Department of Information Engineering, Computer Science and Mathematics
University of L'Aquila, Italy

Exploiting Architecture/Runtime Model-driven Traceability for Performance Improvement

Vittorio Cortellessa, Daniele Di Pompeo, Romina Eramo, Michele Tucci

{name.surname}@univaq.it

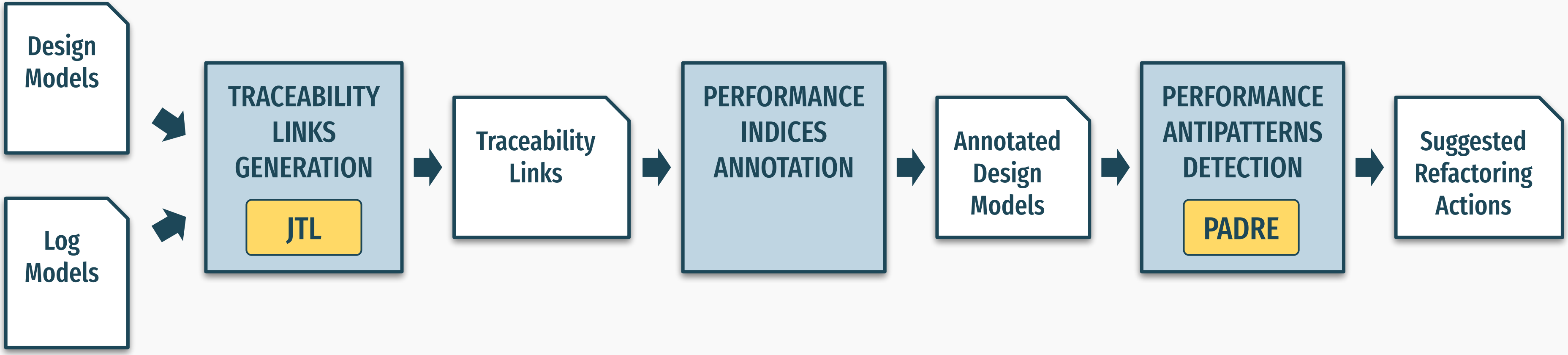
Introduction

- Software architectures are growing in **complexity and heterogeneity**
- Model-Driven Engineering (MDE) has shown to be effective in **managing complexity by introducing automation** at a higher level of abstraction
- Vision: exploiting design-runtime relationships to **detect software problems and deduce improvement actions** (e.g., to meet new (non-)functional requirements)
- A major challenge is to achieve an efficient **integration between design and runtime** aspects of systems
- MDE techniques can support the development of complex systems by **managing relationships between a running system and its architectural models**

Overview of the approach

- A Model-driven approach that exploits design/runtime interactions to support designers in :
  - Performance analysis
  - Architectural refactoring

- The process underlying the approach:



JTL: Janus Transformation Language

Eclipse EMF-based model transformation tool tailored to support **bidirectionality** and **change propagation** and to keep **traceability** during software design.

- Generation of traceability links between **heterogeneous software/runtime models**
- Storage of links in an explicit way by means of **traceability models**
- **Propagation of feedback** obtained from the tracing analysis back to the software models

jtl.univaq.it

PADRE: Performance Antipatterns Detection and model REfactoring

Eclipse-based framework that enables **performance antipatterns detection** on UML-MARTE software models and **model refactoring** based on detection results.

- A **performance antipattern** describes those bad practices in software designing that might introduce performance degradation into the system.
- User-driven **refactoring** of UML-MARTE software design models, **driven by performance antipatterns detection**

git.io/SeaLabAQ-padre

The *E-Shopper* case study

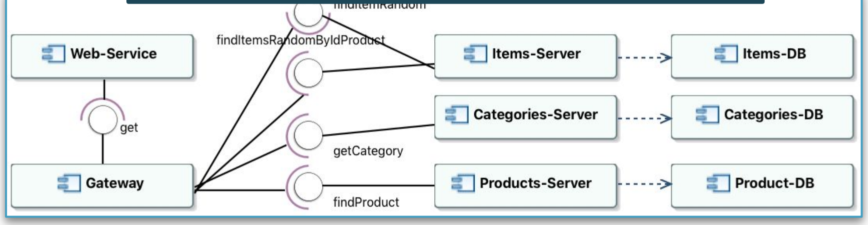
- Open source e-commerce web application **based on microservices**
- 9 application microservices, 8 databases, 4 infrastructure microservices, 42 API endpoints
- **Designed in UML** (Component, Deployment and Sequence Diagrams)
- Developed using the **Spring Cloud framework**
- Deployed on **Docker**

Available at:
git.io/fh9Z8

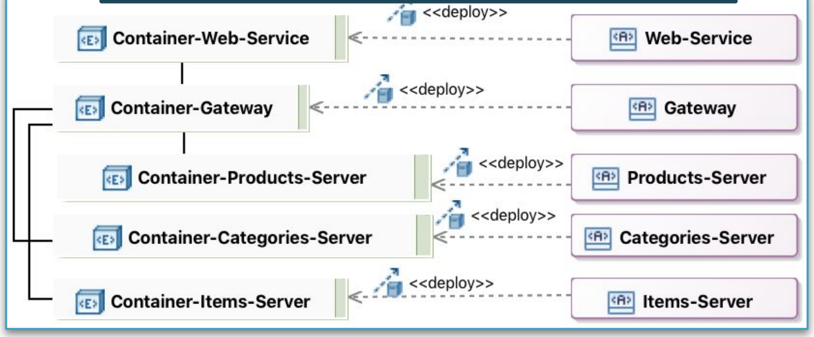
UML design: excerpt of the home page scenario

The approach requires three different design views

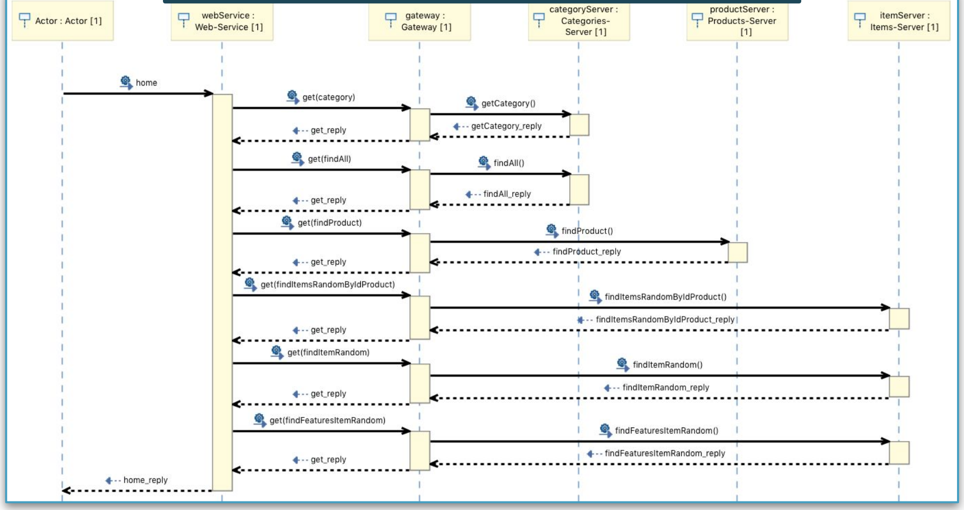
Static view: **COMPONENT DIAGRAM**



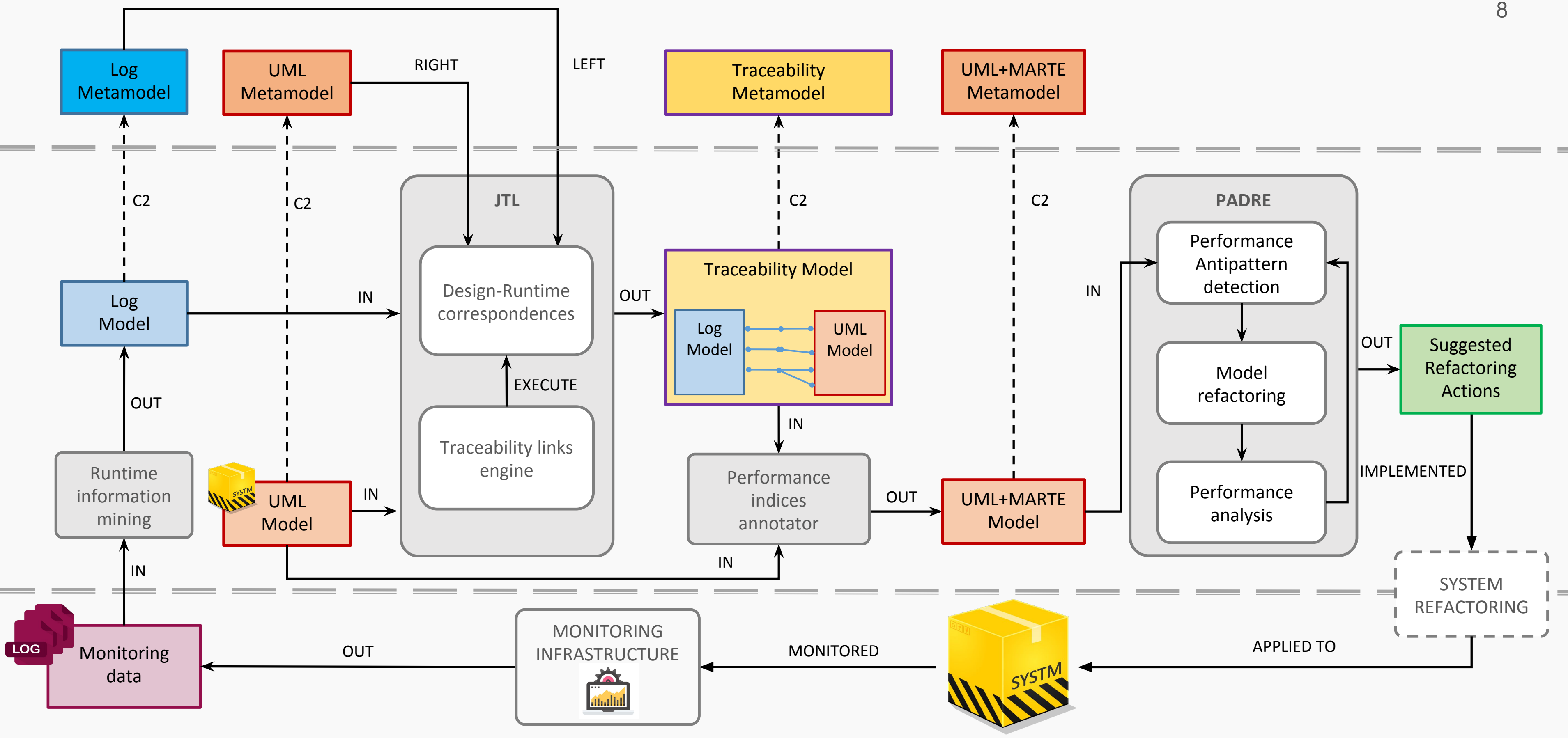
Deployment view: **DEPLOYMENT DIAGRAM**



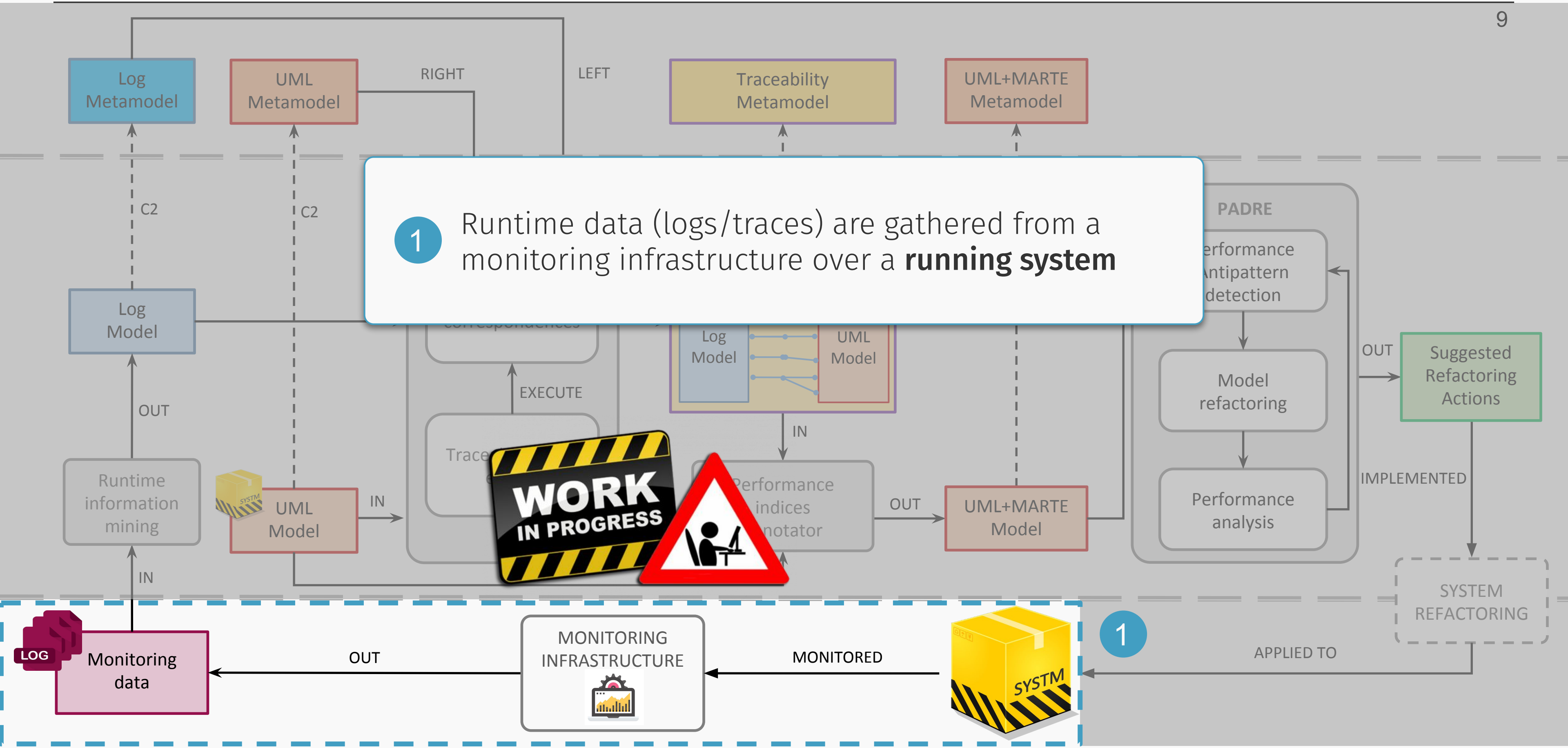
Dynamic view: **SEQUENCE DIAGRAM**



Overall approach



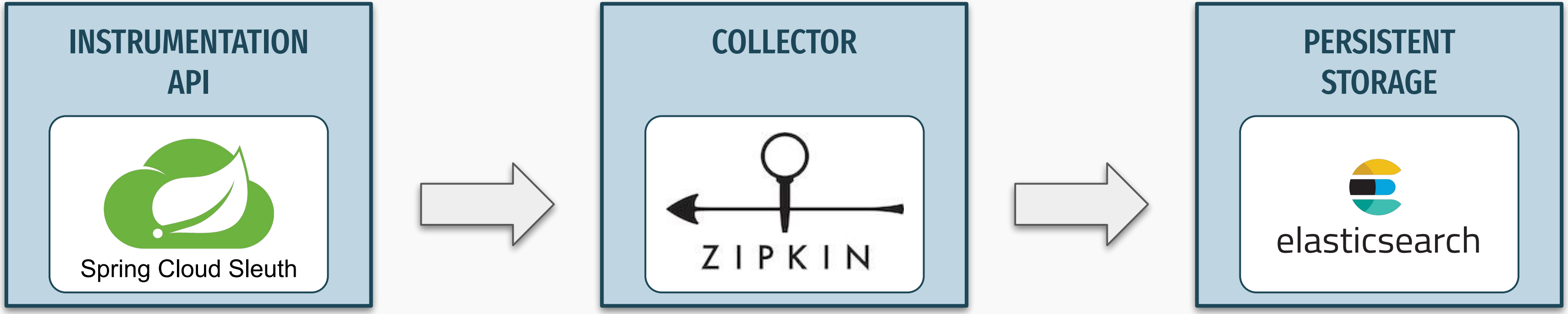
Runtime information mining



1 Runtime data (logs/traces) are gathered from a monitoring infrastructure over a **running system**

Distributed tracing

Method used to profile and monitor applications,
especially those built using a microservices architecture

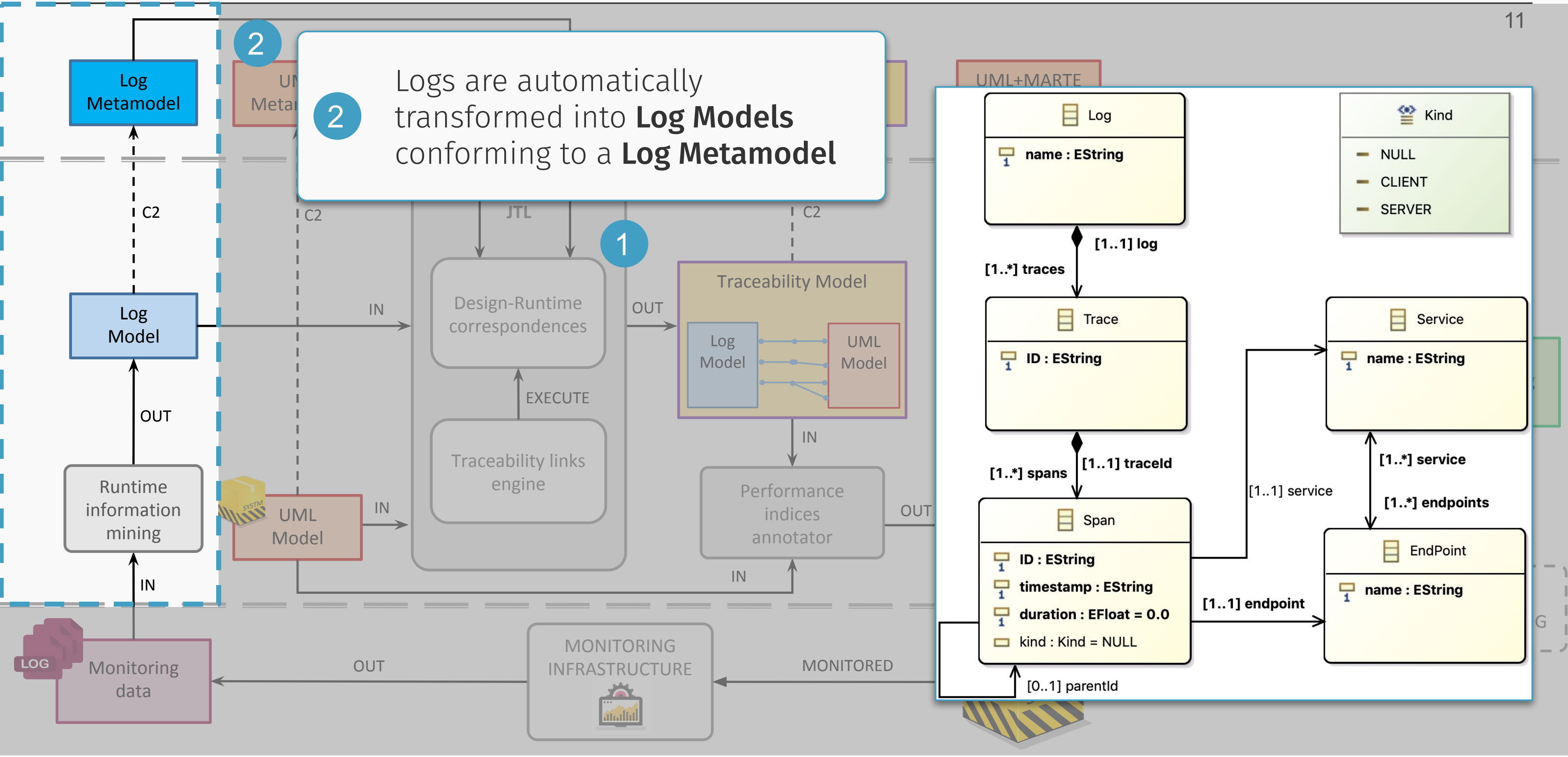


RAW LOG

Time	_source
November 20th 2018, 10:52:48.107	traceId: 149c4cef3ac7f19f duration: 27,000 shared: true localEndpoint.serviceName: gateway localEndpoint.ipv4: 172.28.0.12 localEndpoint.port: 4000 timestamp_millis: November 20th 2018, 10:52:48.107 kind: SERVER name: http:/categories/category id: 16bb4e7b689f807a parentId: 149c4cef3ac7f19f timestamp: 1,542,707,568,107,000 tags.spring.instance_id: 002ffdb287d6:gateway:4000 _id: cE-JMGcBBzL8qQlHYHn4 _type: span _index: zipkin:span-2018-11-20 _score: -
November 20th 2018, 10:52:48.115	traceId: 149c4cef3ac7f19f duration: 17,000 shared: true localEndpoint.serviceName: categories-server localEndpoint.ipv4: 172.28.0.18 localEndpoint.port: 5555 timestamp_millis: November 20th 2018, 10:52:48.115 kind: SERVER name: http:/categories/category id: 4ad2da86e8767b82 parentId: 16bb4e7b689f807a timestamp: 1,542,707,568,115,000 tags.mvc.controller.class: CategoriesController tags.mvc.controller.method: getCategory tags.spring.instance_id: 5b58aea6835e:categories-server:5555 _id: bk-JMGcBBzL8qQlHYHn2 _type: span _index: zipkin:span-2018-11-20 _score: -

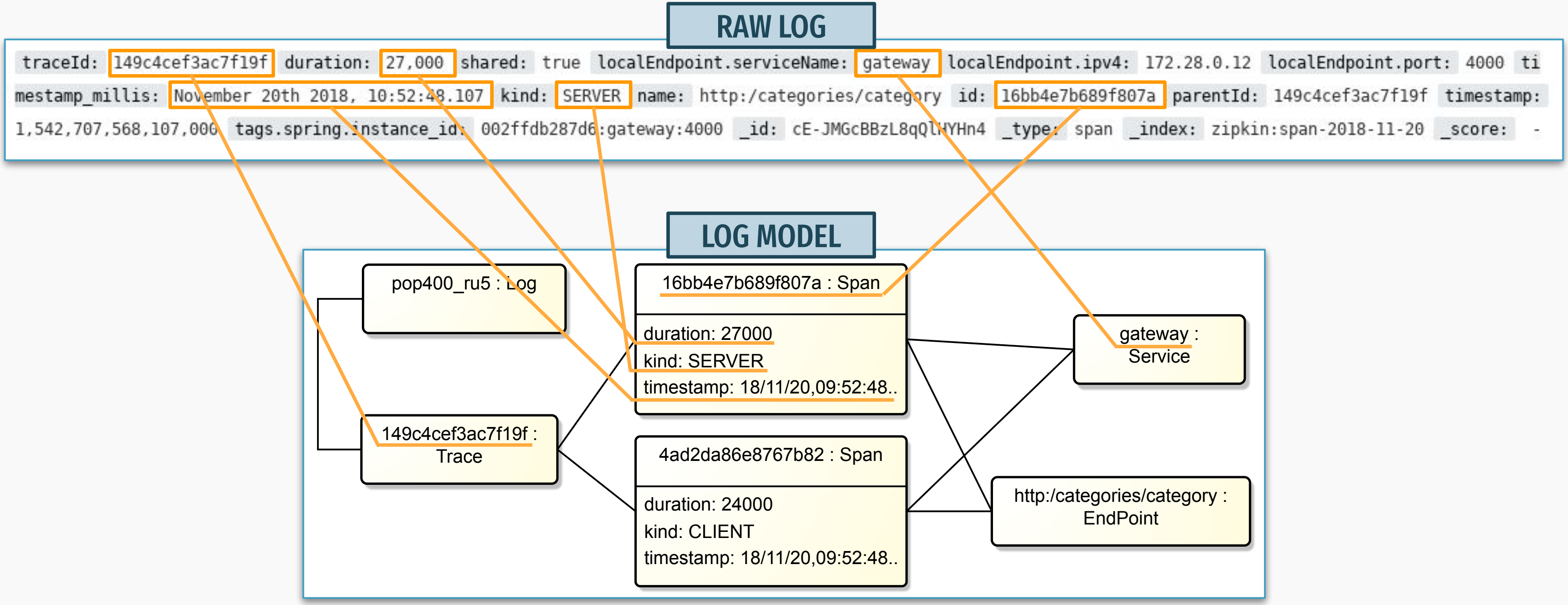
Runtime information mining

2 Logs are automatically transformed into **Log Models** conforming to a **Log Metamodel**

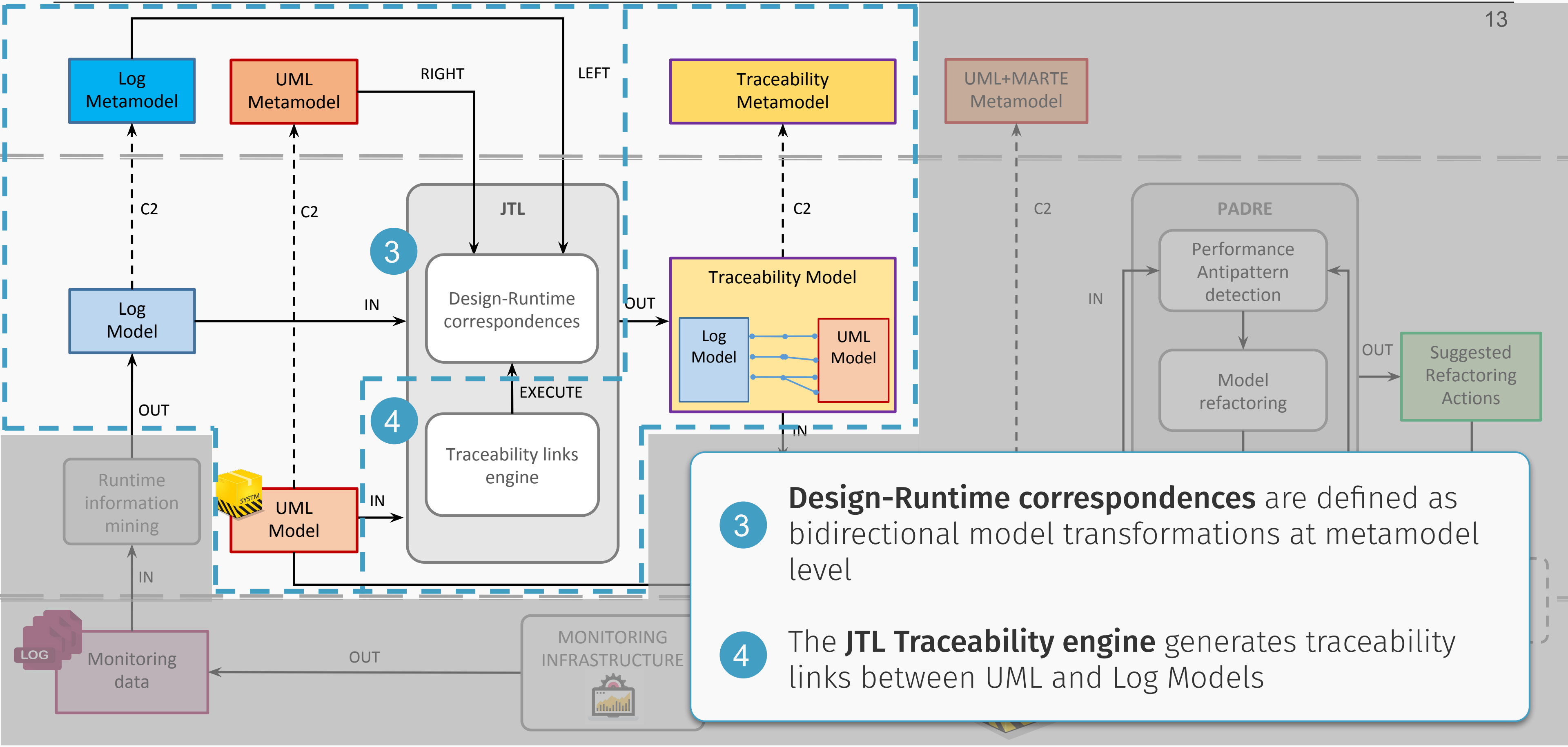


From raw logs to models

A Java transformation automatically generates
Log Models (serialized in XMI) from raw logs



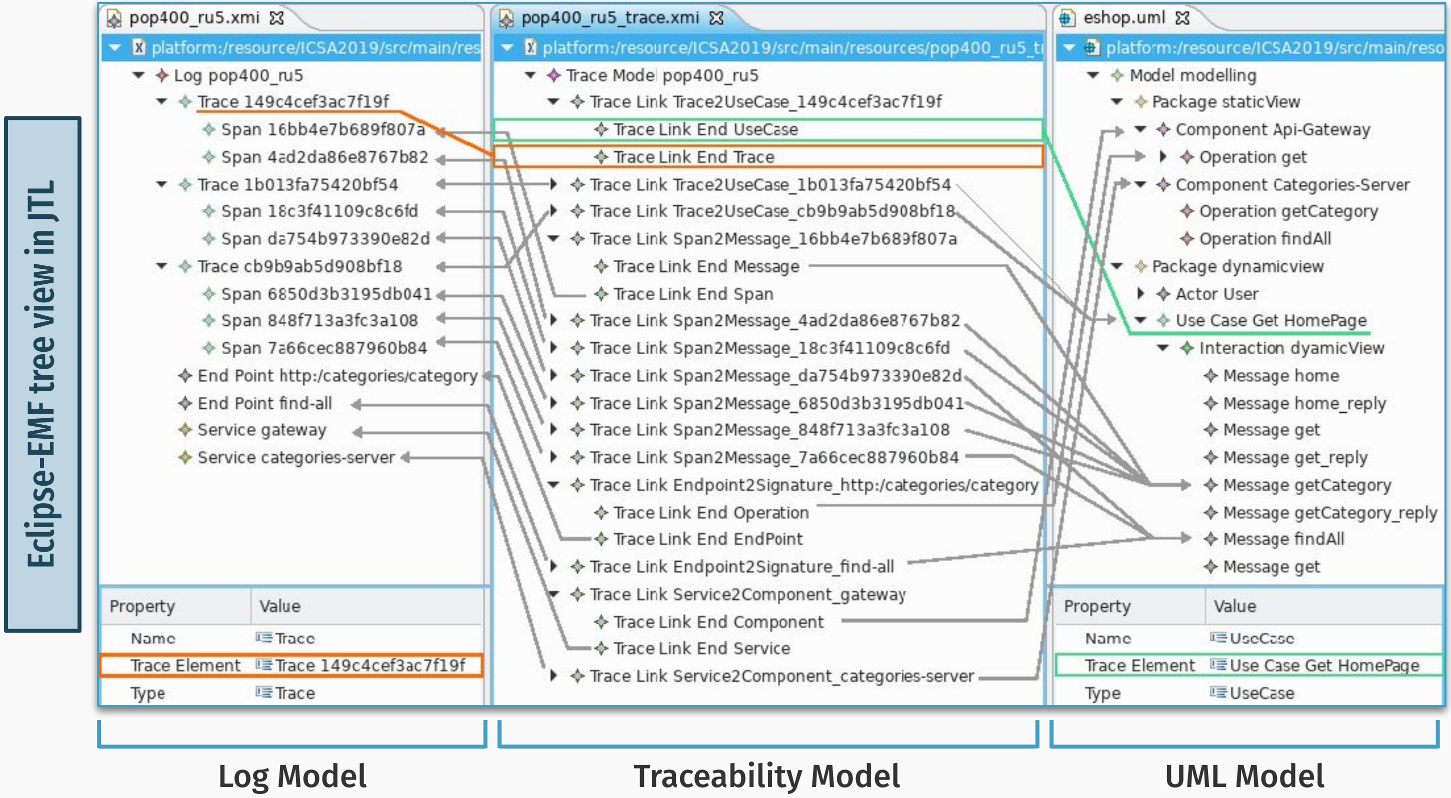
Design-Runtime traceability with JTL



3 **Design-Runtime correspondences** are defined as bidirectional model transformations at metamodel level

4 The **JTL Traceability engine** generates traceability links between UML and Log Models

Traceability model between the Log and UML models



Log2UML correspondences specification


```
transformation Log2UML (log:Log, uml:UML) {
  ...
  top relation Trace2UseCase {
    checkonly domain log t : Log::Trace {
      spans = s : Log::Span {    }
    };
    checkonly domain uml uc : UML::UseCase {
      ownedBehavior = ob : UML::Interaction {
        message = m : UML::Message { }
      }
    };
    where { Span2Message(s, m); }
  }
  relation Span2Message {
    checkonly domain log s : Log::Span {
      endpoint = ep : Log::EndPoint { }
    };
    checkonly domain uml m : UML::Message {
      signature = s : UML::Operation { }
    };
    where { EndPoint2Signature(ep, s); }
  }
  relation EndPoint2Signature {
    n : String;
    checkonly domain log ep : Log::EndPoint {
      name = n
    };
    checkonly domain uml s : UML::Operation {
      name = n
    };
  }
  top relation Service2Component {
    n : String;
    checkonly domain log s : Log::Service {
      name = n
    };
    checkonly domain uml c : UML::Component {
      name = n
    };
  }
  ...
}
```

Map a **Trace** element in the **Log domain** to a **UseCase** element in the **UML domain**. The **where** clause invokes the execution of the **Span2Message** relation

Map a **Log Span** to a **UML Message** inside an Interaction. The **where** clause invokes the execution of the **EndPoint2Signature** relation

A **JTL transformation** defined between **Log** and **UML**

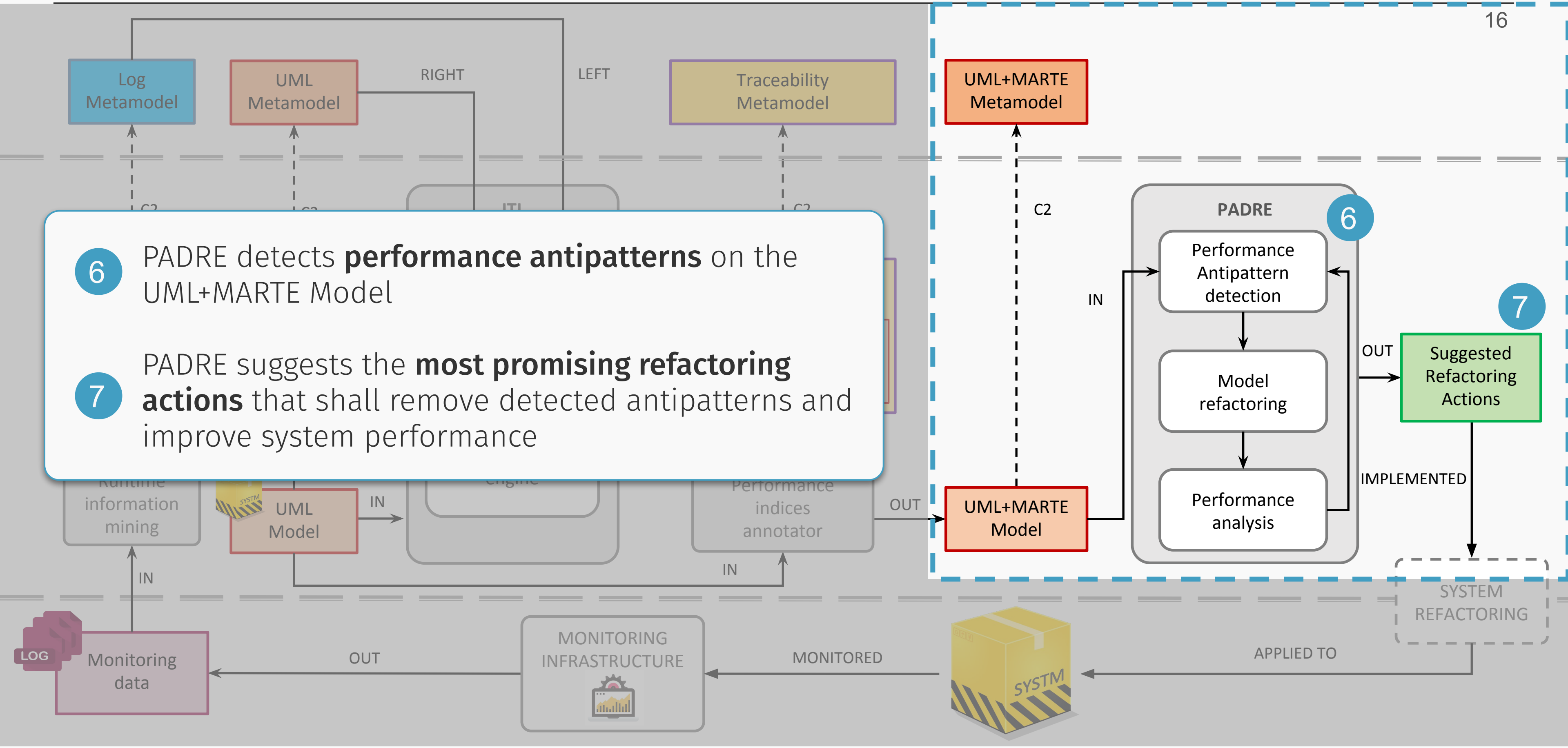
Map a **Log EndPoint** of a Span to a **UML Operation** by matching names. The UML Operation must be referenced in the signature of the Message

Map a **Log Service** to a **UML Component** by matching names

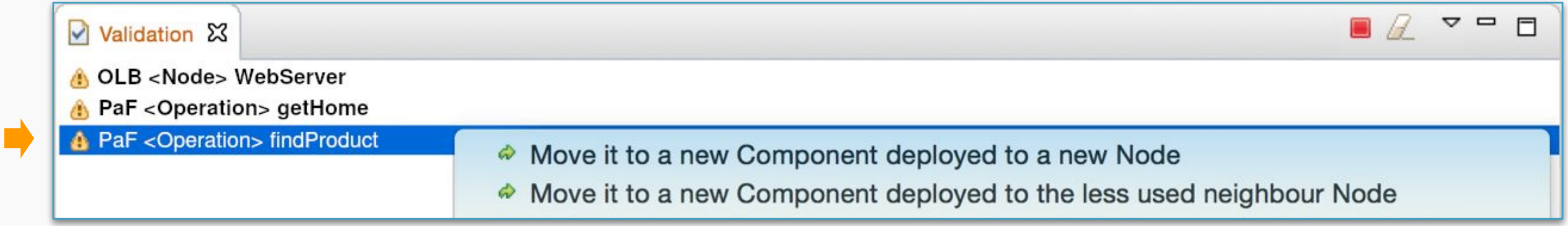
Performance analysis and refactoring with PADRE

6 PADRE detects **performance antipatterns** on the UML+MARTE Model

7 PADRE suggests the **most promising refactoring actions** that shall remove detected antipatterns and improve system performance



Promising refactoring actions - Running Example (1/2)



- PADRE suggests to resolve the **Pipe and Filter (PaF)** performance antipattern on the **Items Server** microservice by applying the **Move operation** refactoring action
- The **most demanding operation** *findProduct()* of Product Server **is moved to a new microservice** (Items Server 2)
- The new Items Server 2 microservice is **deployed on a new node** (the Items Server 2 Docker container)
- After the refactoring, the response time of the **Web scenario** has been improved by **13.34%**, whereas the response time of the **Warehouse scenario** has been improved by **5.04%**

Recap

- We introduced an approach to support the **identification and solution of performance problems on a running system**
- Monitoring information has been linked to design models by means of the **JTL traceability engine**
- Traceability links have been exploited to **annotate performance indices on design models**
- PADRE has been used to **detect performance antipatterns** and provide **promising refactoring actions**
- The approach has been applied on a case study that was developed and monitored using **industrial standard technologies**

Continue… →

Work in progress area

Runtime information mining (**ongoing**)

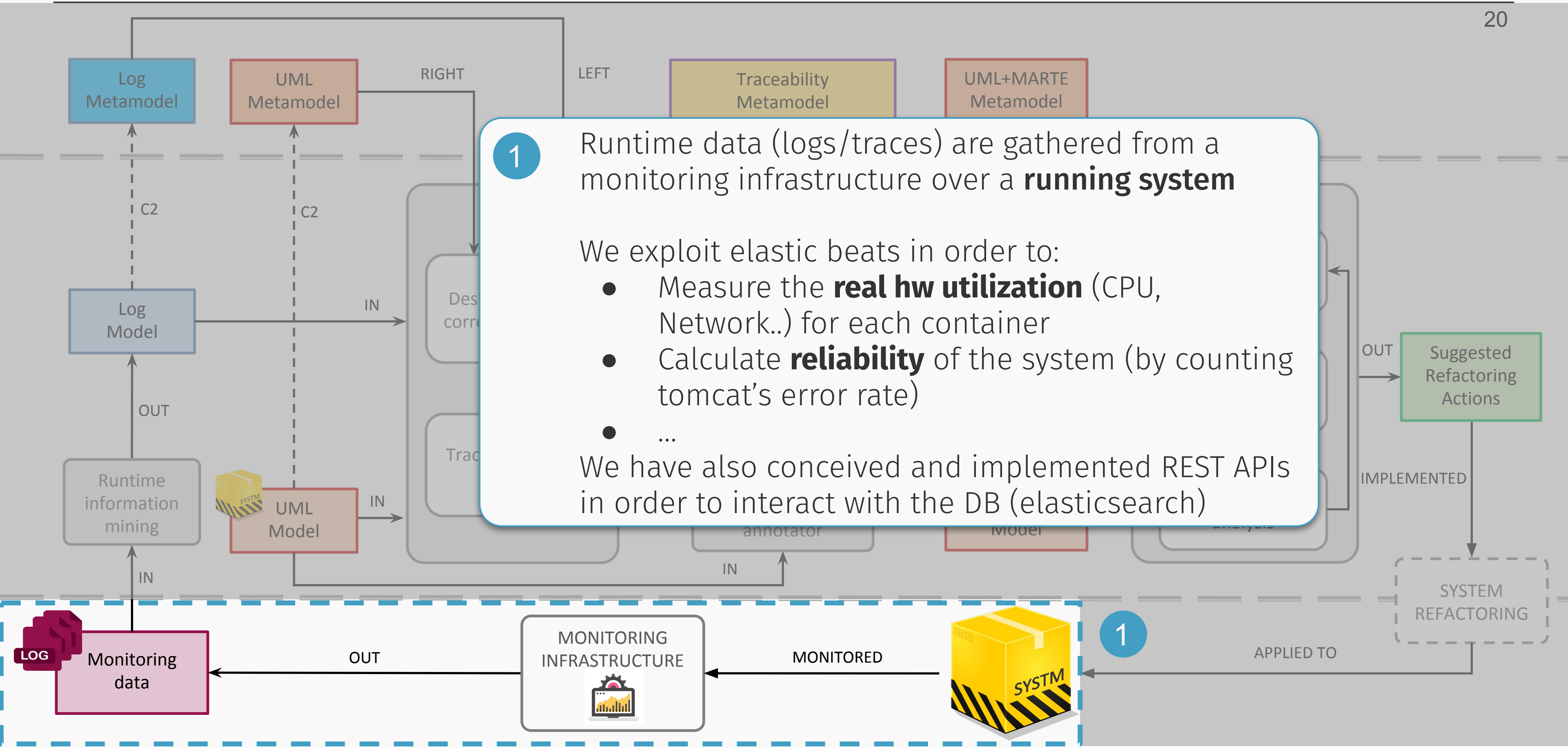
1

Runtime data (logs/traces) are gathered from a monitoring infrastructure over a **running system**

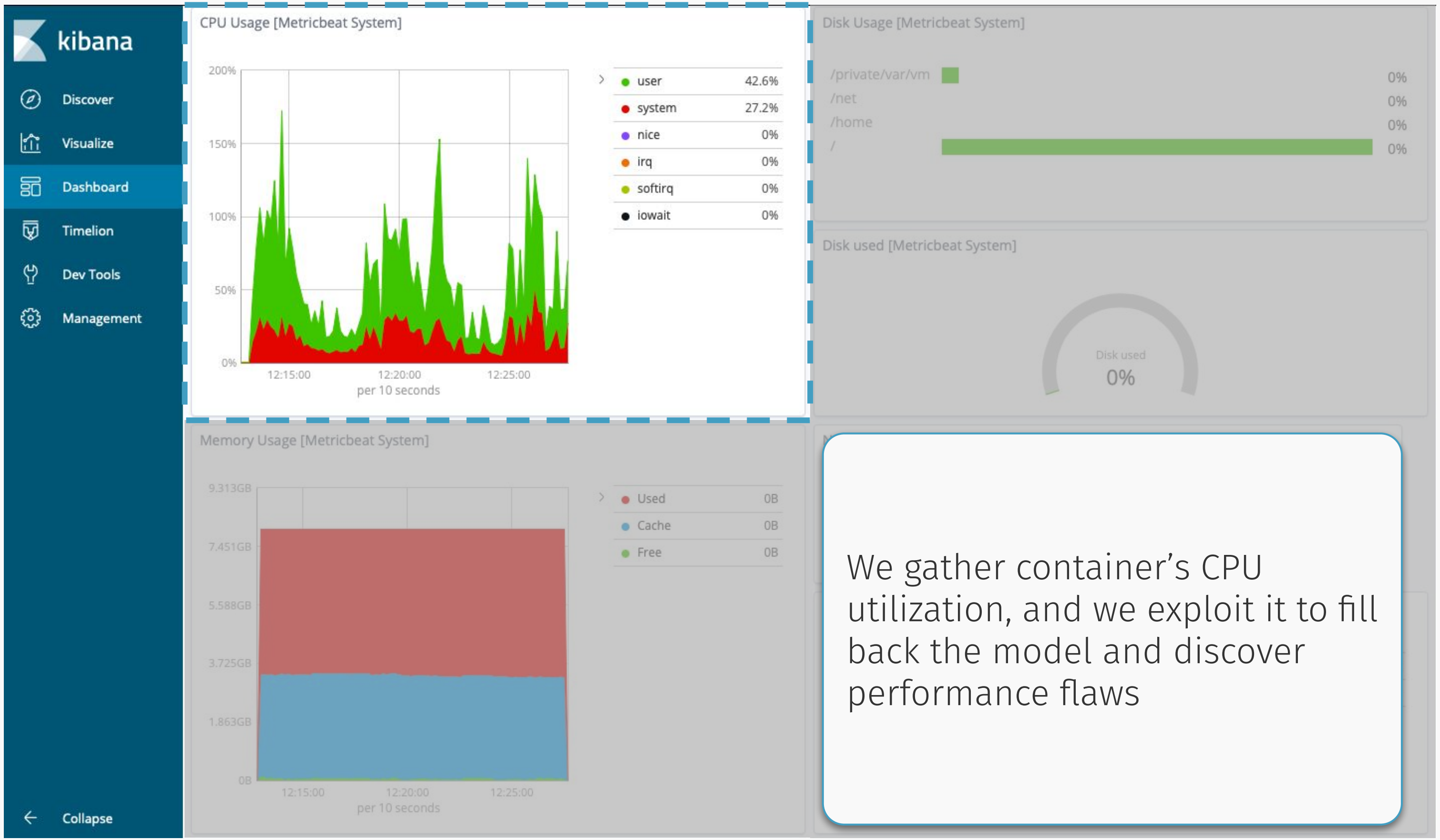
We exploit elastic beats in order to:

- Measure the **real hw utilization** (CPU, Network..) for each container
- Calculate **reliability** of the system (by counting tomcat's error rate)
- ...

We have also conceived and implemented REST APIs in order to interact with the DB (elasticsearch)

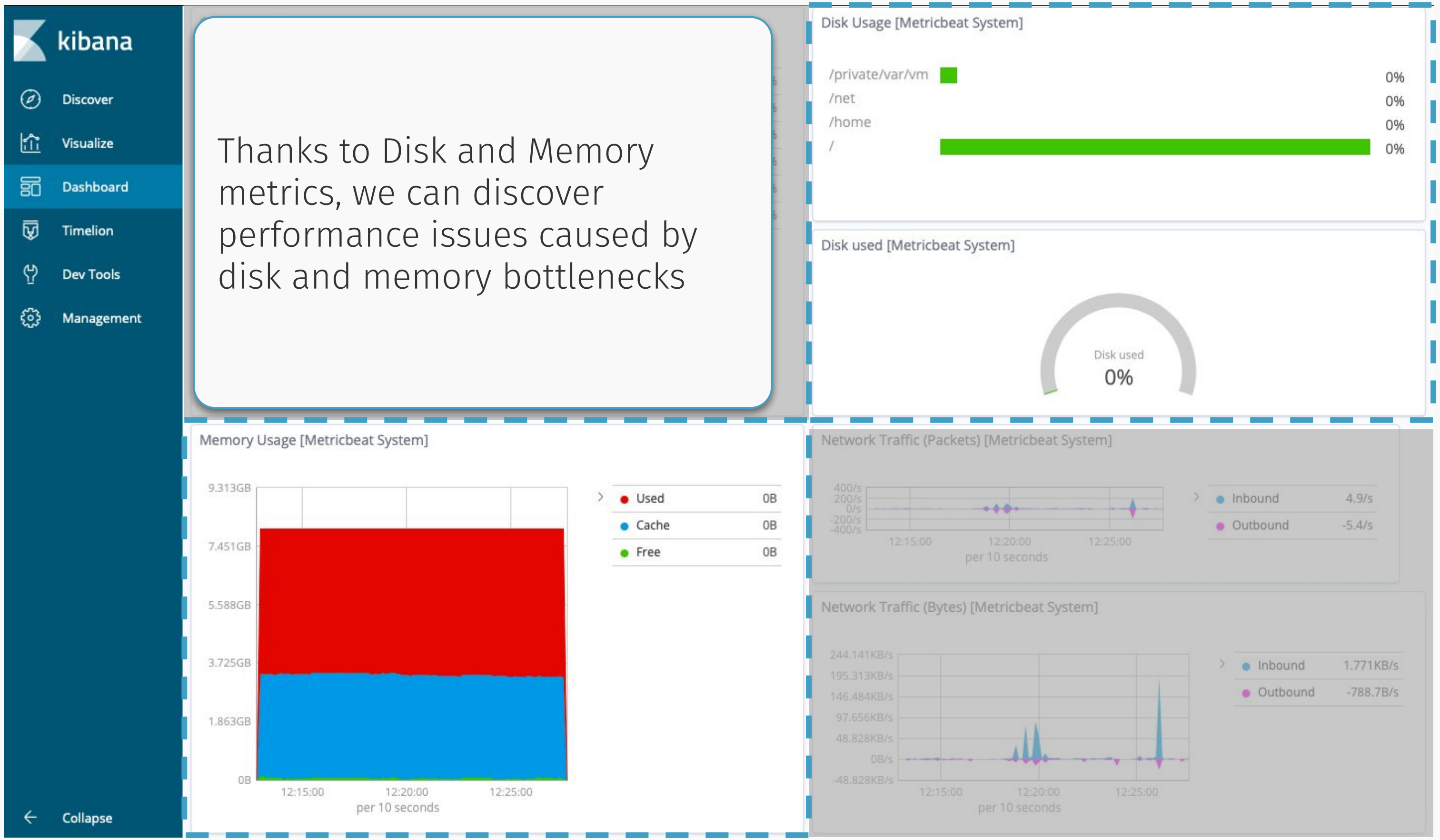


Docker Stats - Metricbeat



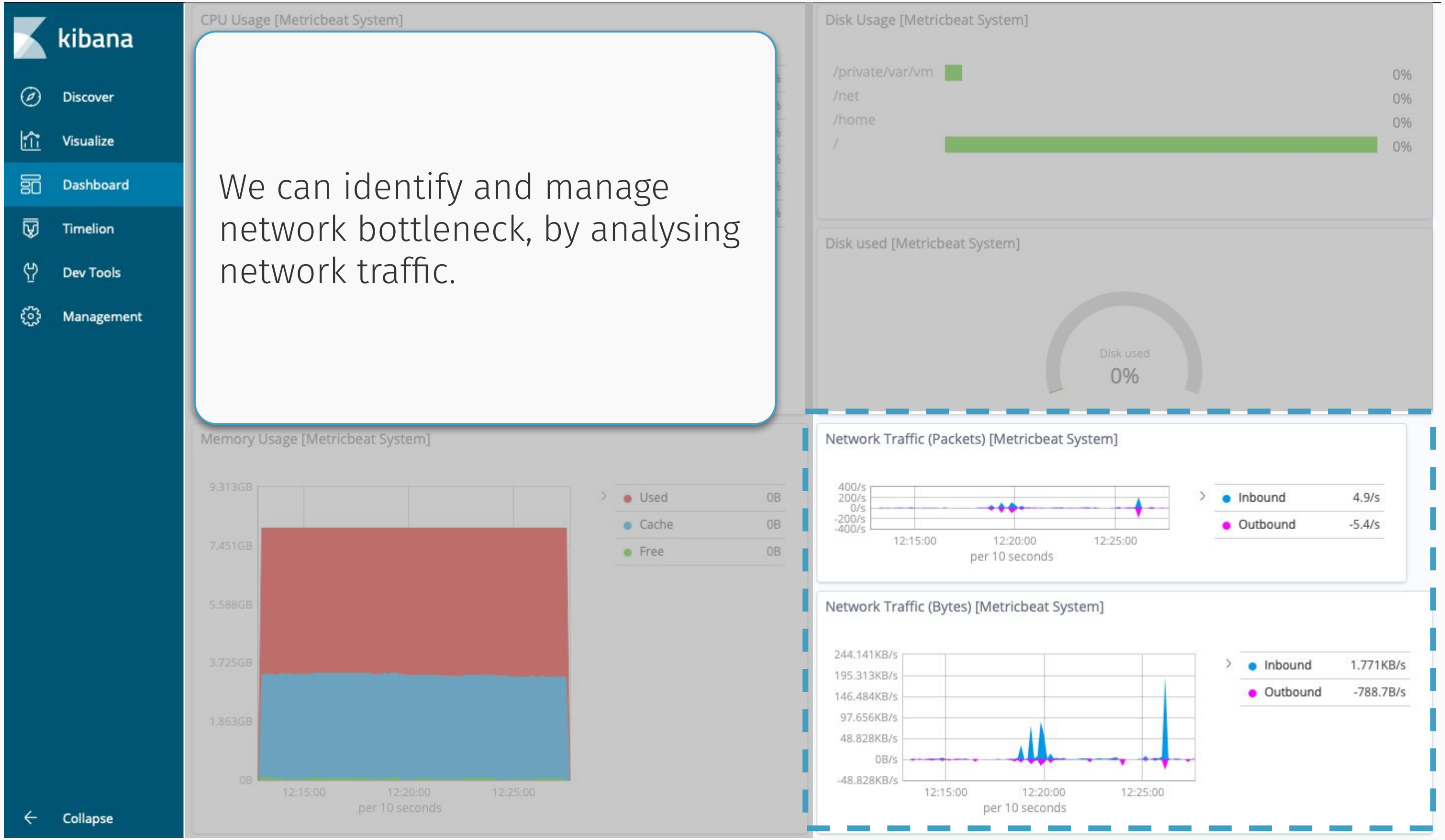
We gather container's CPU utilization, and we exploit it to fill back the model and discover performance flaws

Docker Stats - Metricbeat



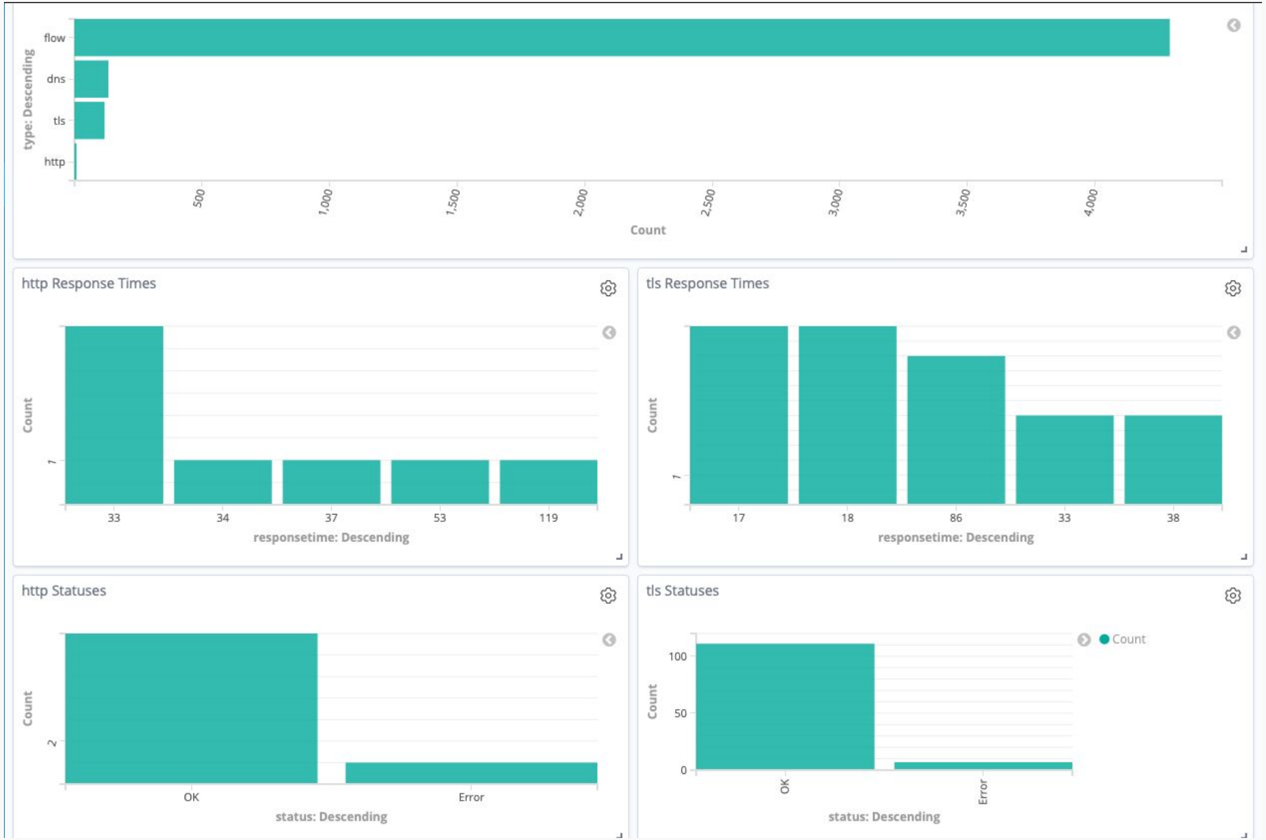
Thanks to Disk and Memory metrics, we can discover performance issues caused by disk and memory bottlenecks

Docker Stats - Metricbeat



We can identify and manage network bottleneck, by analysing network traffic.

Docker Stats - Packetbeat

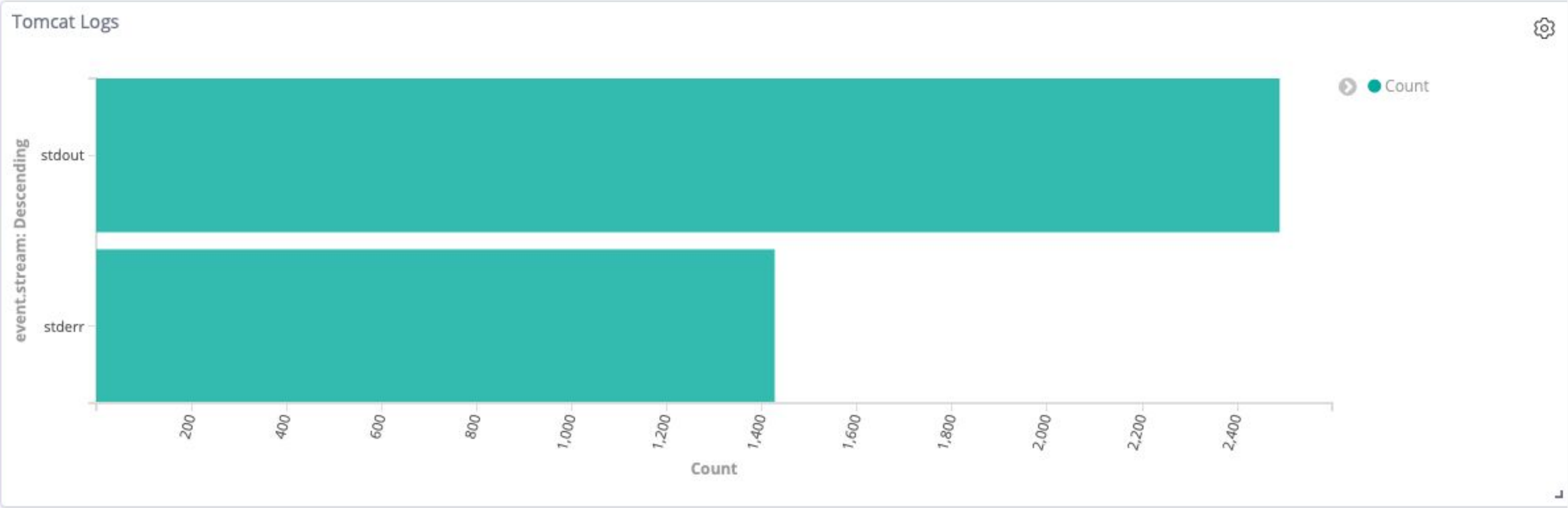


Packet beat plugin helps us to obtain data on exchanged packets

We can measure the average response time for different scenarios and different workloads

We can also measure, for example, the error rate

Docker Stats - Filebeat



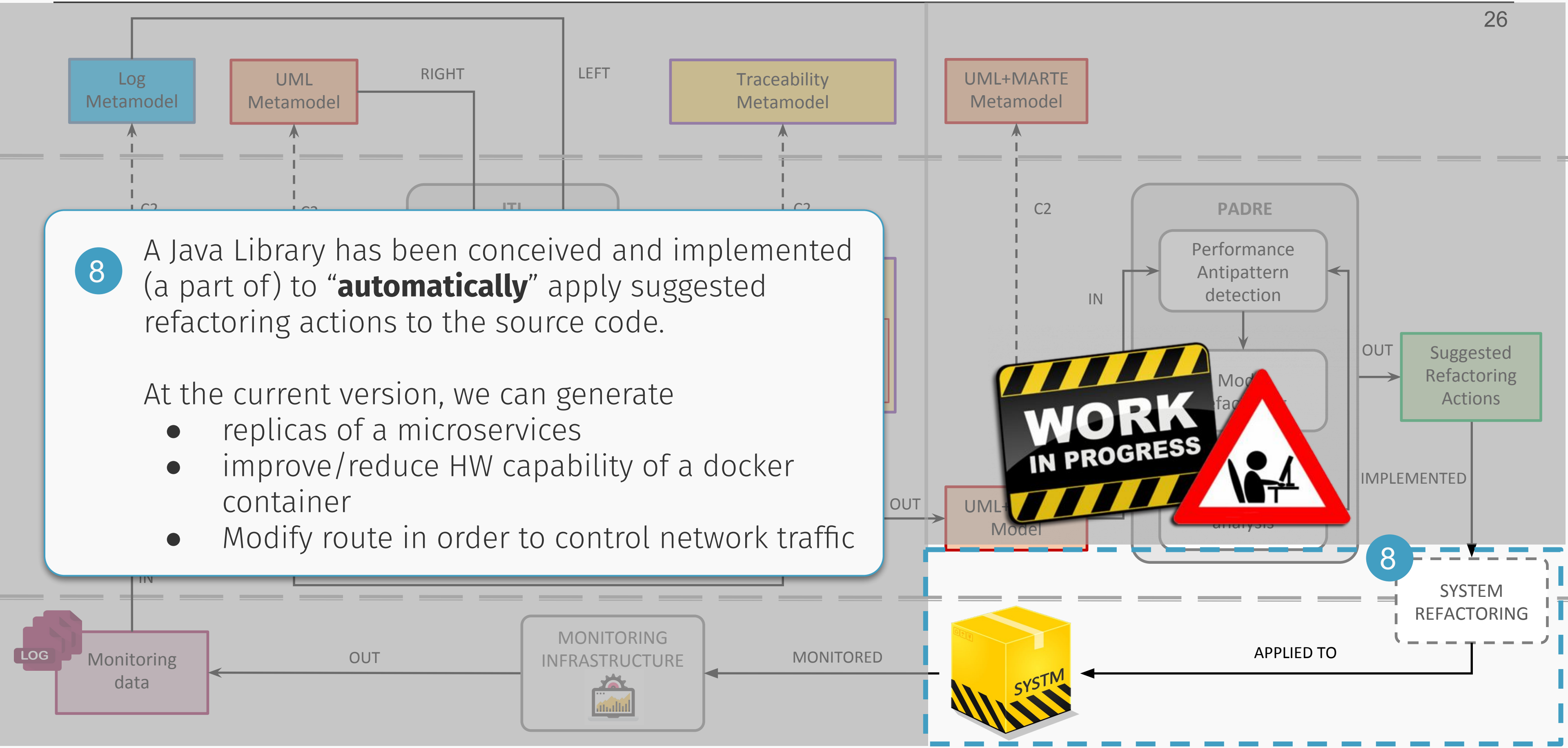
Filebeat plugin helps us to analyse tomcat's log errors, and thus measuring, for example, the reliability/availability of the system

System Refactoring (**ongoing**)

8 A Java Library has been conceived and implemented (a part of) to "**automatically**" apply suggested refactoring actions to the source code.

At the current version, we can generate

- replicas of a microservices
- improve/reduce HW capability of a docker container
- Modify route in order to control network traffic



System Refactoring - Clone, Remove, Update a container

CLONE Container

```
public String cloneContainer(String containerId) {
  try {
    //Lists only running containers
    ContainerInfo containerInfo = docker.inspectContainer(containerId);
    //Get the image of the container to clone
    final ContainerConfig config = ContainerConfig.builder()
      .image(containerInfo.image()).build();
    // Creates the new container
    final String name = "alt_" + containerInfo.name().substring(1);
    final ContainerCreation creation = docker.createContainer(config, name);
    final String newID = creation.id();
    docker.startContainer(newID);
    return newID;
```

REMOVE Container

```
public void removeContainer(String containerId) {
    try {
      System.out.println("List of running containers:");
      List<Container> containers = docker.listContainers();
      docker.stopContainer(containerId, 10);
      docker.removeContainer(containerId);
```

UPDATE Container

```
public void updateContainer(String containerID, long memory, String cpuSetCpus, long cpuShares) {
  final HostConfig newHostConfig = HostConfig.builder()
    .memory(memory).cpusetCpus(cpuSetCpus)
    .cpuShares(cpuShares).build();
```

Department of Information Engineering, Computer Science and Mathematics
University of L'Aquila, Italy

Exploiting Architecture/Runtime Model-driven Traceability for Performance Improvement

SEAQ
QUALITY GROUP

Vittorio Cortellessa, Daniele Di Pompeo, Romina Eramo, Michele Tucci

{name.surname}@univaq.it