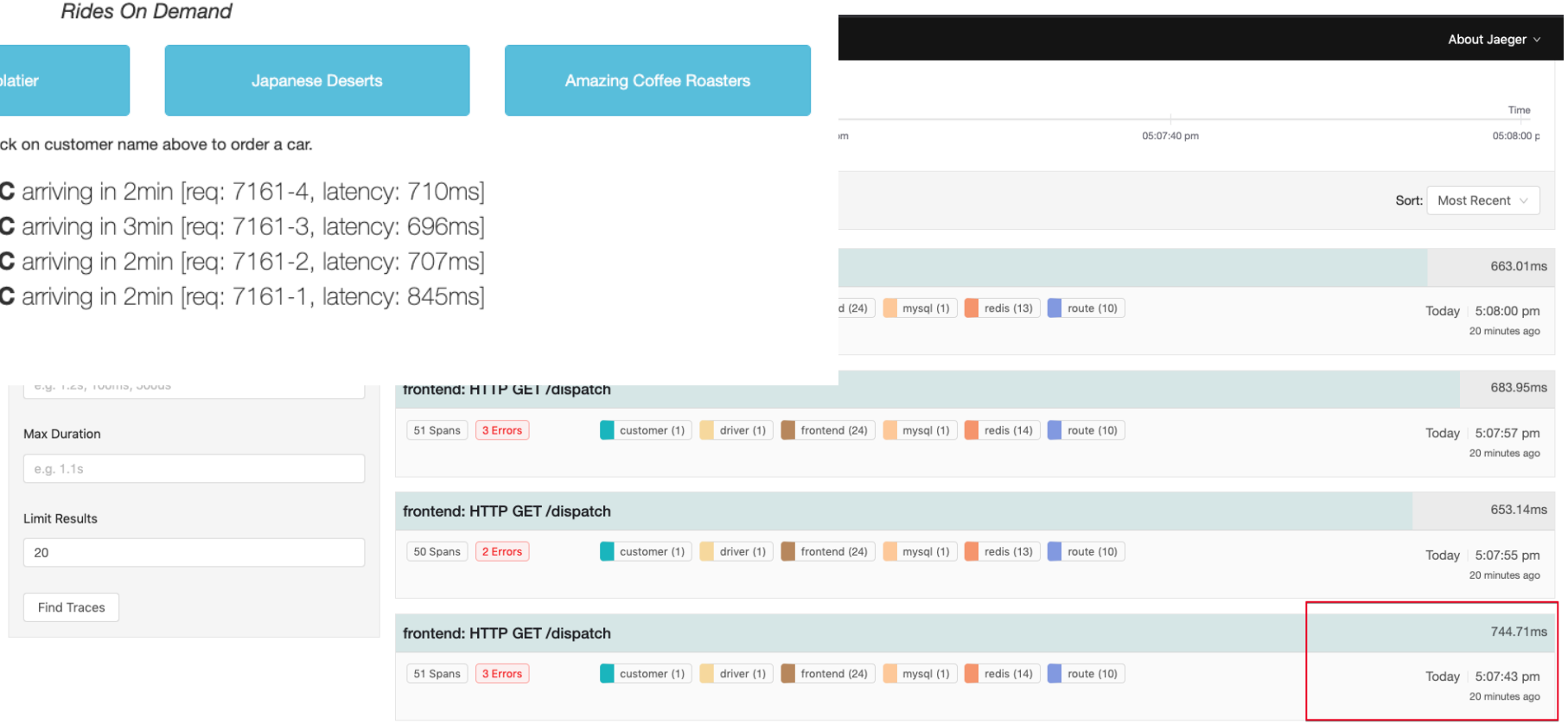
PART 2

Take a first Tracing

HOTROD

Sample Application을 이용한 Jaeger와 친해지기

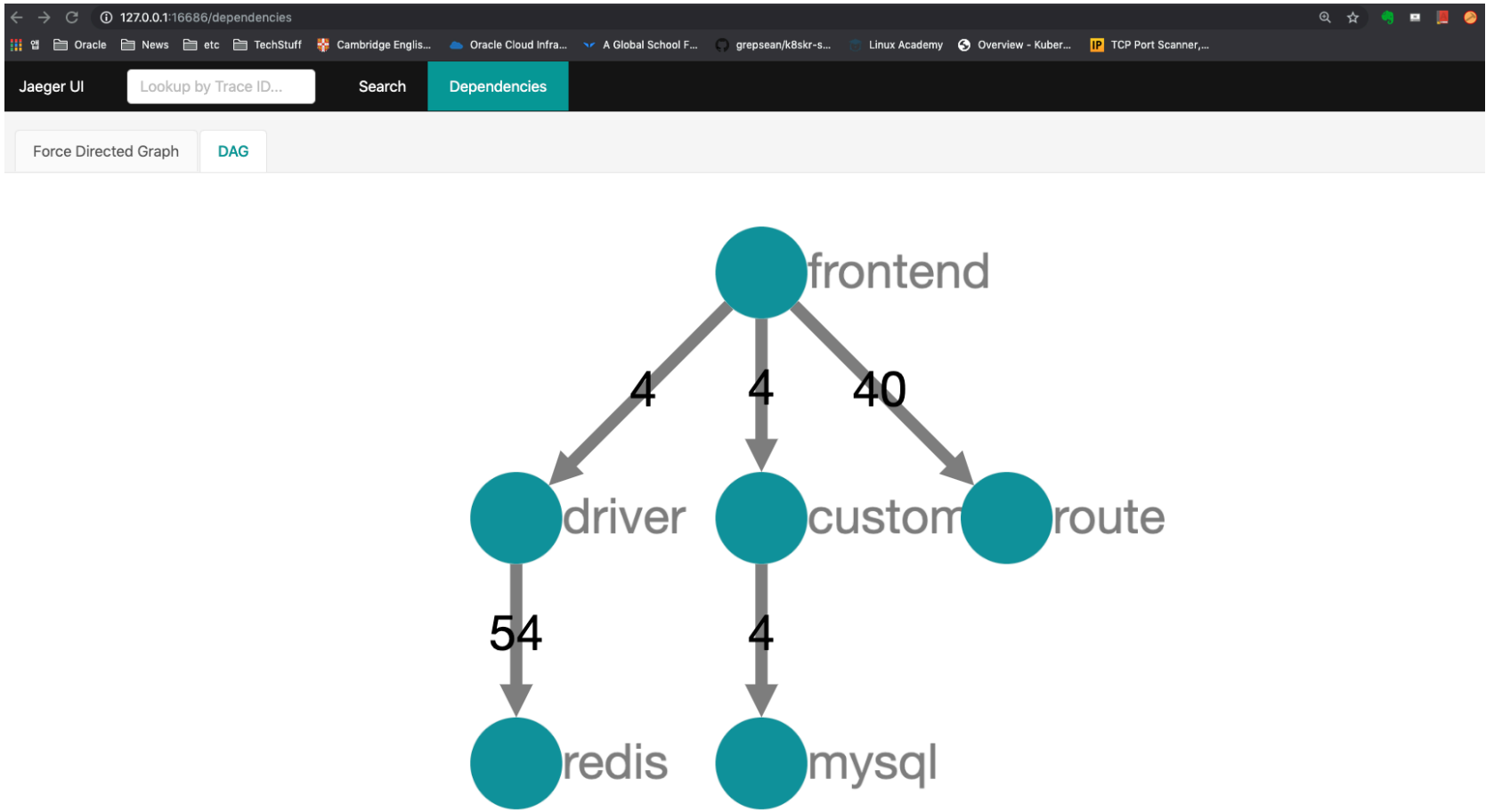




HOTROD

Architecture 확인

- 전체 Architecture을 DAG(Directed Acyclic Graph)로 확인



HOTROD

Sample Application을 이용한 Jaeger와 친해지기

- **https://github.com/DannyKang/OpenTracing**

- 사전 준비 사항
  - Docker
  - Go lang (소스코드에서 직접 실행을 원하는 경우)

- Jaeger 실행

```
$ sudo docker run -d --name jaeger ₩
-p 6831:6831/udp ₩
-p 16686:16686 ₩
-p 14268:14268 ₩
jaegertracing/all-in-one:1.6
```

- Jaeger 접속 : http://localhost:16686/

- HOTROD 실행

```
$ sudo docker run --rm -it ₩
--link jaeger ₩
-p8080-8083:8080-8083 ₩
jaegertracing/example-hotrod:1.6 ₩
all ₩
--jaeger-agent.host-port=jaeger:6831
```

- HotROD 접속 : http://127.0.0.1:8080/

https://medium.com/opentracing/take-opentracing-for-a-hotrod-ride-f6e3141f7941

HOTROD

Sample Application을 이용한 Jaeger와 친해지기

- Source Code로 실행하기 (Go가 설치되었다는 가정하에)

- mkdir -p $GOPATH/src/github.com/jaegertracing
  cd $GOPATH/src/github.com/jaegertracing
  git clone git@github.com:jaegertracing/jaeger.git jaeger
  cd jaeger
  make install
  cd examples/hotrod
  go run ./main.go all

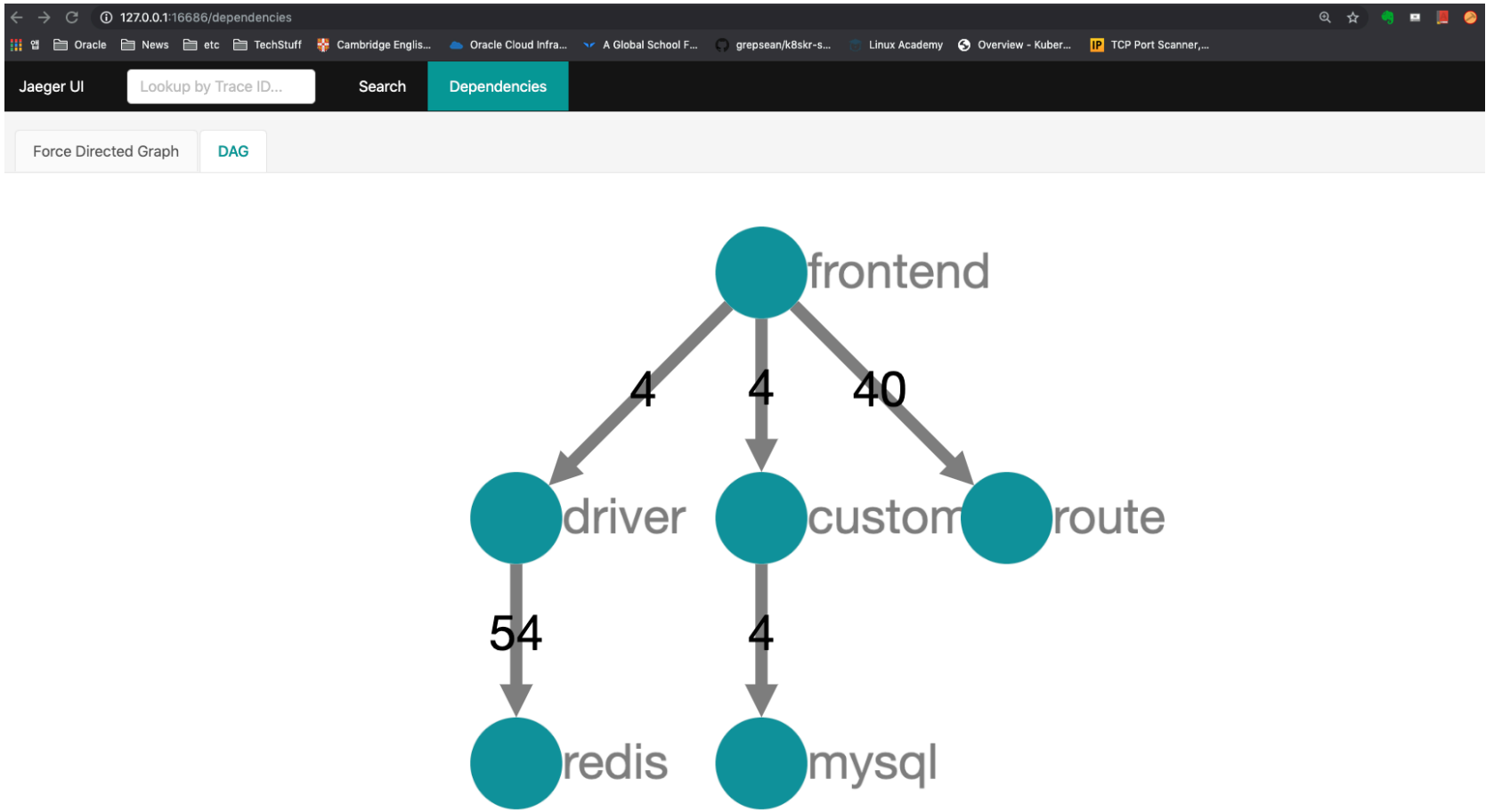
- 이후에 옵션등을 통해서 바로 실행하기 위해서 go build 로 binary를 만든다.
  go build

여기서 'all' 옵션을 전제 마이크로서비스를 하나의 binary로 실행한다는 의미이다.
그리고 log가 Standard out에 쓰기 때문에 console에서 전체 내용을 확인할 수 있다.

HOTROD

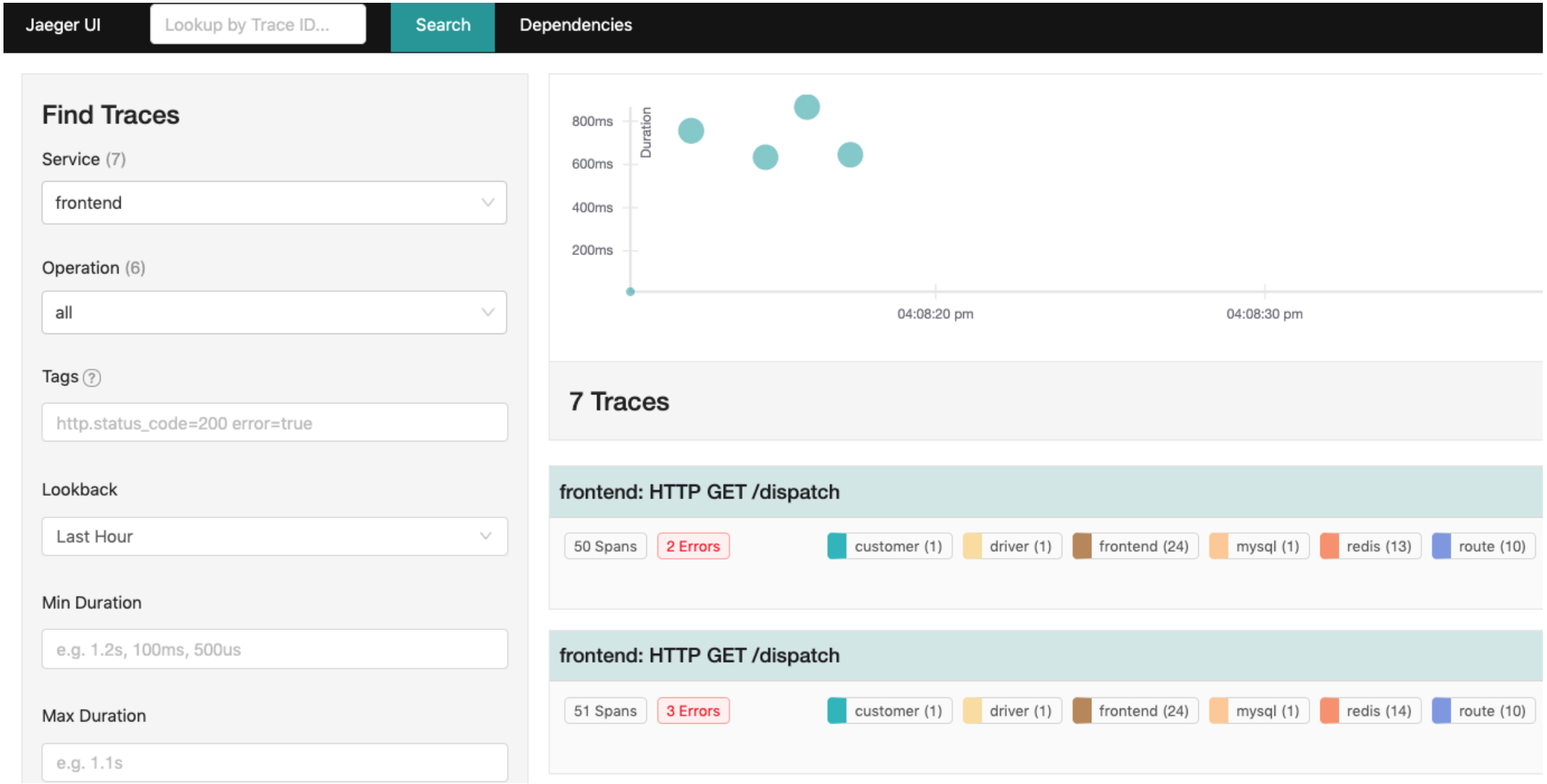
Architecture 확인

- 전체 Architecture을 DAG(Directed Acyclic Graph)로 확인



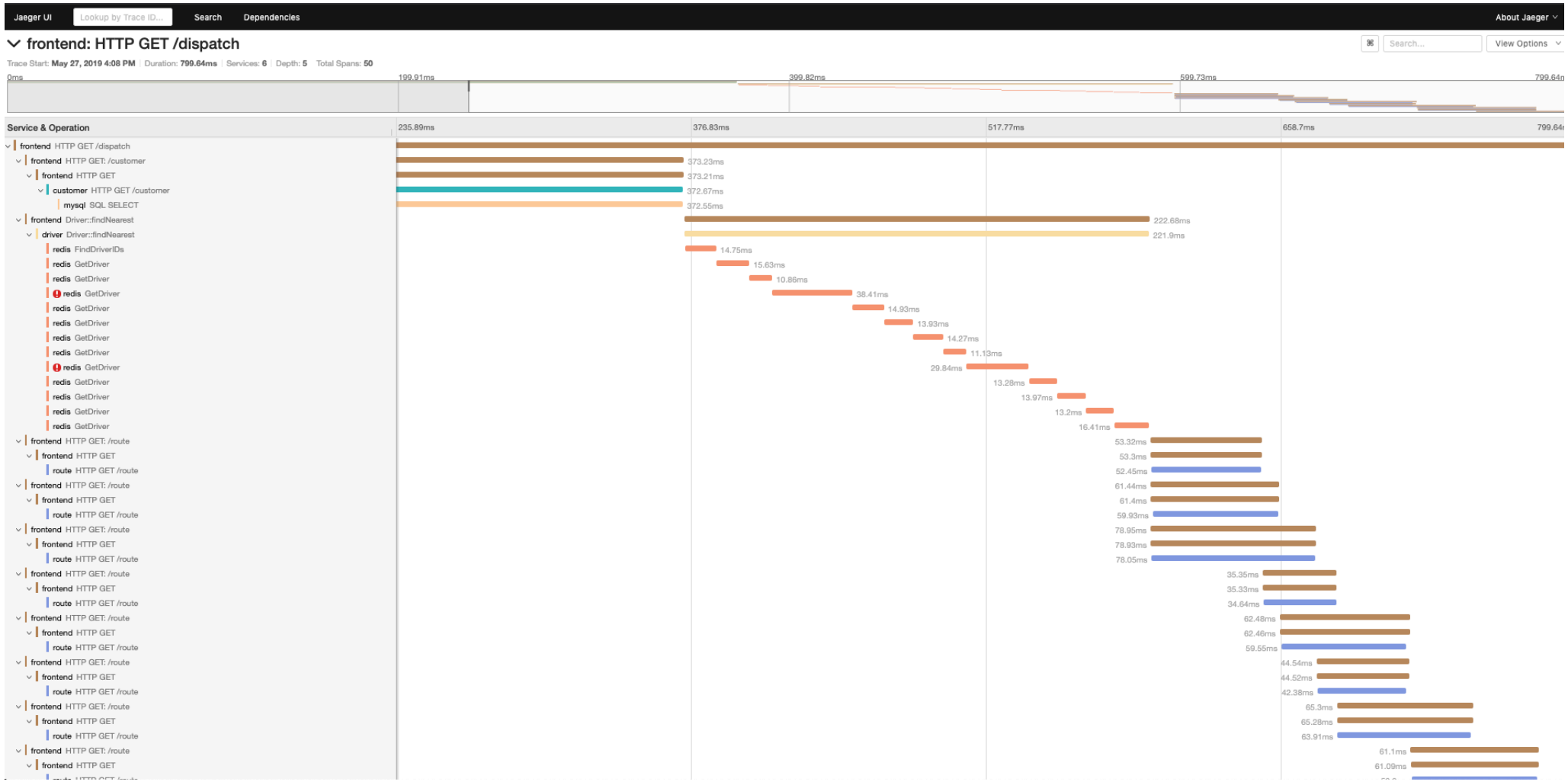
HOTROD

DataFlow 확인



HOTROD

DataFlow 확인



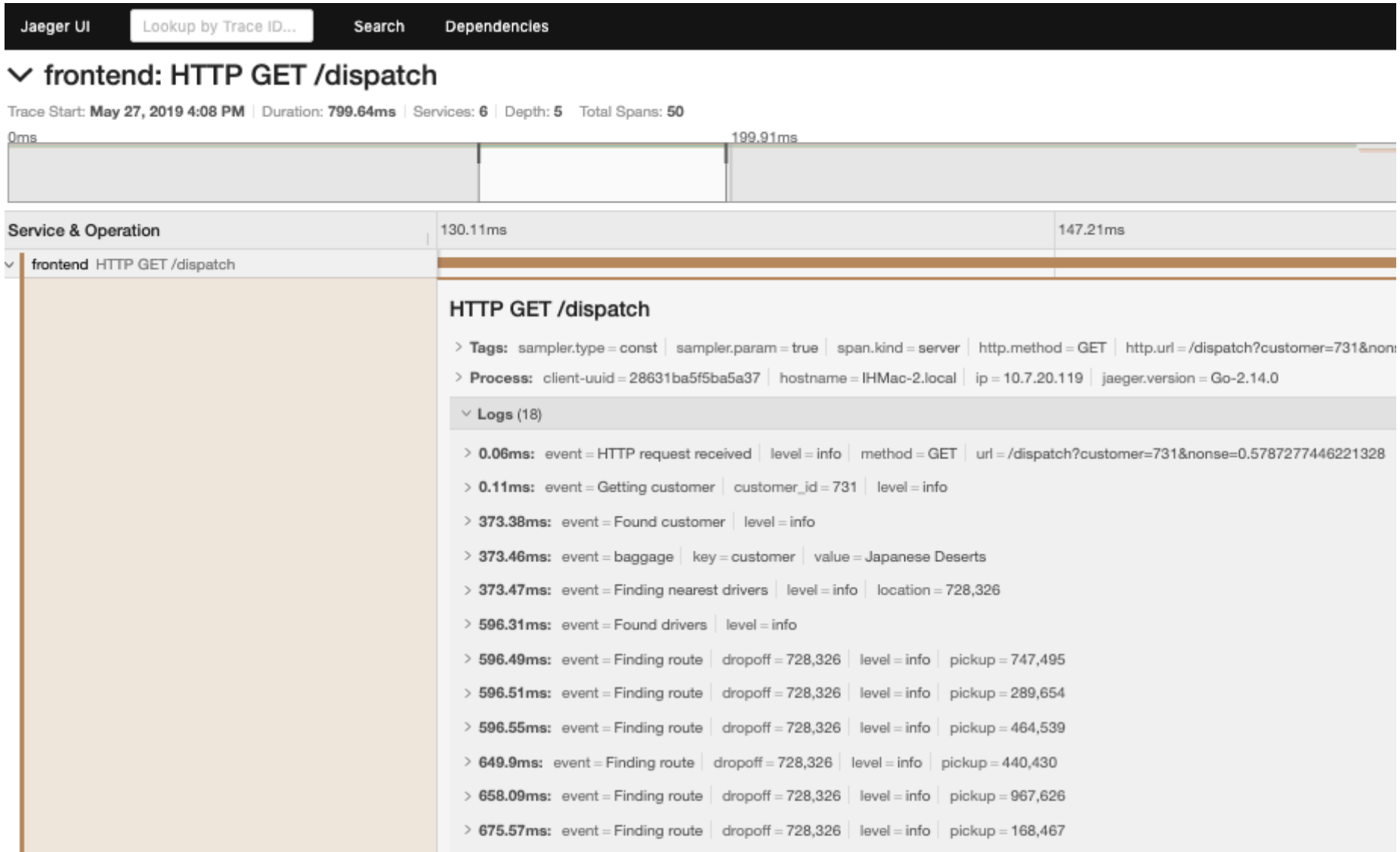
HOTROD

Traditional Log vs Contextual Logs

```
2017-05-03T23:56:20.467-0400    INFO    log/spanlogger.go:40    HTTP    {"service": "frontend", "method": "GET", "url": "/favicon.ico"}
2017-05-03T23:59:30.356-0400    INFO    log/spanlogger.go:40    HTTP    {"service": "frontend", "method": "GET", "url": "/"}
2017-05-03T23:59:30.463-0400    INFO    log/spanlogger.go:40    HTTP    {"service": "frontend", "method": "GET", "url": "/favicon.ico"}
2017-05-04T00:36:34.417-0400    INFO    log/spanlogger.go:40    HTTP request received   {"service": "frontend", "method": "GET", "url": "/dispatch?customer=123&nonse=0.8534072559455979"}
2017-05-04T00:36:34.417-0400    INFO    log/spanlogger.go:40    Getting customer        {"service": "frontend", "component": "customer_client", "customer_id": "123"}
2017-05-04T00:36:34.419-0400    INFO    log/spanlogger.go:40    HTTP request received   {"service": "customer", "method": "GET", "url": "/customer?customer=123"}
2017-05-04T00:36:34.419-0400    INFO    log/spanlogger.go:40    Loading customer        {"service": "customer", "component": "mysql", "customer_id": "123"}
2017-05-04T00:36:34.725-0400    INFO    log/spanlogger.go:40    Found customer  {"service": "frontend", "customer": {"ID":"123","Name":"Rachel's Floral Designs","Location":"115,277"}}
2017-05-04T00:36:34.725-0400    INFO    log/spanlogger.go:40    Finding nearest drivers {"service": "frontend", "component": "driver_client", "location": "115,277"}
2017-05-04T00:36:34.728-0400    INFO    log/spanlogger.go:40    Searching for nearby drivers    {"service": "driver", "location": "115,277"}
2017-05-04T00:36:34.775-0400    ERROR   log/spanlogger.go:45    redis timeout   {"service": "driver", "driver_id": "T748713C", "error": "redis timeout"}
github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap.Stack
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap/field.go:209
github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap.(*Logger).check
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap/logger.go:273
github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap.(*Logger).Error
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap/logger.go:176
github.com/uber/jaeger/examples/hotrod/pkg/log.spanLogger.Error
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/pkg/log/spanlogger.go:45
github.com/uber/jaeger/examples/hotrod/pkg/log.(*spanLogger).Error
        <autogenerated>:9
github.com/uber/jaeger/examples/hotrod/services/driver.(*Redis).GetDriver
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/services/driver/redis.go:89
github.com/uber/jaeger/examples/hotrod/services/driver.(*Server).FindNearest
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/services/driver/server.go:91
github.com/uber/jaeger/examples/hotrod/services/driver/thrift-gen/driver.(*tchanDriverServer).handleFindNearest
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/services/driver/thrift-gen/driver/tchan-driver.go:92
github.com/uber/jaeger/examples/hotrod/services/driver/thrift-gen/driver.(*tchanDriverServer).Handle
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/services/driver/thrift-gen/driver/tchan-driver.go:76
github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go/thrift.(*Server).handle
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go/thrift/server.go:133
github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go/thrift.(*Server).Handle
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go/thrift/server.go:283
github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go.(*handlerMap).Handle
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go/handlers.go:118
github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go.channelHandler.Handle
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go/handlers.go:126
github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go.(*Connection).dispatchInbound
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/github.com/uber/tchannel-go/inbound.go:195
2017-05-04T00:36:34.777-0400    ERROR   log/spanlogger.go:45    Retrying GetDriver after error  {"service": "driver", "retry_no": 1, "error": "redis timeout"}
github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap.Stack
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap/field.go:209
github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap.(*Logger).check
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap/logger.go:273
github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap.(*Logger).Error
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/vendor/go.uber.org/zap/logger.go:176
github.com/uber/jaeger/examples/hotrod/pkg/log.spanLogger.Error
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/pkg/log/spanlogger.go:45
github.com/uber/jaeger/examples/hotrod/pkg/log.(*spanLogger).Error
        <autogenerated>:9
github.com/uber/jaeger/examples/hotrod/services/driver.(*Server).FindNearest
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/services/driver/server.go:96
github.com/uber/jaeger/examples/hotrod/services/driver/thrift-gen/driver.(*tchanDriverServer).handleFindNearest
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/services/driver/thrift-gen/driver/tchan-driver.go:92
github.com/uber/jaeger/examples/hotrod/services/driver/thrift-gen/driver.(*tchanDriverServer).Handle
        /Users/yurishkuro/golang/src/github.com/uber/jaeger/examples/hotrod/services/driver/thrift-gen/driver/tchan-driver.go:76
```

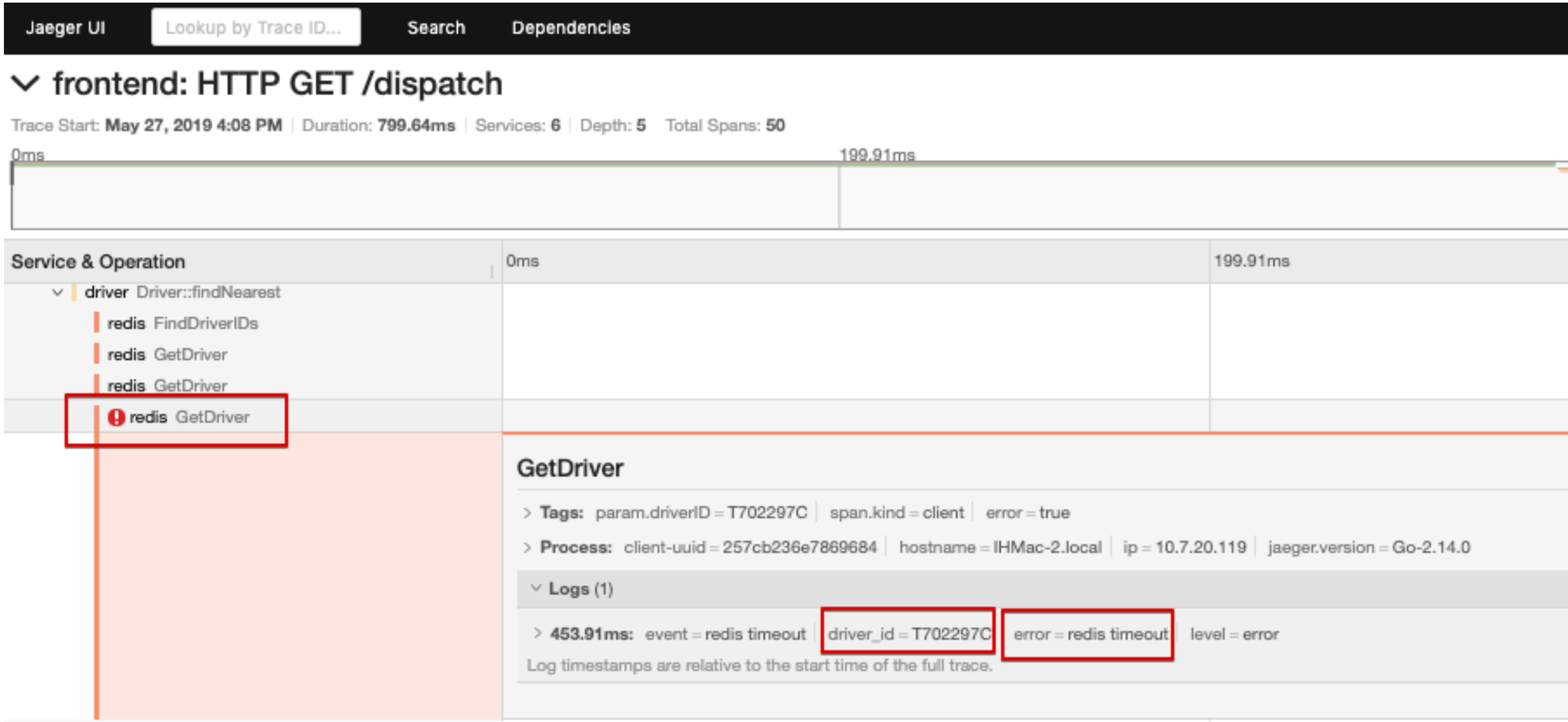
HOTROD

Traditional Log vs Contextual Logs



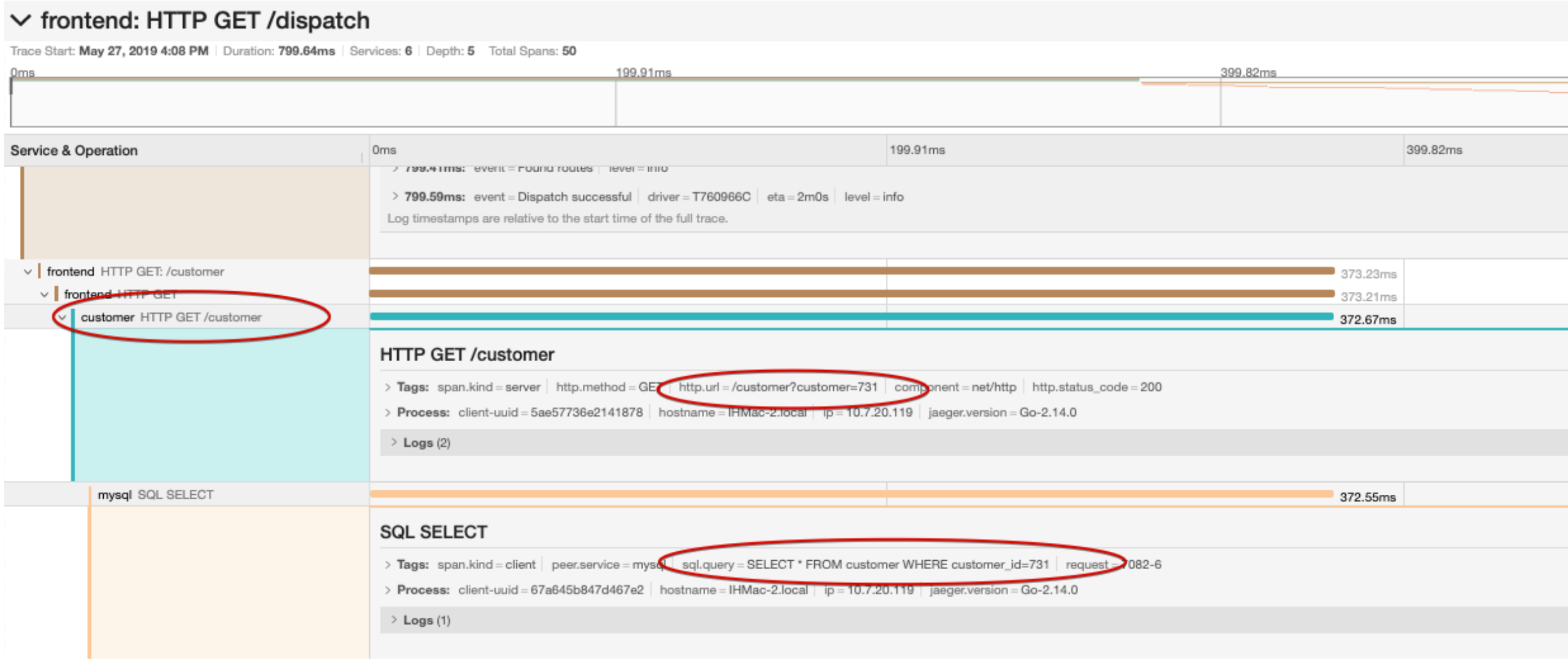
HOTROD

Traditional Log vs Contextual Logs



HOTROD

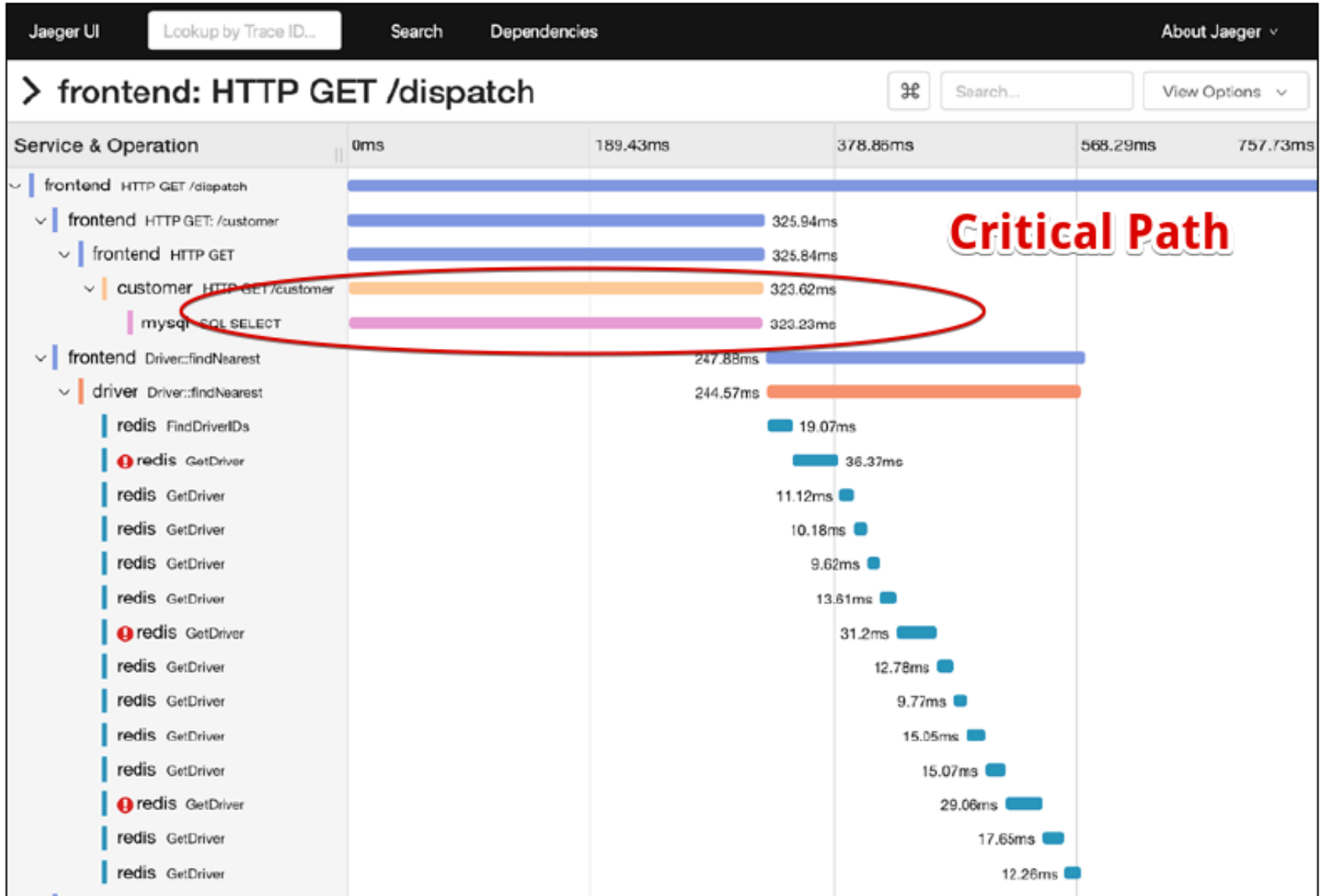
Span tags versus logs



HOTROD

Identifying source of latency

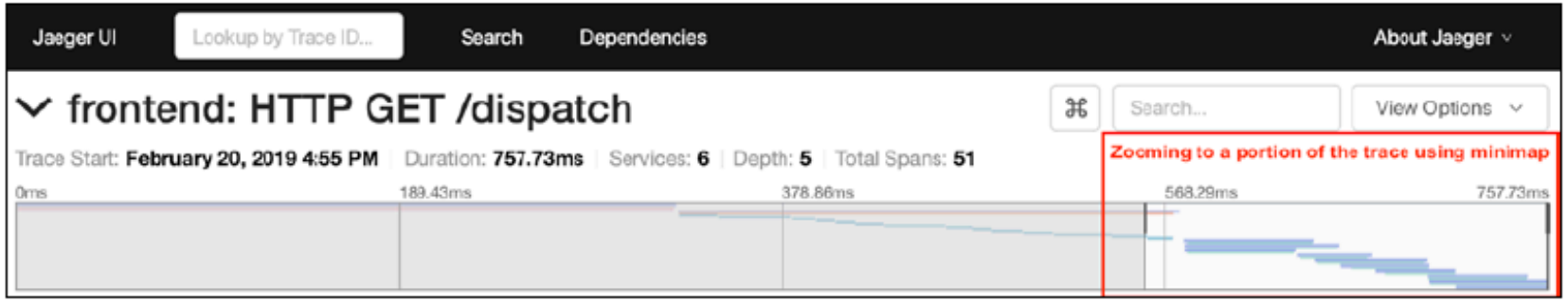
Mysql이 전체 수행 시간에 가장 많은 시간(latency)이 걸림

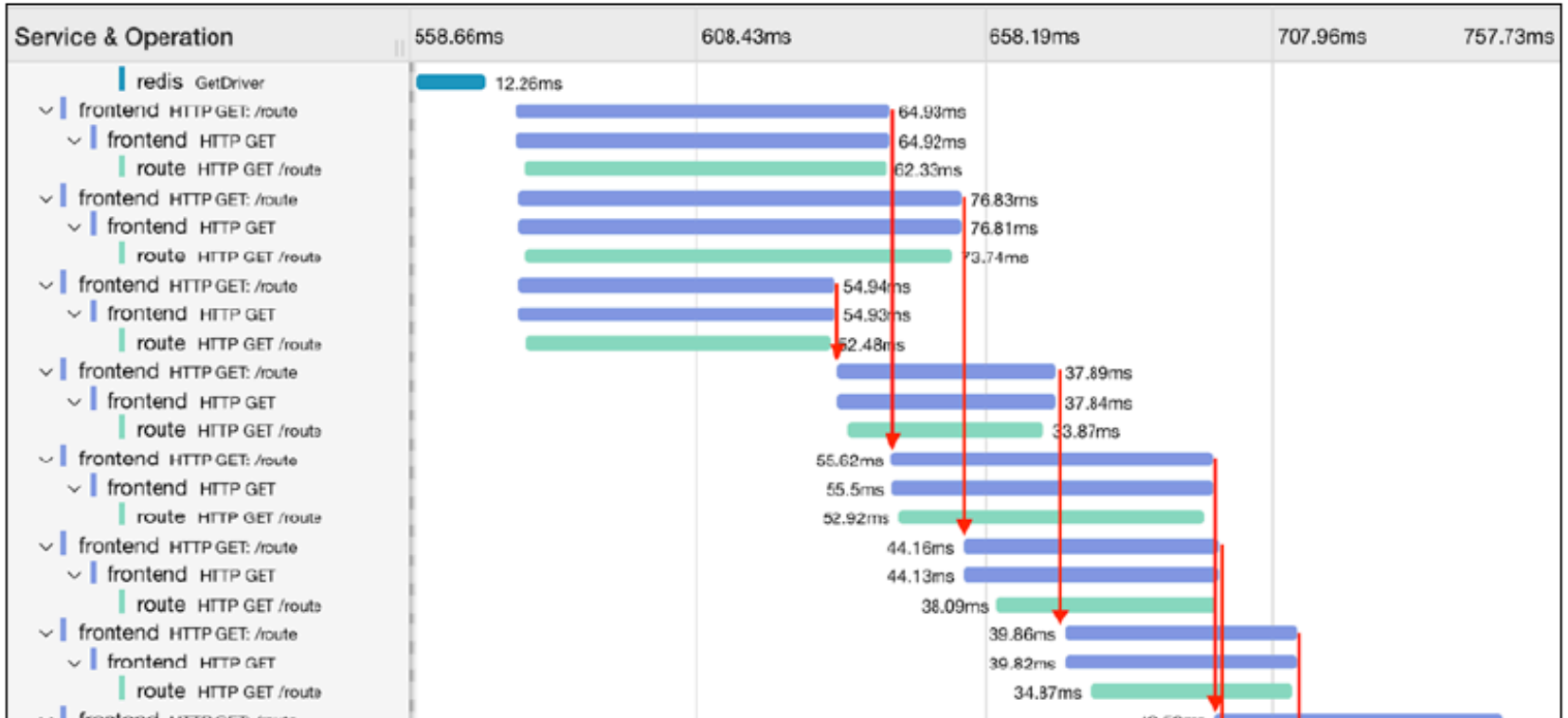


HOTROD

Identifying source of latency

Zoom in 을 통해서 보면 route service가 parallel이 아니라 3개씩 쌍으로 수행되는 것으로 나타남.

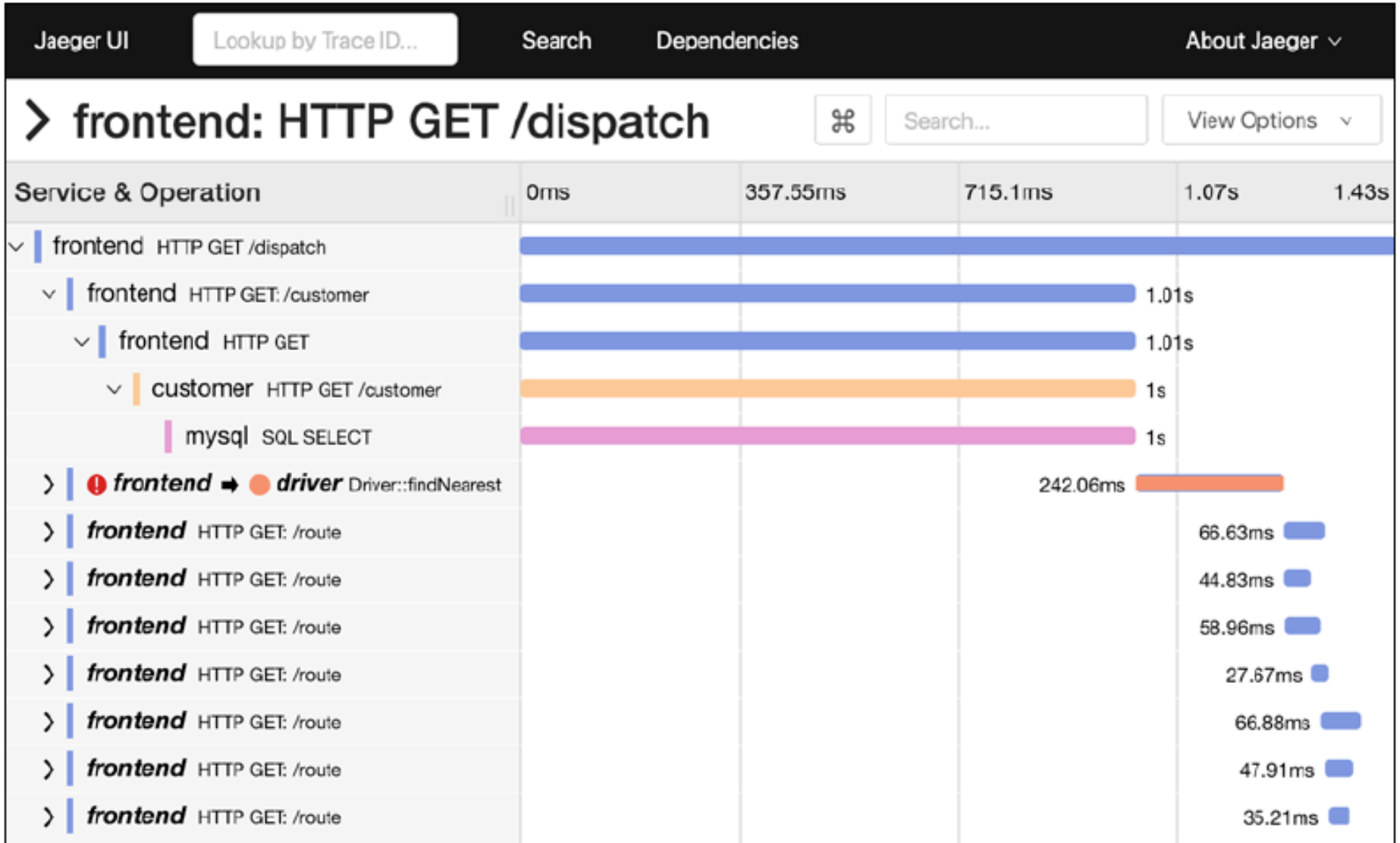




HOTROD

Identifying source of latency

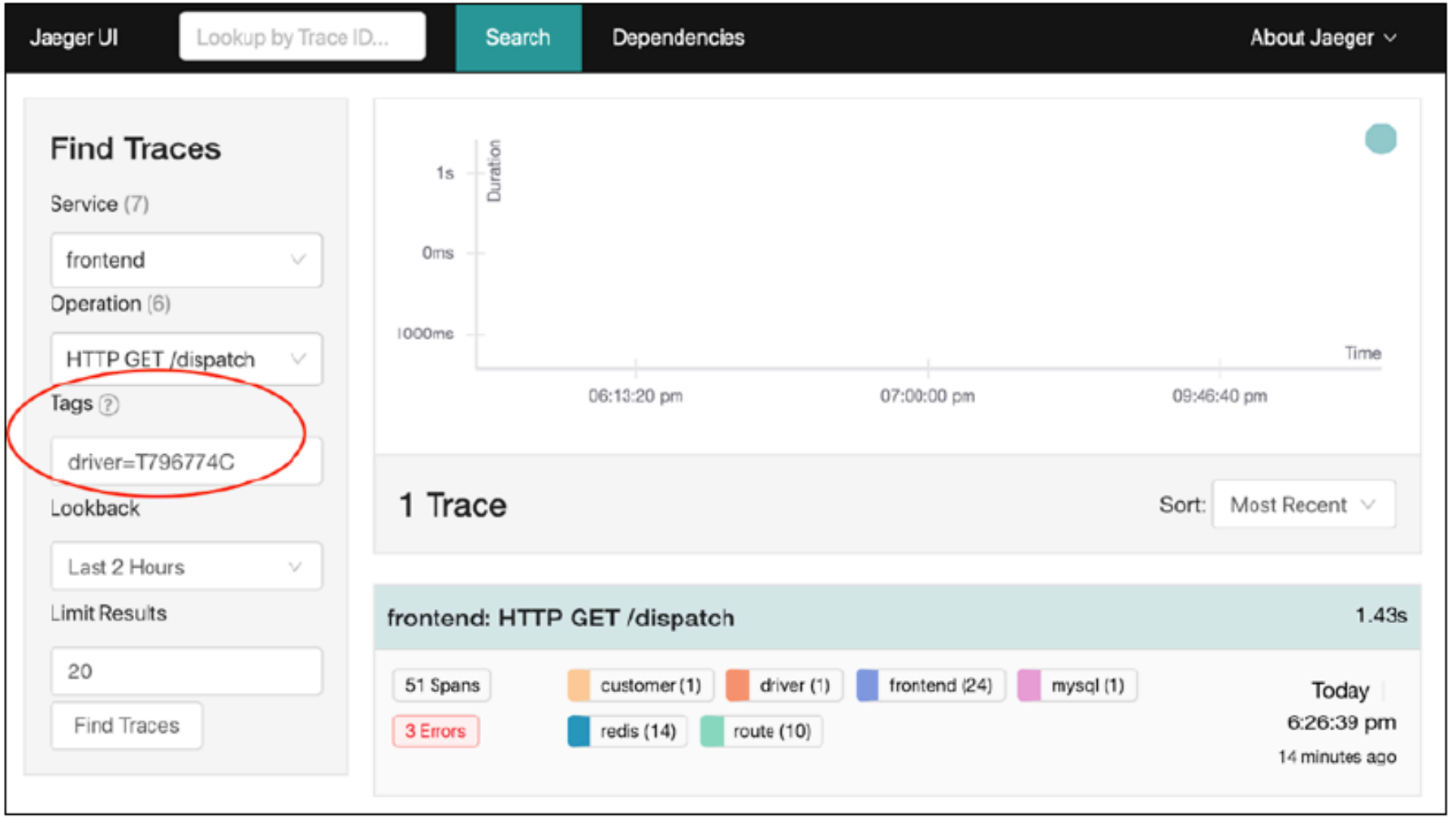
화면에서 같은 사용자로 Request를 증가 시키면 mysql의 수행시간이 1초 이상으로 소요됨 (기존 300ms)



HOTROD

Identifying source of latency

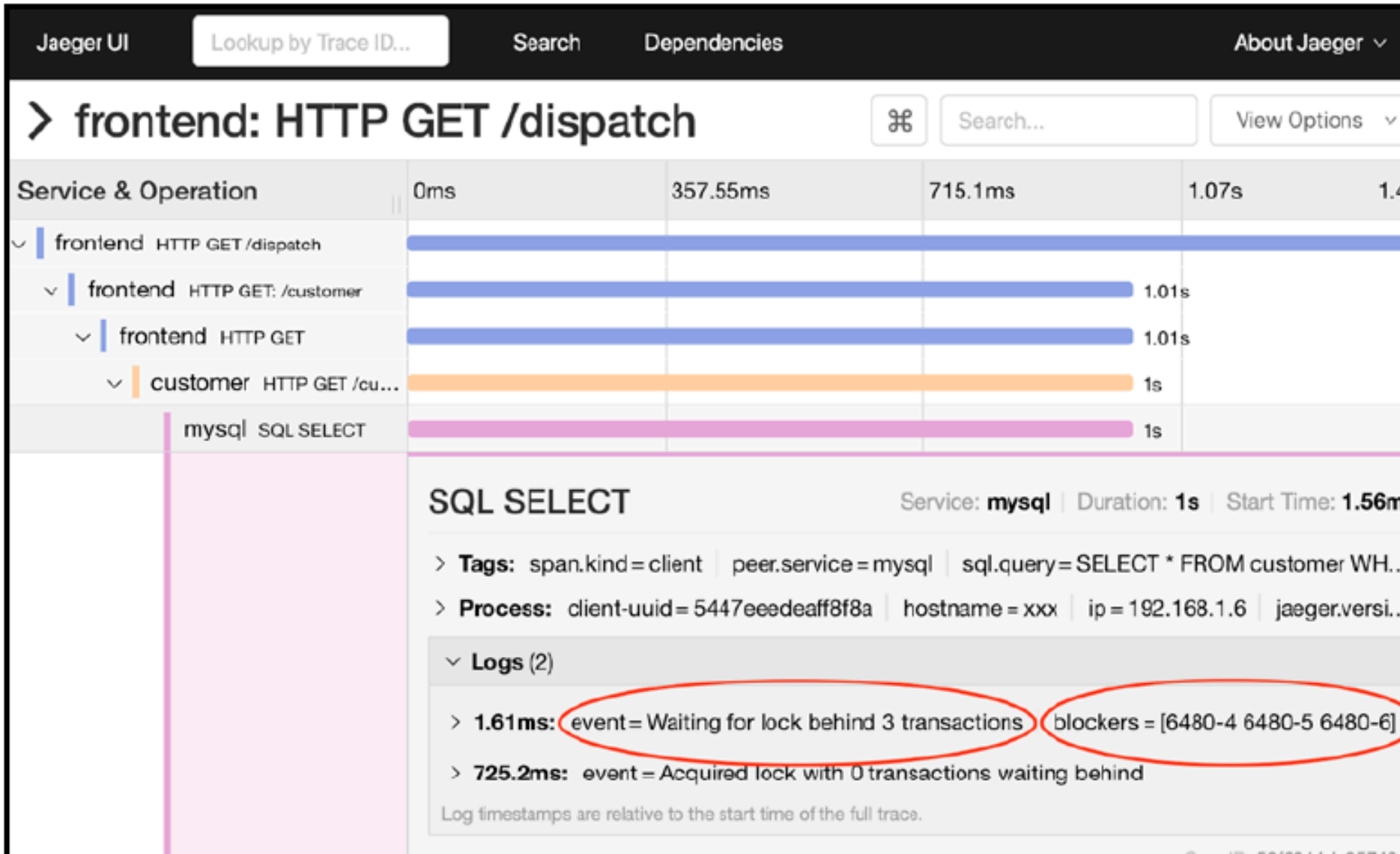
특정 Driver의 Request를 추적



HOTROD

Identifying source of latency

내부적으로 log를 보면 block을 걸고 있는 것을 볼 수 있음



HOTROD

Identifying source of latency

examples/hotrod/services/customer/database.go

```
if !config.MySQLMutexDisabled {
    // simulate misconfigured connection pool that only gives
    // one connection at a time
    d.lock.Lock(ctx)
    defer d.lock.Unlock()
}

// simulate db query delay
delay.Sleep(config.MySQLGetDelay, config.MySQLGetDelayStdDev)
```

소스코드를 보면 “d.lock.Lock”을 확인할 수 있다.

$ ./example-hotrod help

```
Flags:
  -D, --fix-db-query-delay, duration     Average lagency of MySQL DB
                                         query (default 300ms)
  -M, --fix-disable-db-conn-mutex        Disables the mutex guarding
                                         db connection
  -W, --fix-route-worker-pool-size, int  Default worker pool size
                                         (default 3)
```

-M은 lock을 없애는 옵션
-D는 db-query-delay를 300ms에서 100ms으로 줄이는 옵션

$ ./example-hotrod -M -D 100ms all

HOTROD

Identifying source of latency

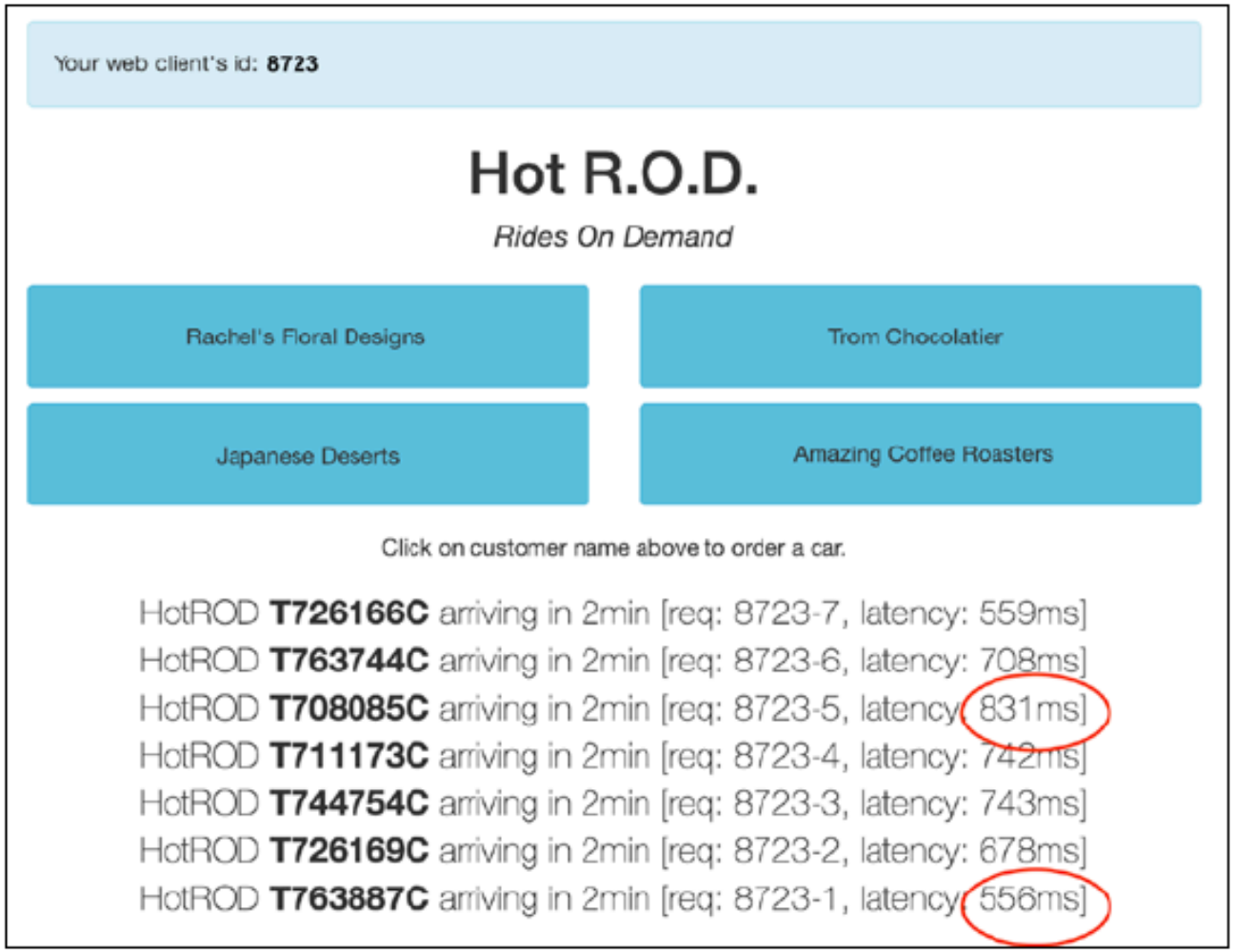
Go를 설치하지 않은 경우는 Docker의 옵션으로 추가하면 된다.

```
$ docker run --rm -it ₩
--link jaeger ₩
-p8080-8083:8080-8083 ₩
jaegertracing/example-hotrod:1.6 ₩
-M -D 100ms all ₩
--jaeger-agent.host-port=jaeger:6831
```

HOTROD

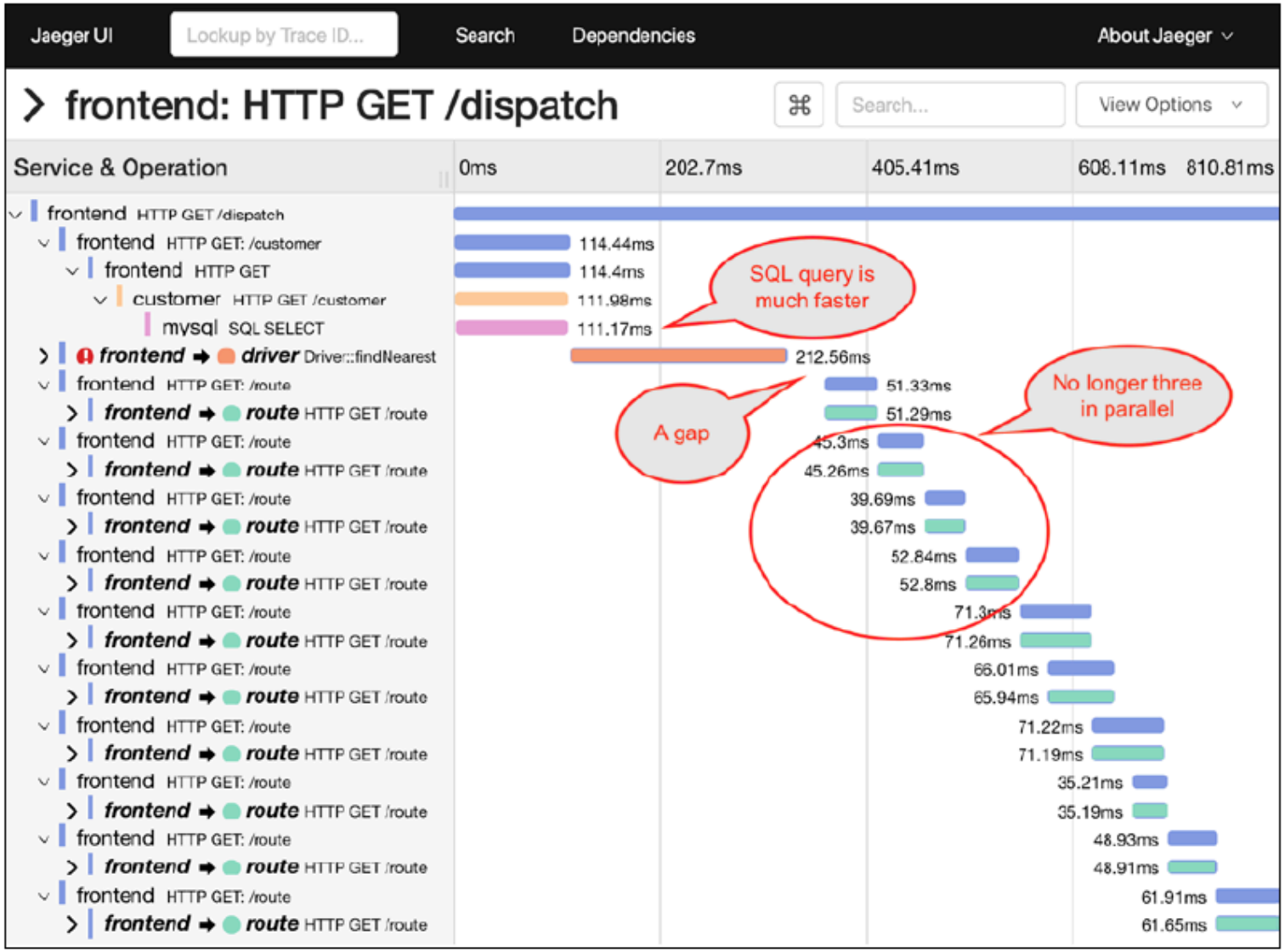
Identifying source of latency

다시 다량의 Request를 발생 시키면 여전히 delay를 확인할 수 있다.



HOTROD

Identifying source of latency



HOTROD

Identifying source of latency

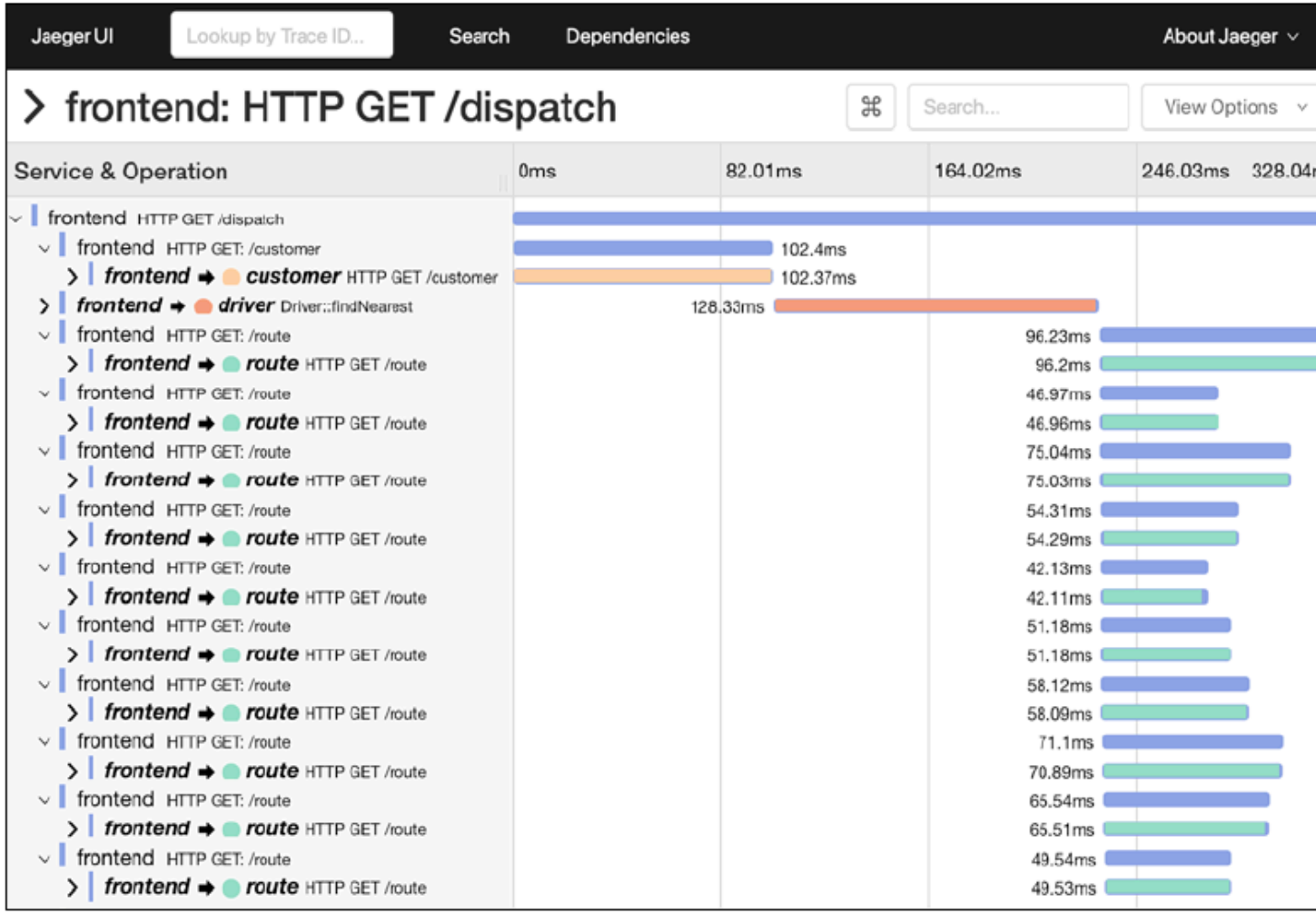
services/config/config.go

```
RouteWorkerPoolSize = 3
```

$ ./example-hotrod -M -D 100ms -W 100 all

W는 WorkerPoolSize이다 Default 3

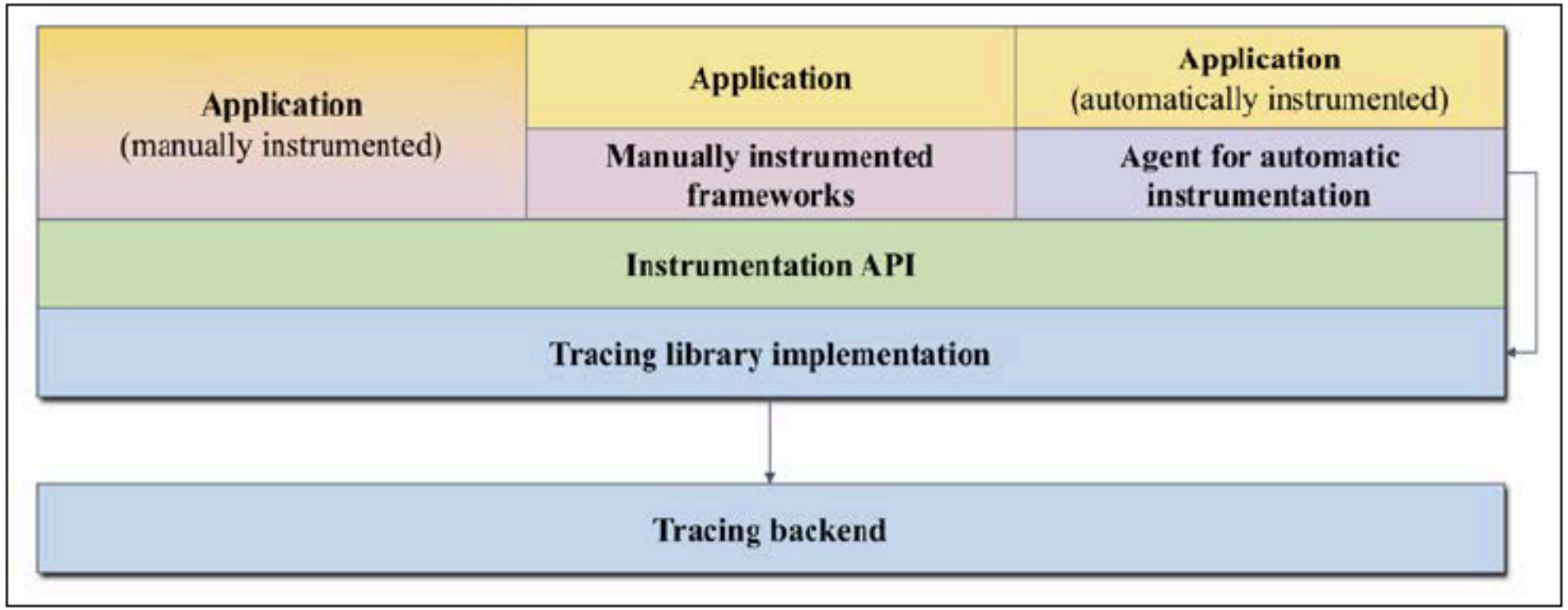
변경후 다량의 Request로 결과 확인



PART 3

Tracing Standards and EcoSystem

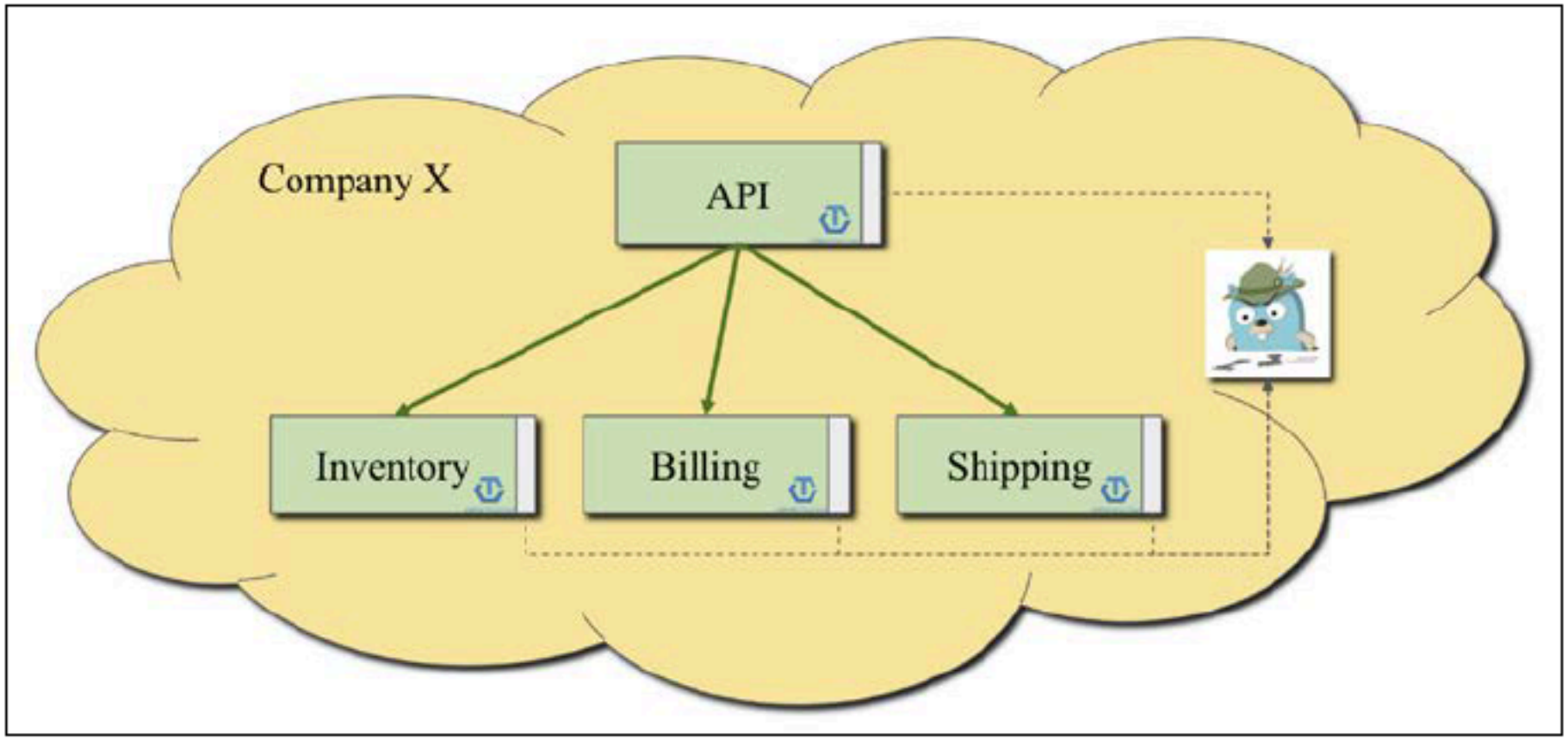
Tracing Instrumentation



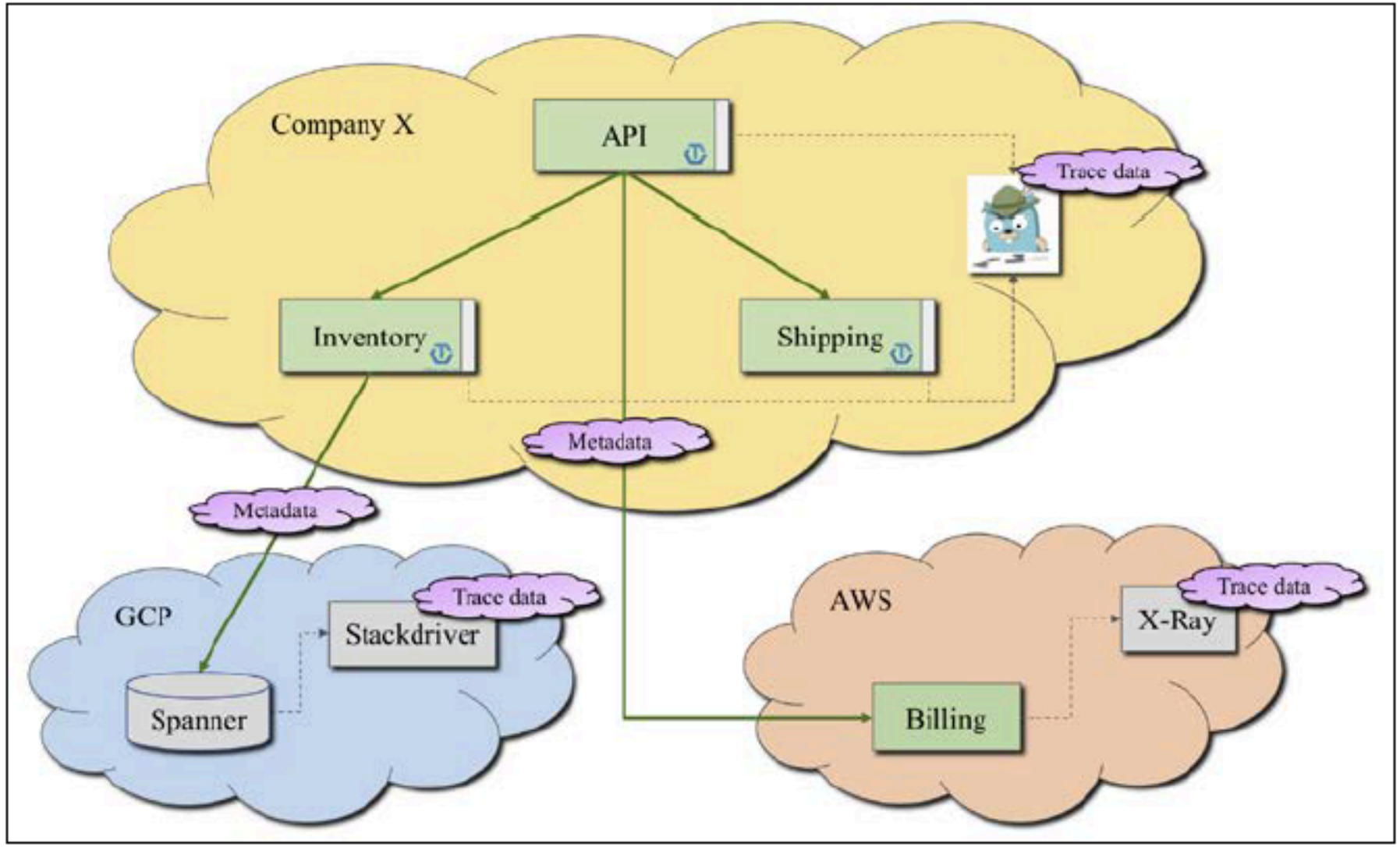
3 Type of tracing instrumentation

- Manual Instrument
- Framework Based Manual Instrument
- Agent Based automatic instrument

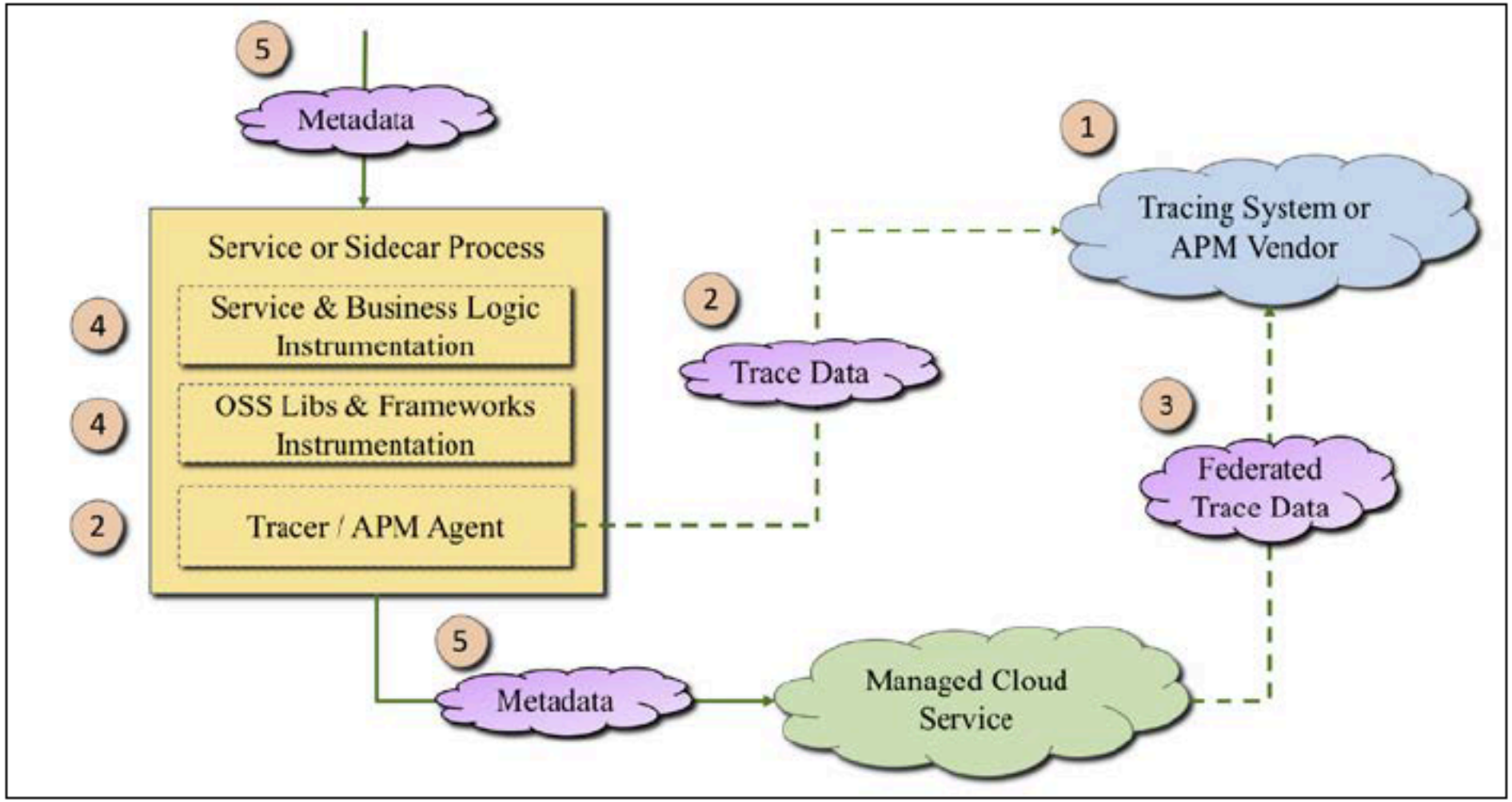
Company X



Company X


Company X
API
Trace data
Inventory
Shipping
Metadata
Metadata
GCP
Trace data
Stackdriver
Spanner
AWS
Trace data
X-Ray
Billing

Different meanings of tracing



Different meanings of tracing: analyzing transactions (1), recording transactions (2), federating transactions (3), describing transactions (4), and correlating transactions (5)

	Analyzing transactions	Recording transactions	Federating transactions	Describing transactions	Correlating transactions
Project	Tracing tool	Tracer/ agent	Trace data	App/OSS instrumentation	Metadata
Zipkin	✔	✔	✔	✔	✔
Jaeger	✔	✔	✔		✔
SkyWalking	✔	✔	✔	✔	✔
Stackdriver, X-Ray, and so on	✔	✔	✔	✔	✔
W3C Trace Context					✔
W3C "Data Interchange Format"			✔		
OpenCensus		✔	✔	✔	✔
OpenTracing				✔	

Tracing Systems

Zipkin and OpenZipkin

- 최초의 Open Source Tracing system
- 2012년, Twitter 공개
- 많은 지원 생태계 구성
- 자체 Tracer인 Brave API 기반으로 다양한 fraework가 이를 지원 (ex: Spring, Spark, Kafka, gRPC등)
- 다른 tracing System과는 data-format level에서 연동

Jaeger

- 2015년 Uber에서 만들어졌고, 2017년 Open Source Project 로 공개
- CNCF산하의 프로젝트로 OpenTracing에 집중하면서 많은 주목을 받고 있음
- Zipkin과의 호환 (B3 metadata format)
- 직접적인 instrumentation을 제공하지 않으며 OpenTracing-compatible tracer를 이용

Tracing Systems

SkyWalking

- 상대적으로 최신 프로젝트로 중국에서 시작
- 2017년 Apache Foundation 의 incubation project
- Tracing 시스템으로 시작해서 full-blown APM solution으로 진화중
- 일부 OpenTracing-compatible(only Java)
- 중국에서 많이 사용되는 framework에 대한 Agent-based instrumentation에 집중

X-Ray, Stackdriver

- 클라우드 사업자의 Managed 서비스
- 기존에 Zipkin이나 Jaeger를 사용하던 On-Prem과의 연계가 문제가 됨
- 따라서 클라우드 사업자들이 data 표준화에 많은 투자를 하기 시작함

Tracing Systems

W3C Trace Context

- 2017년 World Wide Web Consortium(W3C) 산하 Distributed Tracing Working Group 발족
- 다양한 vendors, cloud providers, open source projects 참여
- Trace tool간의 interoperability를 제공하기 위한 표준 정의
- Trace Context :Tracing metadata를 주고받을 때 사용할 format에만 focus
- 2018년 Trace ID conceptual model 합의
- 2개의 protocol header 제안

1. Traceparent
   spandID : 16 byte and 8 byte array
   Traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01

2. Tracestate
   tracing vendor specific 정보 저장 영역
   Tracestate: vendorname1=opaqueValue1,vendorname2=opaqueValue2
3. data 표준화에 많은 투자를 하기 시작함

Tracing Systems

W3C "Data Interchange Format"

- 데이터를 주고 받을때 사용할 수 있는 format를 정의
- Vendor마다의 이해가 달라 진전이 없는 상태

OpenCensus

- Google의 Census라는 library의 일부로 시작한 프로젝트
- Misssion : "vendor-agnostic single distribution of libraries to provide metrics collection and tracing for your services."
- 기존 Google의 Dapper format 을 그래도 사용
- Google Stackdrivre와 Zipkin, Jaeger의 Metadata와 일치 (Dapper의 후손)
- 그렇지만 다른 APM Vendor와는 많이 다름

Tracing Systems

OpenTracing

- Goal of OpenTracing Project

- To provide a single API that application and framework developers can use to instrument their code
- To enable truly reusable, portable, and composable instrumentation, especially for the other open source frameworks, libraries, and systems
- To empower other developers, not just those working on the tracing systems, to write instrumentation for their software and be comfortable that it will be compatible with other modules in the same application that are instrumented with OpenTracing, while not being bound to a specific tracing vendor

Understanding Sampling

- Tracing data > than business traffic
- Most tracing systems sample transactions
- **Head-based sampling**: the sampling decision is made just before the trace is started, and it is respected by all nodes in the graph
- **Tail-based sampling**: the sampling decision is made after the trace is completed / collected
  - Head-based sampling don't catch 0.001 probability anomalous behavior.

OpenTelemetry

OpenTelemetry is an effort to combine all three verticals into a single set of system components and language-specific telemetry libraries. It is meant to replace both the **OpenTracing** project, which focused exclusively on tracing, and the **OpenCensus** project, which focused on tracing and metrics.

OpenTelemetry will not initially support logging, though we aim to incorporate this over time.

OpenTelemetry

Timeline

Milestone	Date
Project launch	May 2019
Reference candidate released (Java)*	June 2019
Language libraries released (Golang, Python, NodeJS, C#)*	September 2019
Sunsetting of the OpenCensus and OpenTracing libraries*	November 2019
OpenTelemetry at KubeCon San Diego*	November 2019

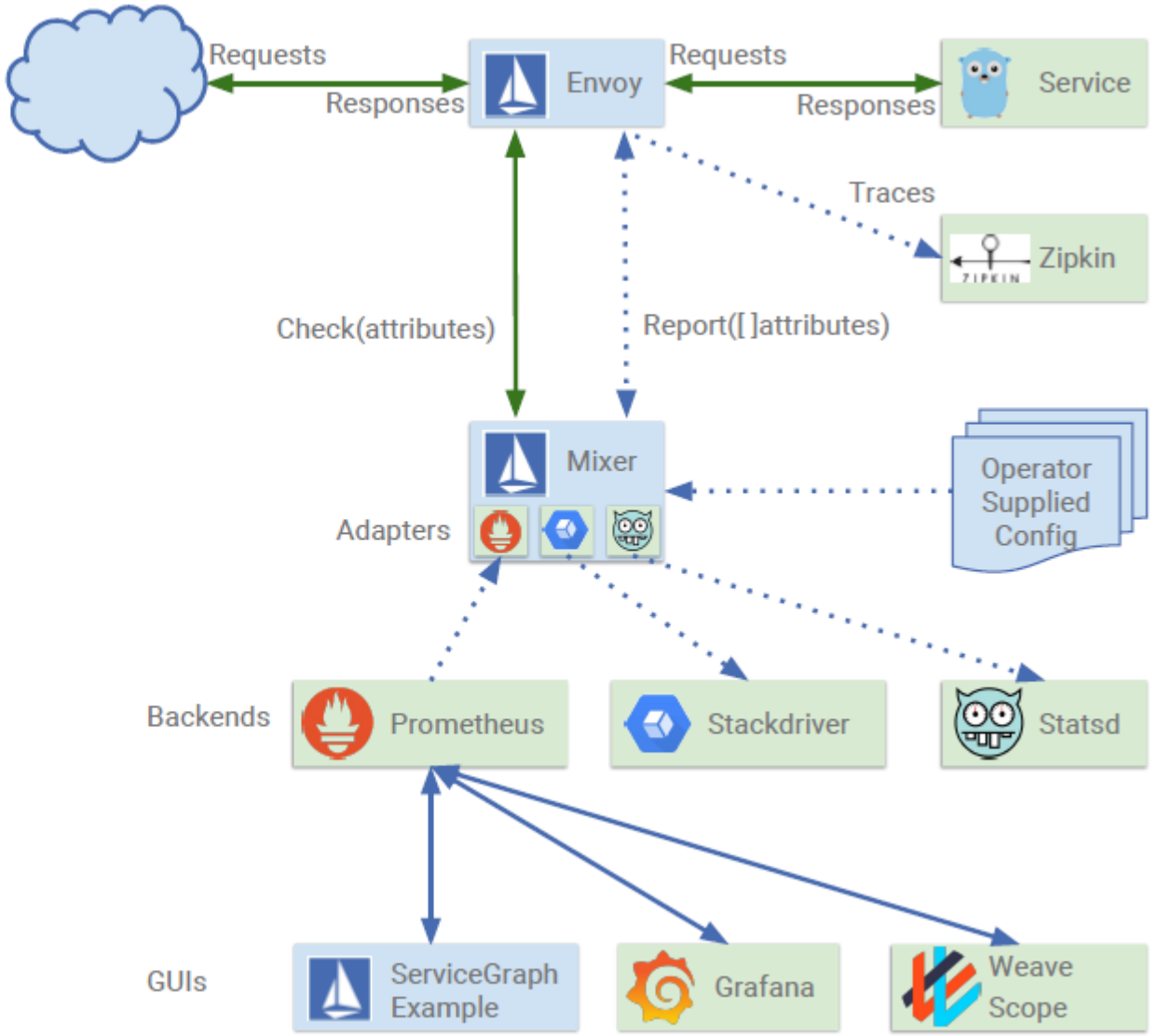
* Dates are subject to change based on community participation and other factors


Container Based Observability
To be Continued

Istio – Visibility

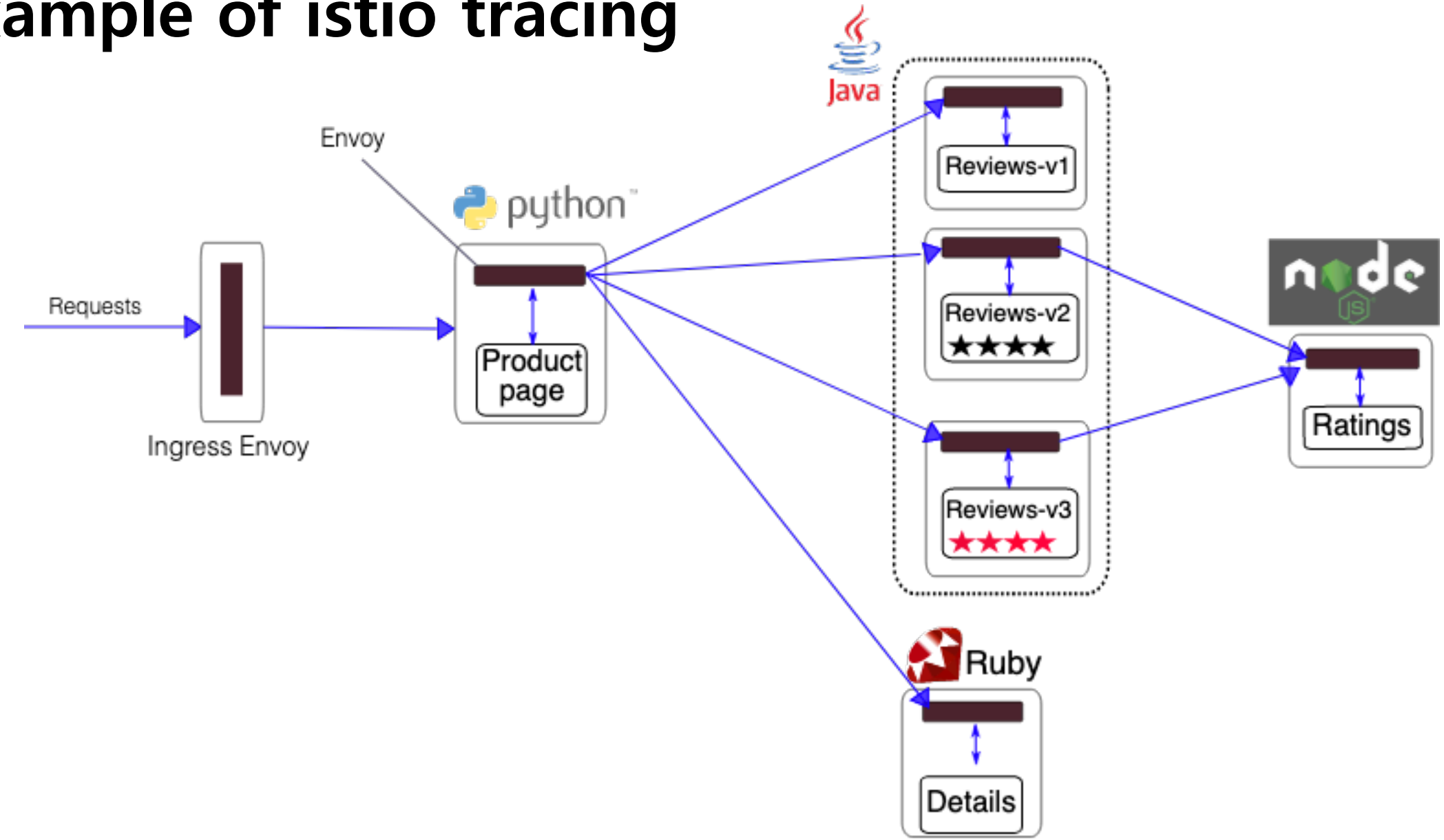
Visibility: Tracing

- Applications do not have to deal with generating spans or correlating causality
- Envoys generate spans
  - Applications need to forward context headers on outbound calls
- Envoy sends traces to Mixer
- Adapters at Mixer send traces to respective backends



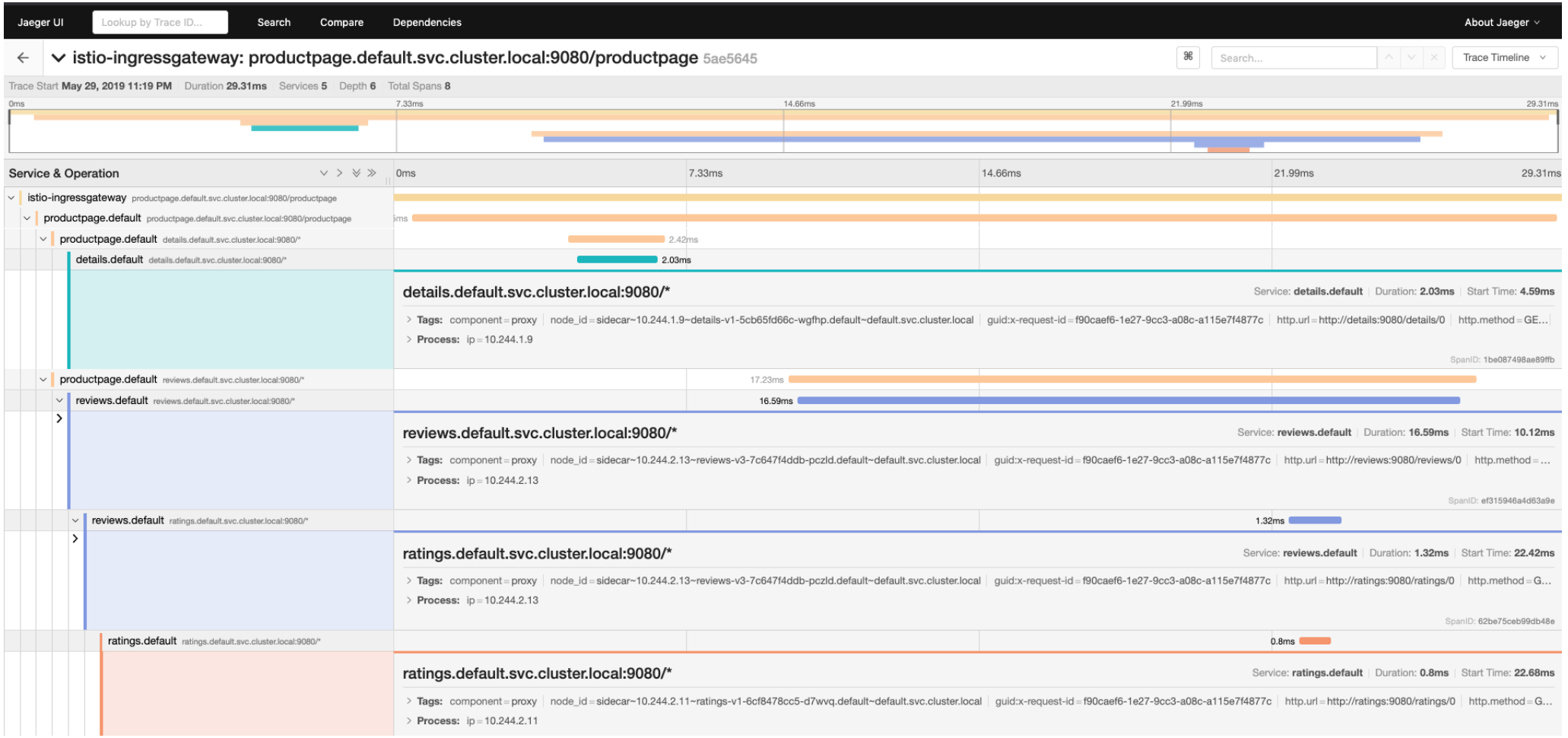
Reference: https://istio.io/talks/istio_talk_gluecon_2017.pdf

Example of istio tracing



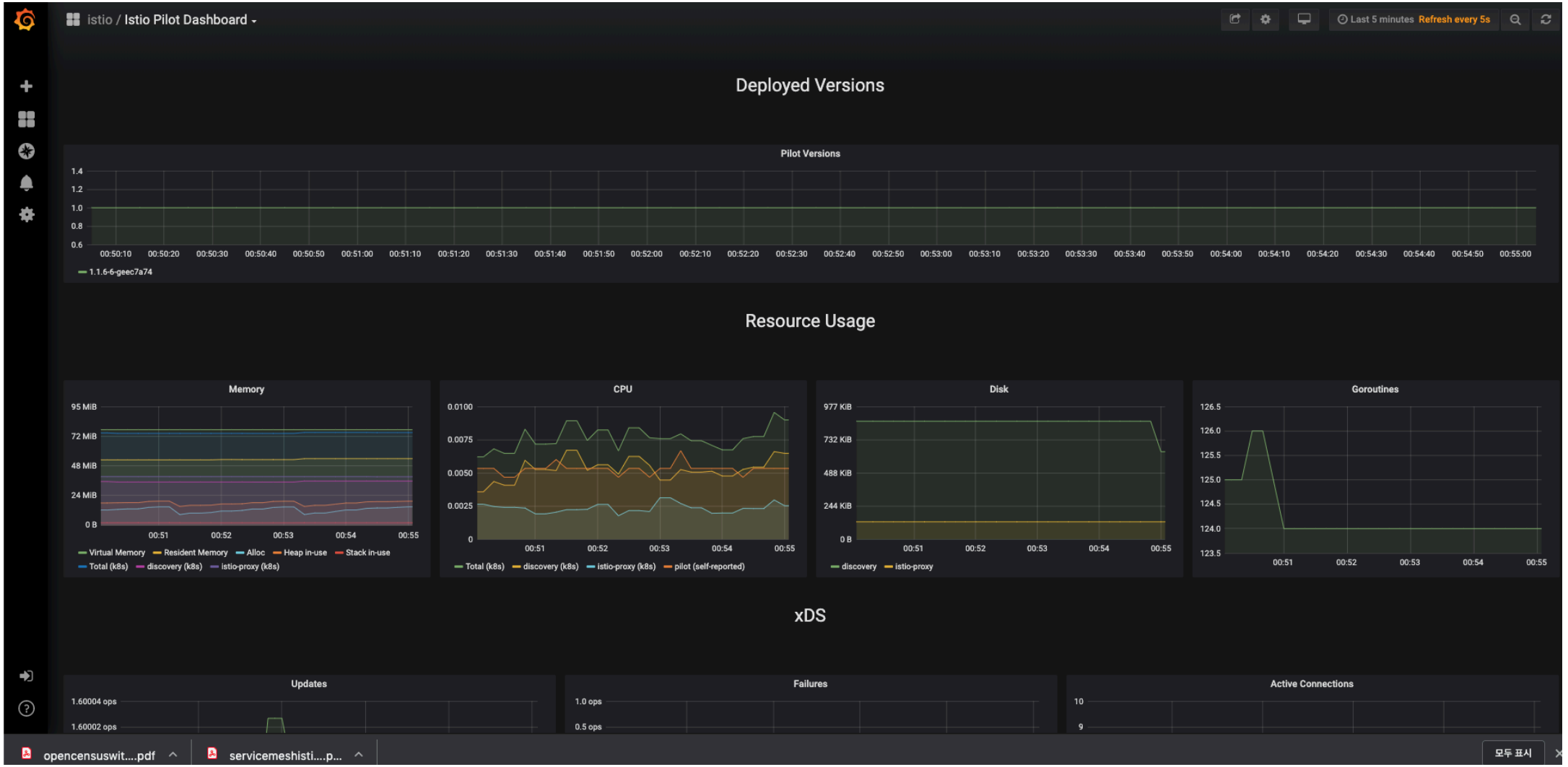
Bookinfo Application with Istio

Example of istio tracing



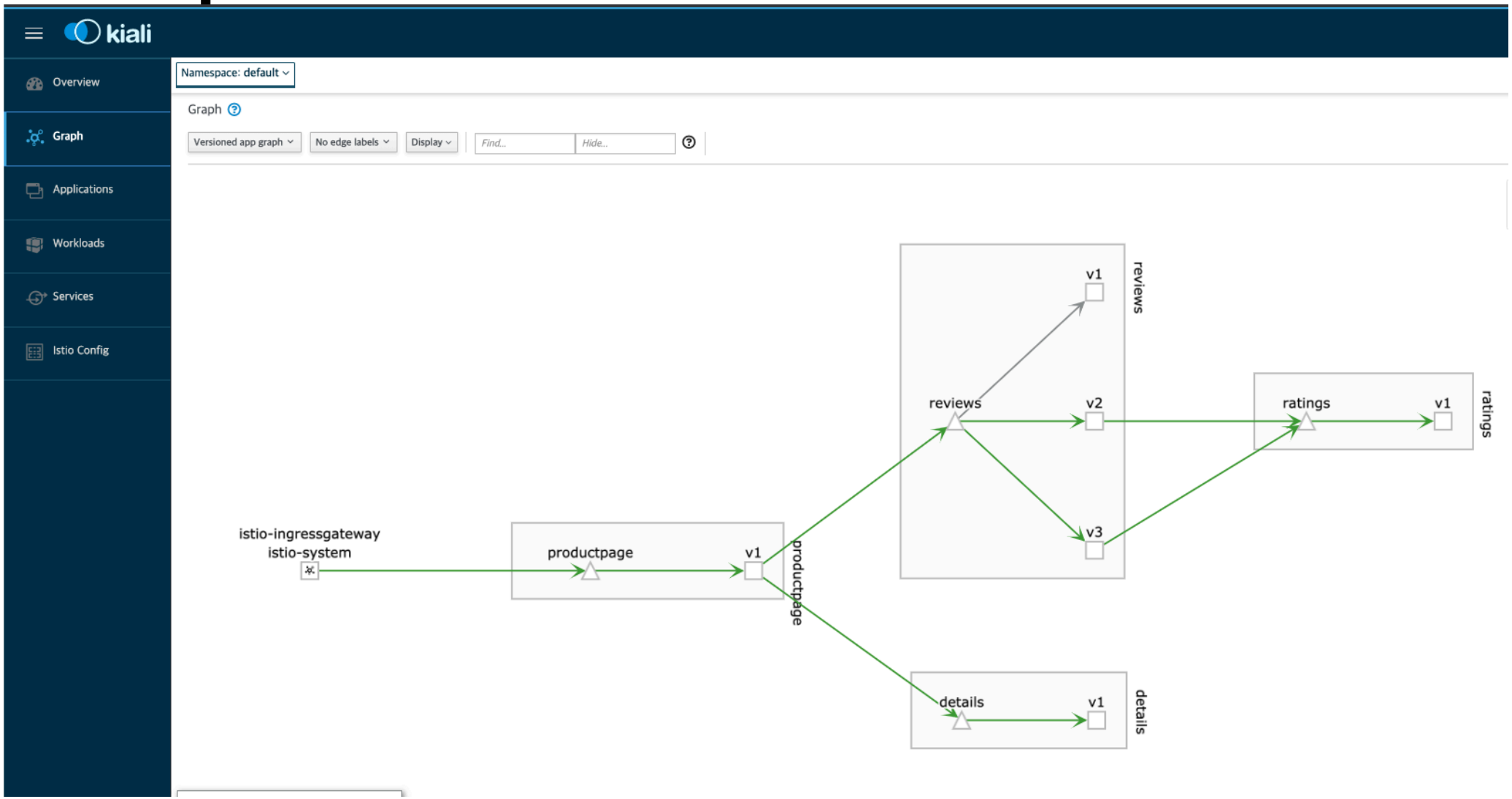
kubectl port-forward -n istio-system $(kubectl get pod -n istio-system -l app=jaeger -o jsonpath='{.items[0].metadata.name}') 16686:16686 &

Example of istio Metics



kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l app=grafana -o jsonpath='{.items[0].metadata.name}') 3000:3000 &

Example of Istio service



kubectl -n istio-system port-forward $(kubectl -n istio-system get pod -l app=kiali -o jsonpath='{.items[0].metadata.name}') 20001:20001

References

- Mastering Distributed Tracing
- Alibaba Cloud – OpenTracing implementation of Jaeger : https://www.alibabacloud.com/help/doc-detail/68035.htm
- OpenTracing Best Practices : https://opentracing.io/docs/best-practices/instrumenting-your-application/
- OpenTracing HOTROD : https://medium.com/opentracing/take-opentracing-for-a-hotrod-ride-f6e3141f7941


Thank You