

Desenvolvimento de Aplicações com Arquitetura Baseada em Microservices

Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: [@vinicius3w](https://twitter.com/vinicius3w) :: assertlab.com

[IF1007] - Tópicos Avançados em SI 4
<https://github.com/vinicius3w/if1007-Microservices>

Licença do material

Este Trabalho foi licenciado com uma Licença
Creative Commons - Atribuição-NãoComercial-
Compartilhagual 3.0 Não Adaptada

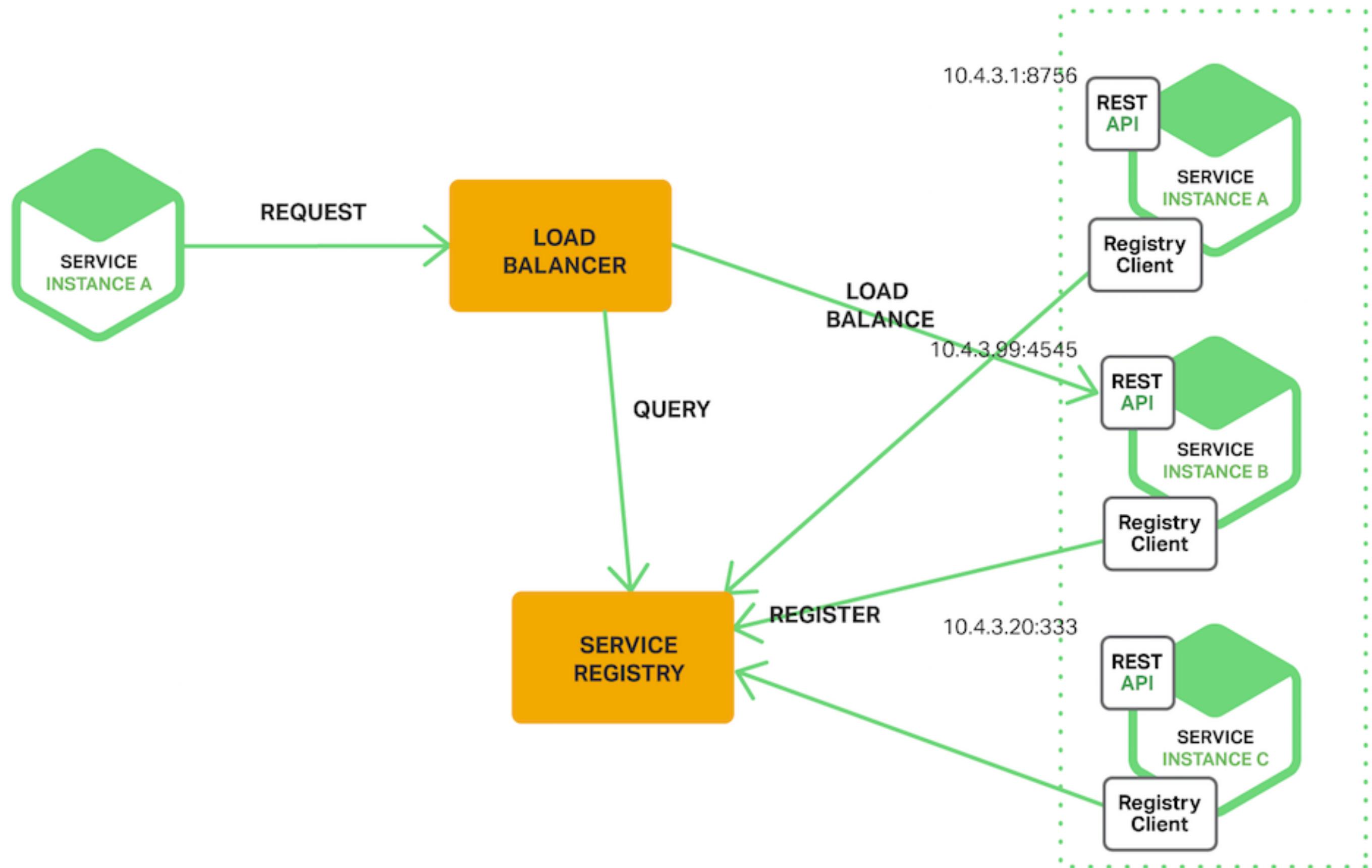


Mais informações visite

[http://creativecommons.org/licenses/by-nc-sa/
3.0/deed.pt](http://creativecommons.org/licenses/by-nc-sa/3.0/deed.pt)

Resources

- There is no textbook required. However, the following are some books that may be recommended:
 - [Building Microservices: Designing Fine-Grained Systems](#)
 - [Spring Microservices](#)
 - [Spring Boot: Acelere o desenvolvimento de microsserviços](#)
 - [Microservices for Java Developers A Hands-on Introduction to Frameworks and Containers](#)
 - [Migrating to Cloud-Native Application Architectures](#)
 - [Continuous Integration](#)
 - [Getting started guides from spring.io](#)

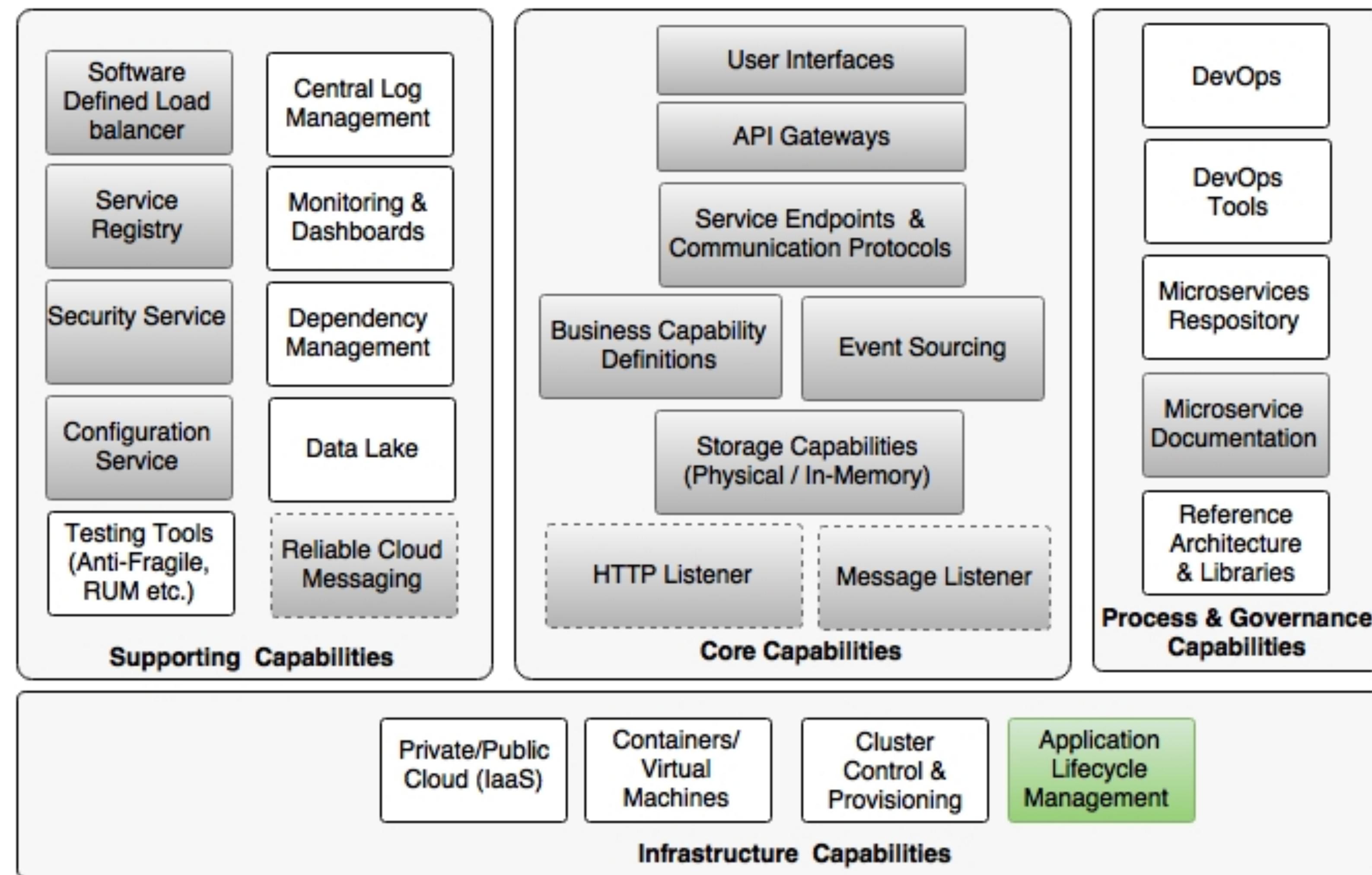


Autoscaling Microservices

Context

- Spring Cloud provides the support essential for the deployment of microservices at scale
- This lecture will detail out how to make microservices elastically grow and shrink by effectively using the actuator data collected from Spring Boot microservices to control the deployment topology by implementing a simple life cycle manager

Reviewing the microservice capability model



Scaling microservices with Spring Cloud

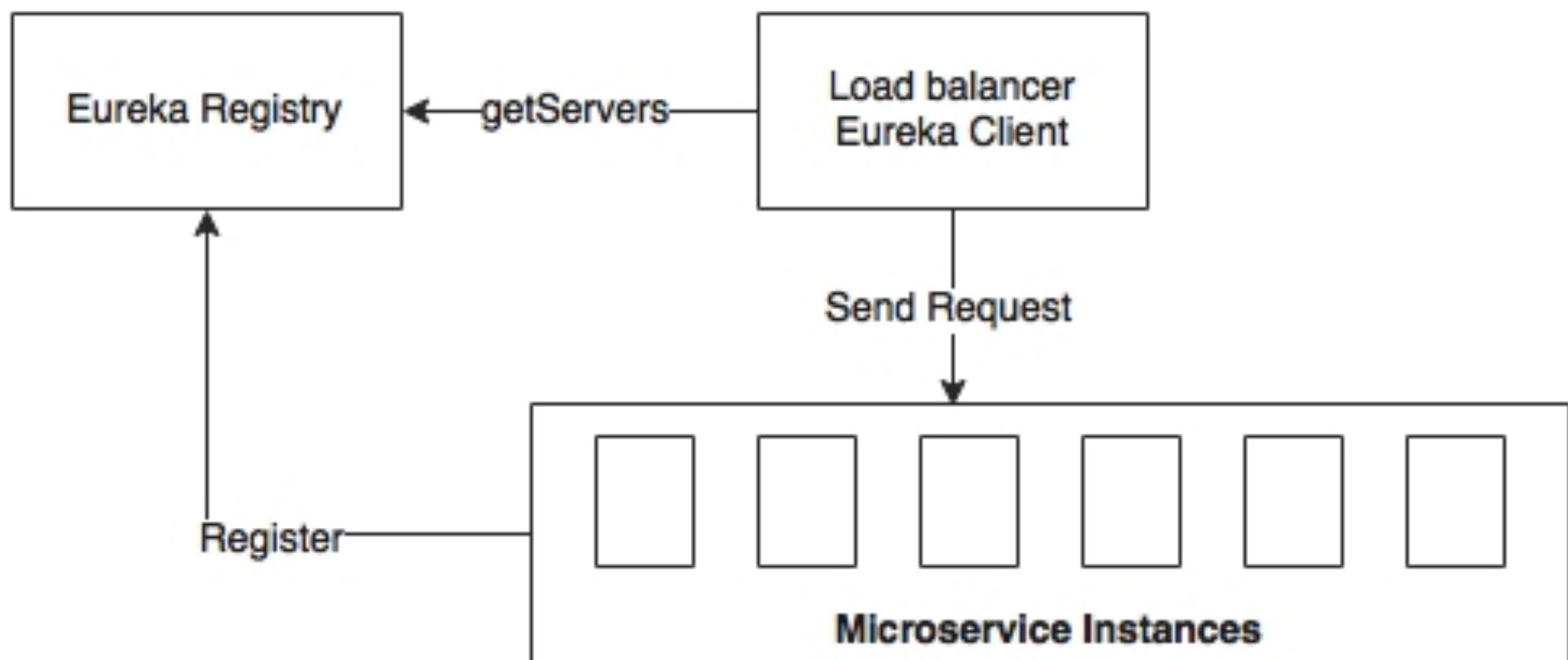
- How to scale Spring Boot microservices using Spring Cloud components
- Two key concepts of Spring Cloud that we implemented are self-registration and self-discovery
~> enable automated microservices deployments
- Self-registration ~> central service registry ~> discovering service instances
- Registry is at the heart of this automation

Scaling with traditional J2EE

- Server instances' IP addresses are more or less statically configured in a load balancer
 - Cluster approach is not the best solution for automatic scaling in Internet-scale deployments
 - They have to have exactly the same version of binaries on all cluster nodes

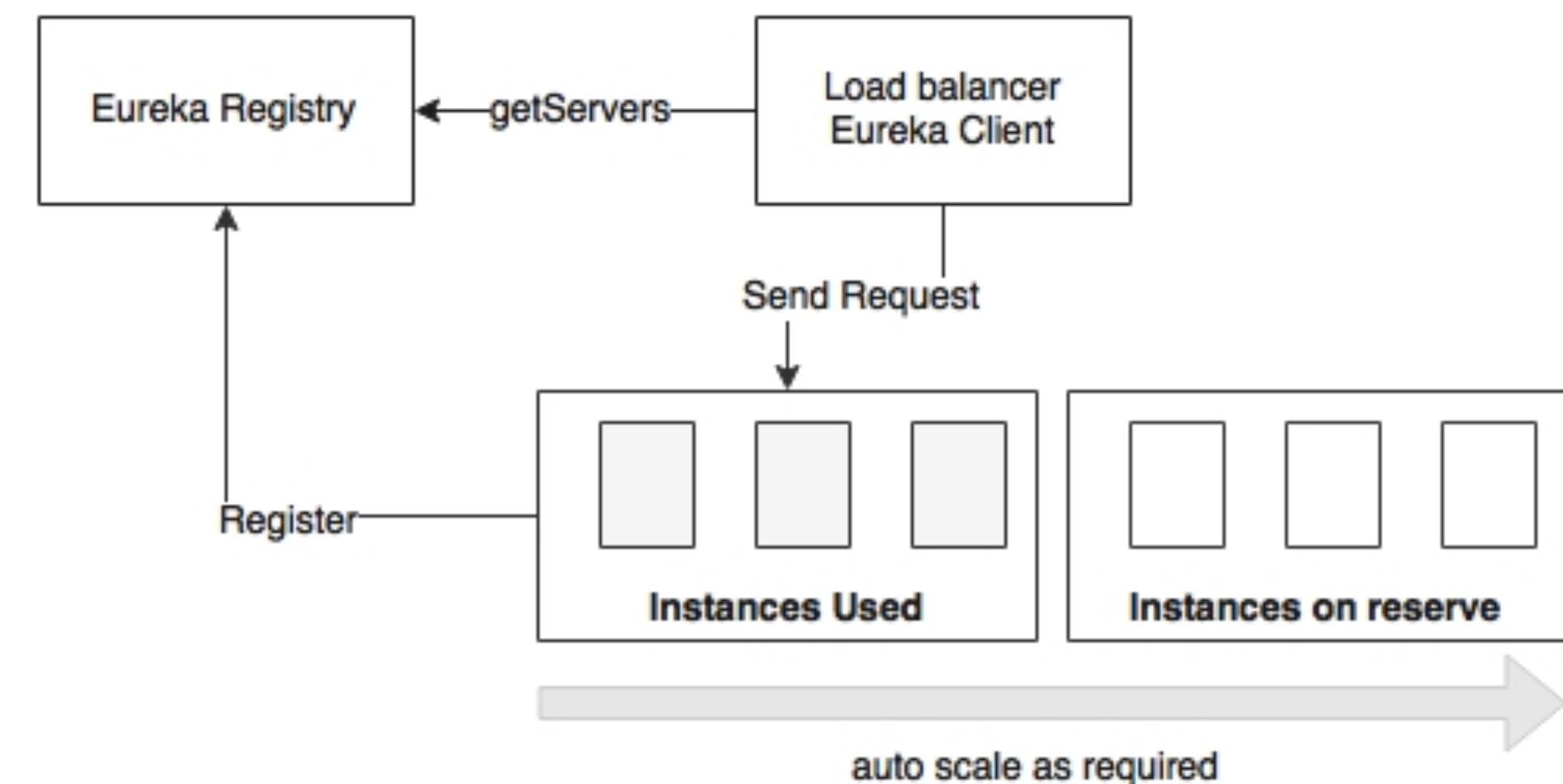
Key components

- Key components in our automated microservices deployment topology
 - Eureka registry
 - Eureka client
 - Microservices instances
- However, there is one gap in this approach
 - When there is need for an additional microservice instance, a manual task is required to kick off a new instance



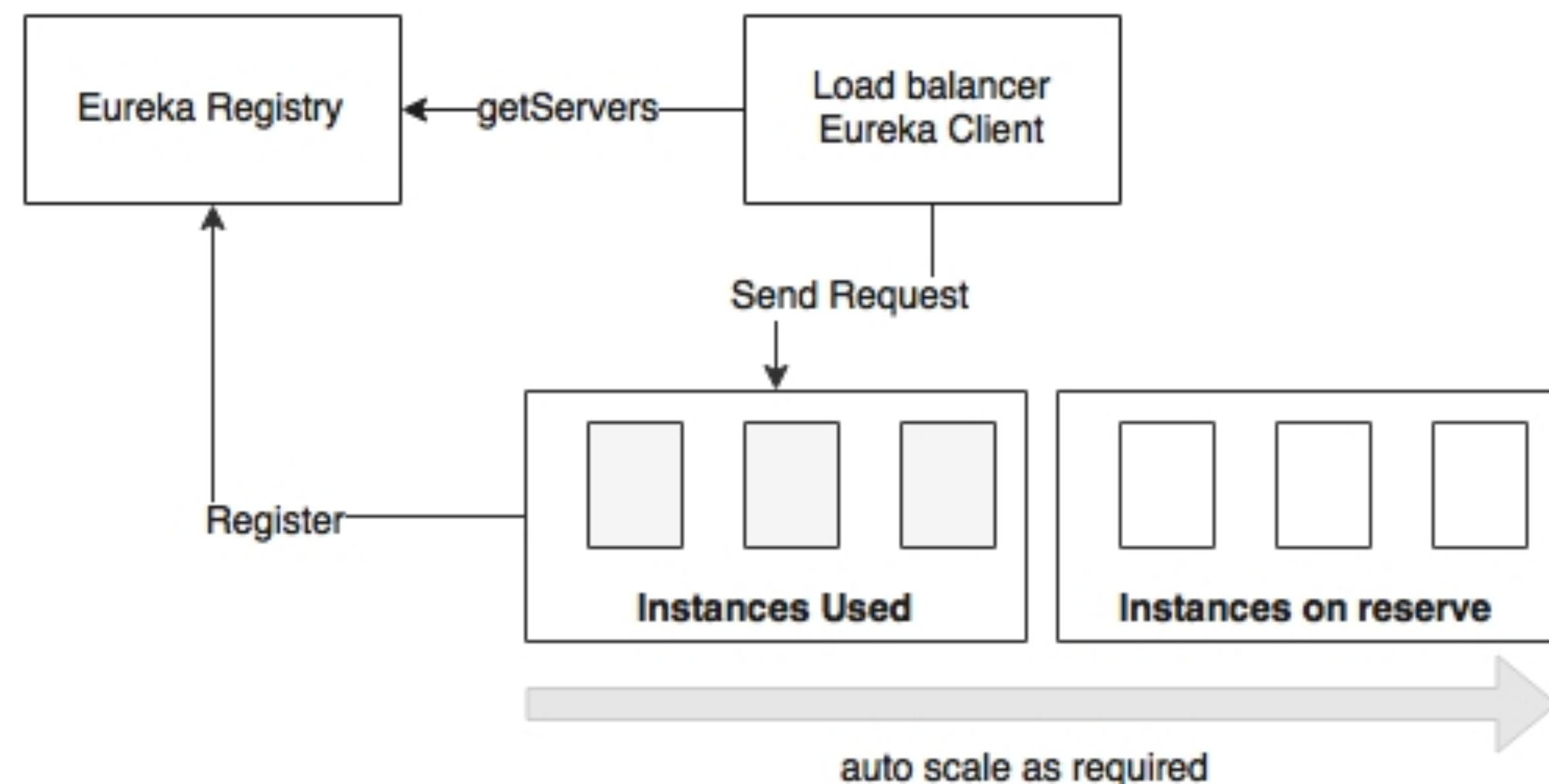
Understanding the concept of autoscaling

- Automatically scaling out instances based on the resource usage to meet the SLAs by replicating the services to be scaled
- Autoscaling is done, generally, using a set of reserve machines
- This approach is often called elasticity or dynamic resource provisioning and deprovisioning



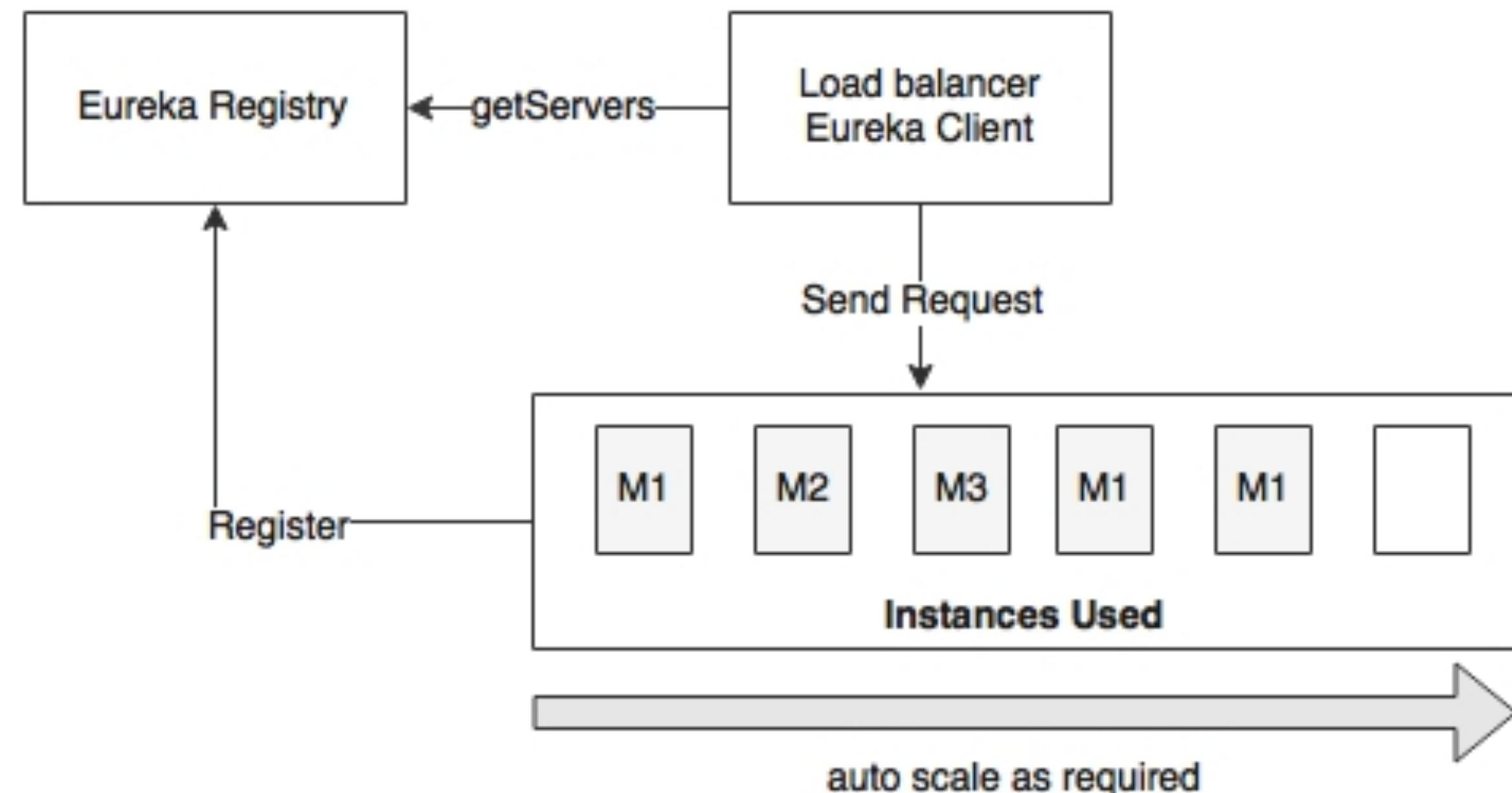
Resource pool of instances

- Based on the demand, instances will be moved from the resource pool to the active state to meet the surplus demand
- In advanced deployments, the Spring Boot binaries are downloaded on demand from an artifact repository such as [Nexus](#) or [JFrog](#)



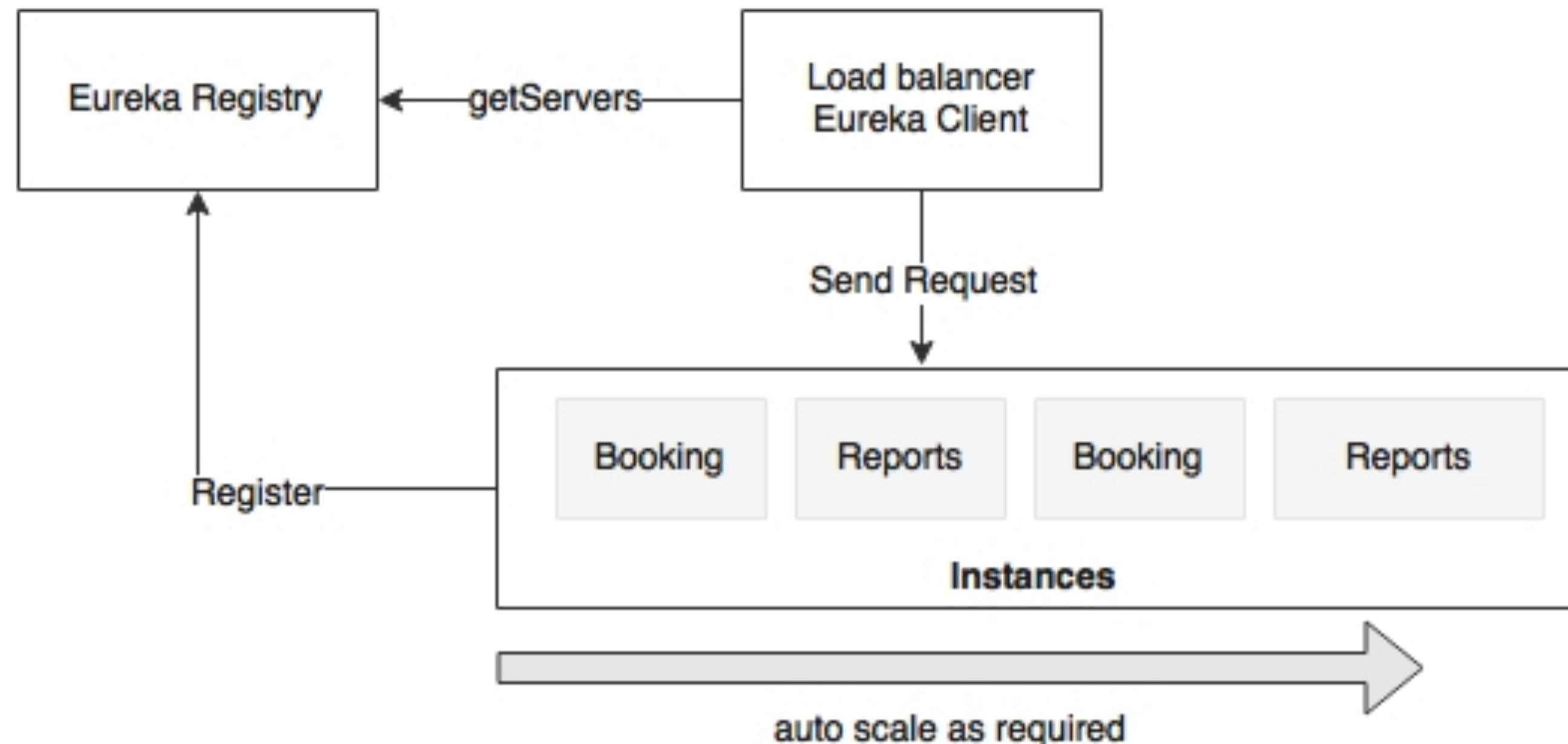
The benefits of autoscaling

- In traditional deployments, administrators reserve a set of servers against each application
- With autoscaling, this preallocation is no longer required



The benefits of autoscaling

- It has high availability and is fault tolerant
- It increases [horizontal] scalability
- It has optimal usage and is cost saving
- It gives priority to certain services or group of services

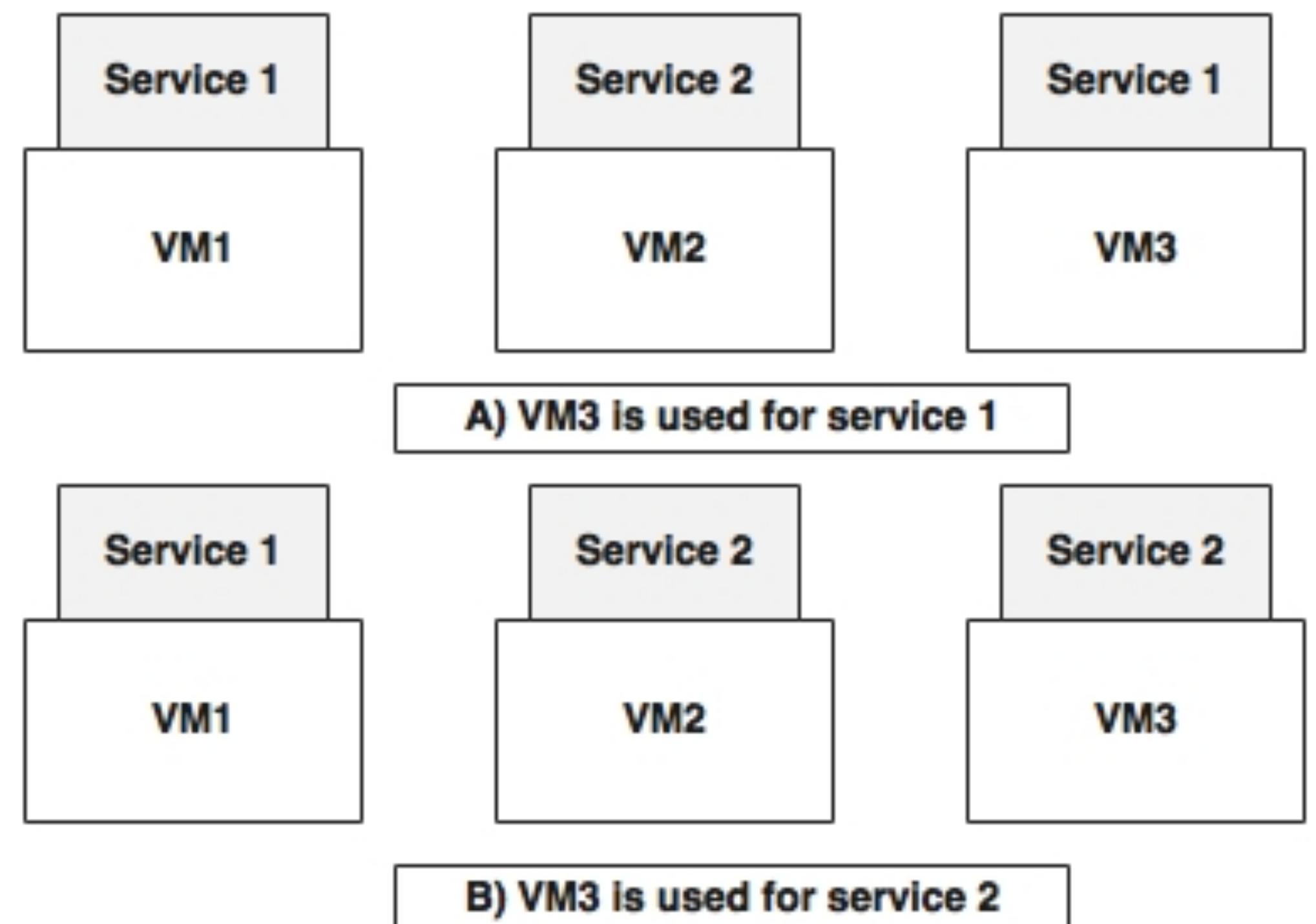


Different autoscaling models

- Autoscaling can be applied at the application level or at the infrastructure level
- In a nutshell, application scaling is scaling by replicating application binaries only, whereas infrastructure scaling is replicating the entire virtual machine, including application binaries

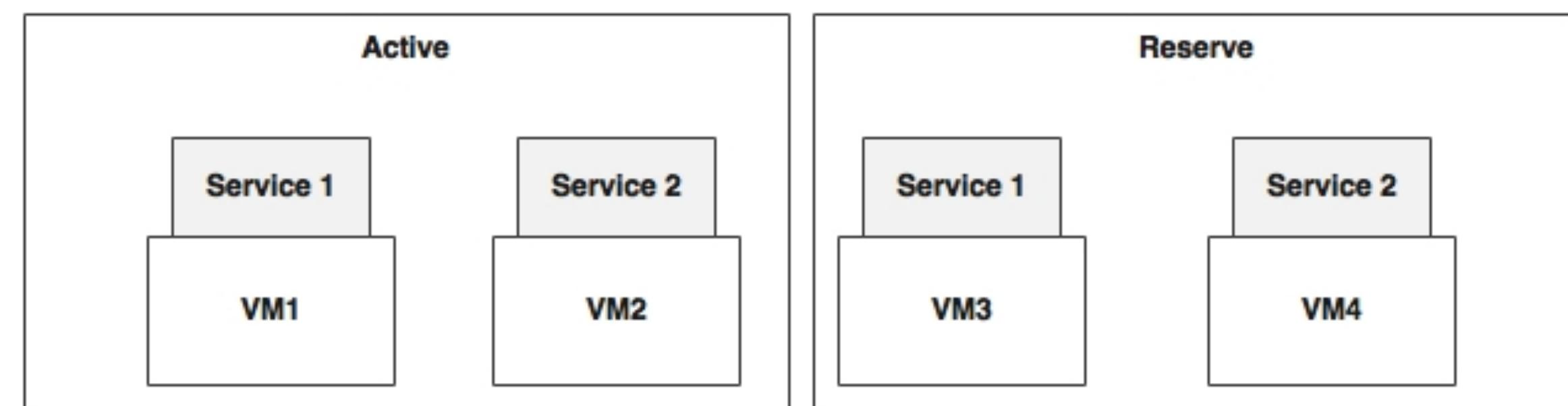
Autoscaling an application

- Scaling is done by replicating the microservices, not the underlying infrastructure
- Flexibility in reusing the same virtual or physical machines for different services
- Is there a downside in this approach?



Autoscaling the infrastructure

- The infrastructure is also provisioned automatically
- In most cases, this will create a new VM on the fly or destroy the VMs based on the demand
- This approach is efficient if the applications depend upon the parameters and libraries at the infrastructure level, such as the OS
 - Also, this approach is better for polyglot microservices



Autoscaling in the cloud

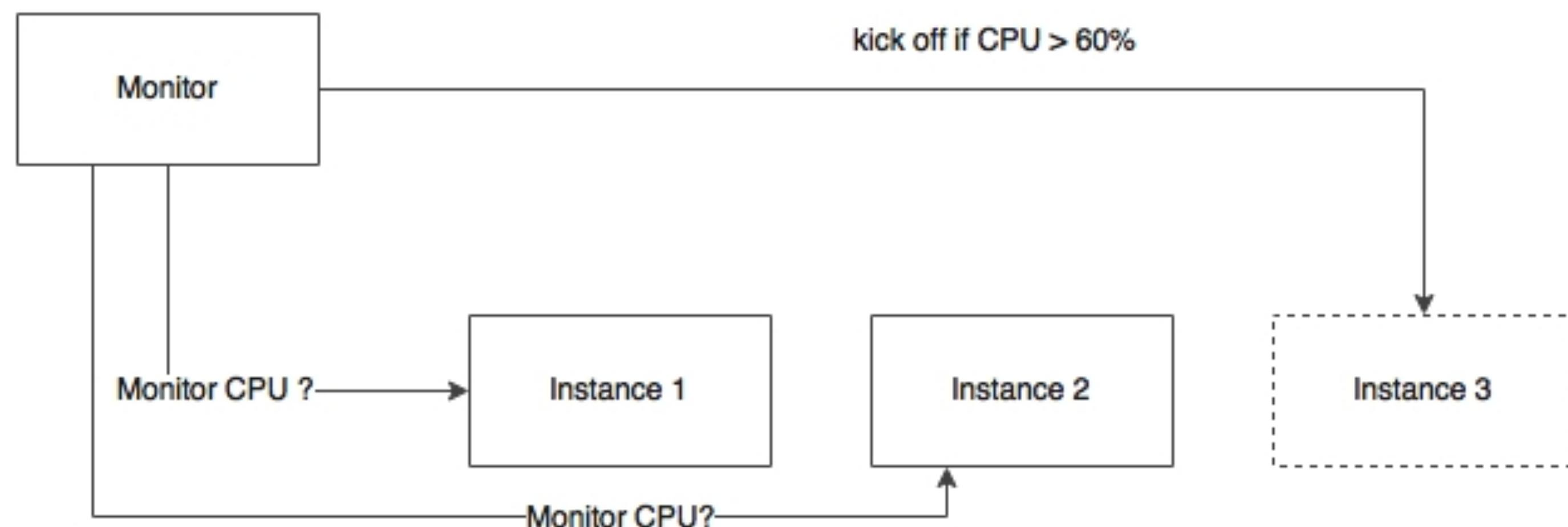
- Elasticity or autoscaling is one of the fundamental features of most cloud providers
- Cloud providers use infrastructure scaling patterns, as discussed in the previous section
- These are typically based on a set of pooled machines

Autoscaling approaches

- Autoscaling is handled by considering different parameters and thresholds
- What are the different approaches and policies that are typically applied to take decisions on when to scale up or down?

Scaling with resource constraints

- Based on real-time service metrics collected through monitoring mechanisms
- Generally, the resource-scaling approach takes decisions based on the CPU, memory, or the disk of machines
- This can also be done by looking at the statistics collected on the service instances themselves, such as heap memory usage



Sliding window or a waiting period

- An example of a response sliding window is if 60% of the response time of a particular transaction is consistently more than the set threshold value in a 60-second sampling window, increase service instances
- In a CPU sliding window, if the CPU utilization is consistently beyond 70% in a 5 minutes sliding window, then a new instance is created
- An example of the exception sliding window is if 80% of the transactions in a sliding window of 60 seconds or 10 consecutive executions result in a particular system exception, such as a connection timeout due to exhausting the thread pool, then a new service instance is created

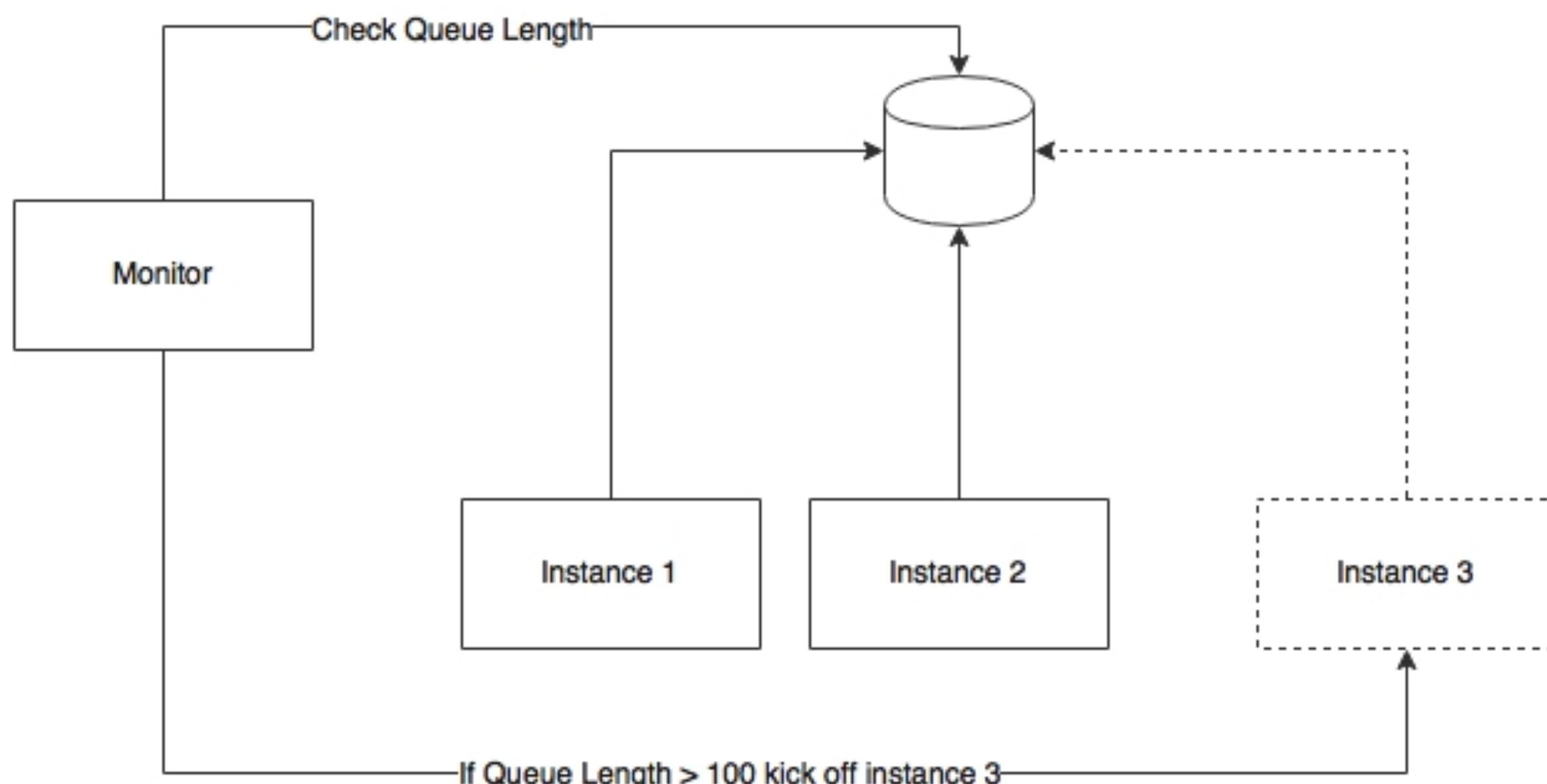
Scaling during specific time periods

- Time-based scaling is an approach to scaling services based on certain periods of the day, month, or year to handle seasonal or business peaks
- For example, some services may experience a higher number of transactions during office hours and a considerably low number of transactions outside office hours



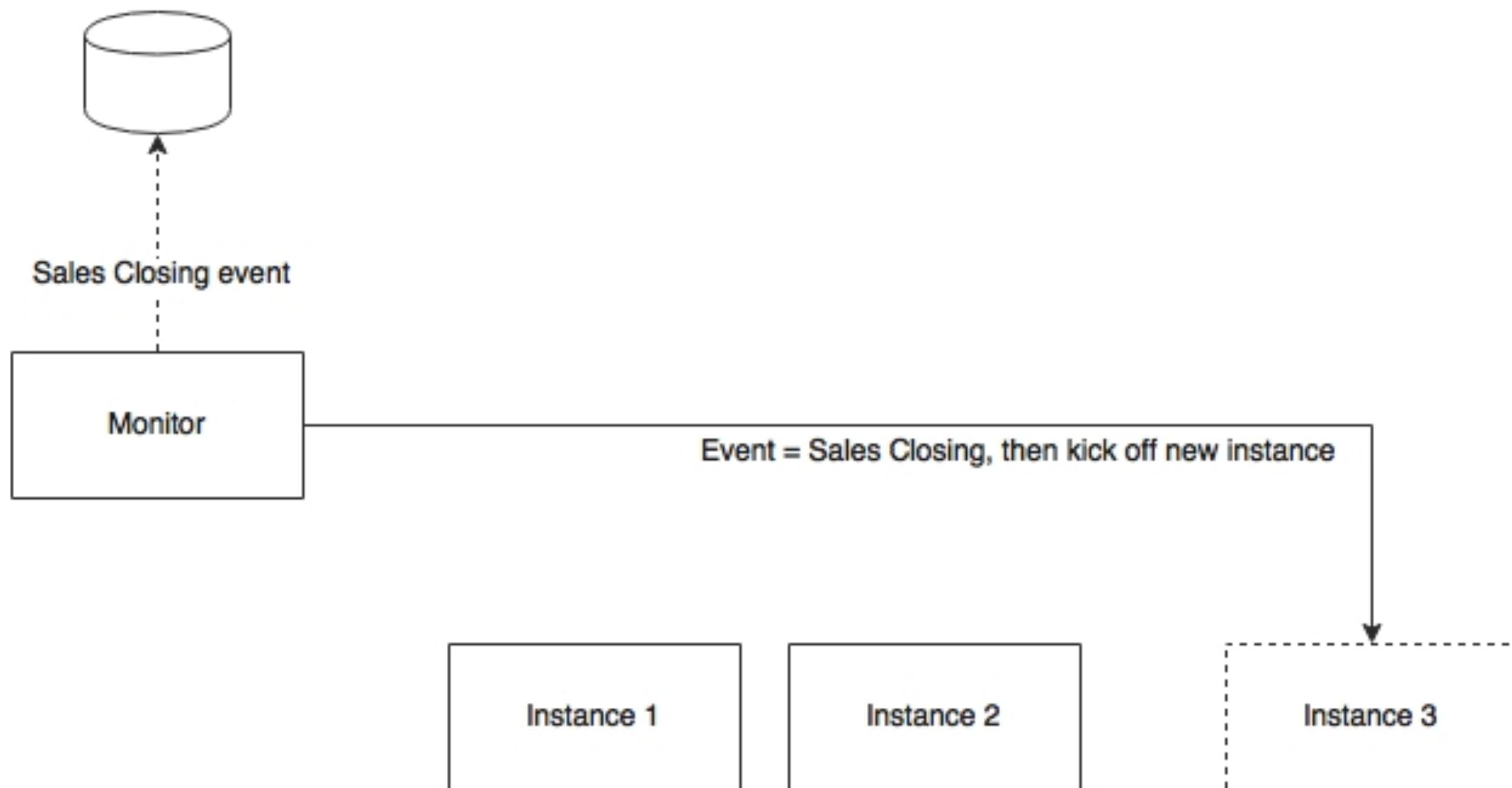
Scaling based on the message queue length

- Particularly useful when the microservices are based on asynchronous messaging
- New consumers are automatically added when the messages in the queue go beyond certain limits
- This approach is based on the competing consumer pattern



Scaling based on business parameters

- Adding instances is based on certain business parameters—for example, spinning up a new instance just before handling **sales closing** transactions
- As soon as the monitoring service receives a preconfigured business event (such as **sales closing minus 1 hour**), a new instance will be brought up in anticipation of large volumes of transactions
- This will provide fine-grained control on scaling based on business rules



Predictive autoscaling

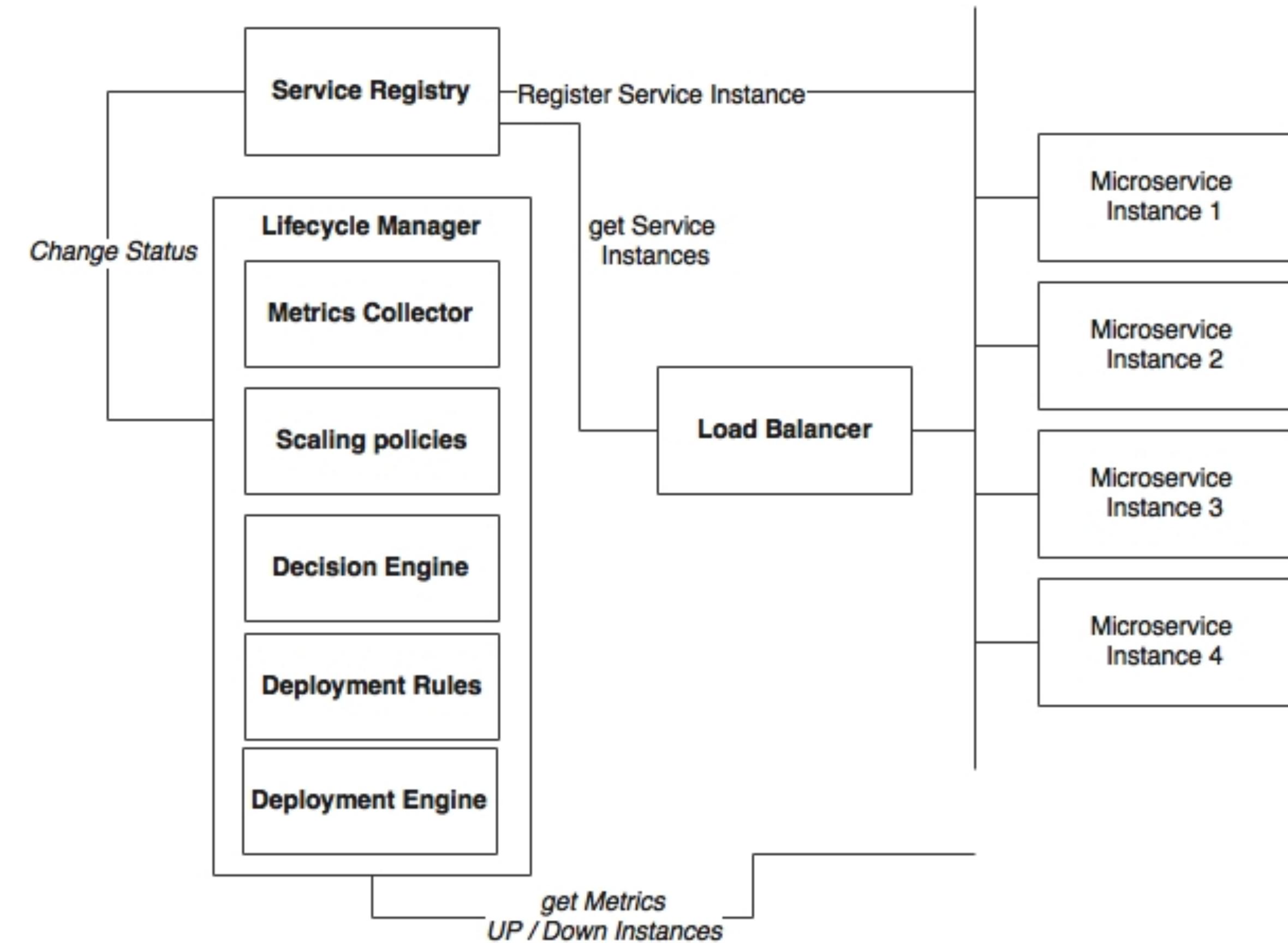
- Is a new paradigm of autoscaling that is different from the traditional real-time metrics-based autoscaling
- A prediction engine will take multiple inputs, such as historical information, current trends, and so on, to predict possible traffic patterns
- [Netflix Scryer](#) is an example of such a system that can predict resource requirements in advance

Autoscaling BrownField PSS microservices

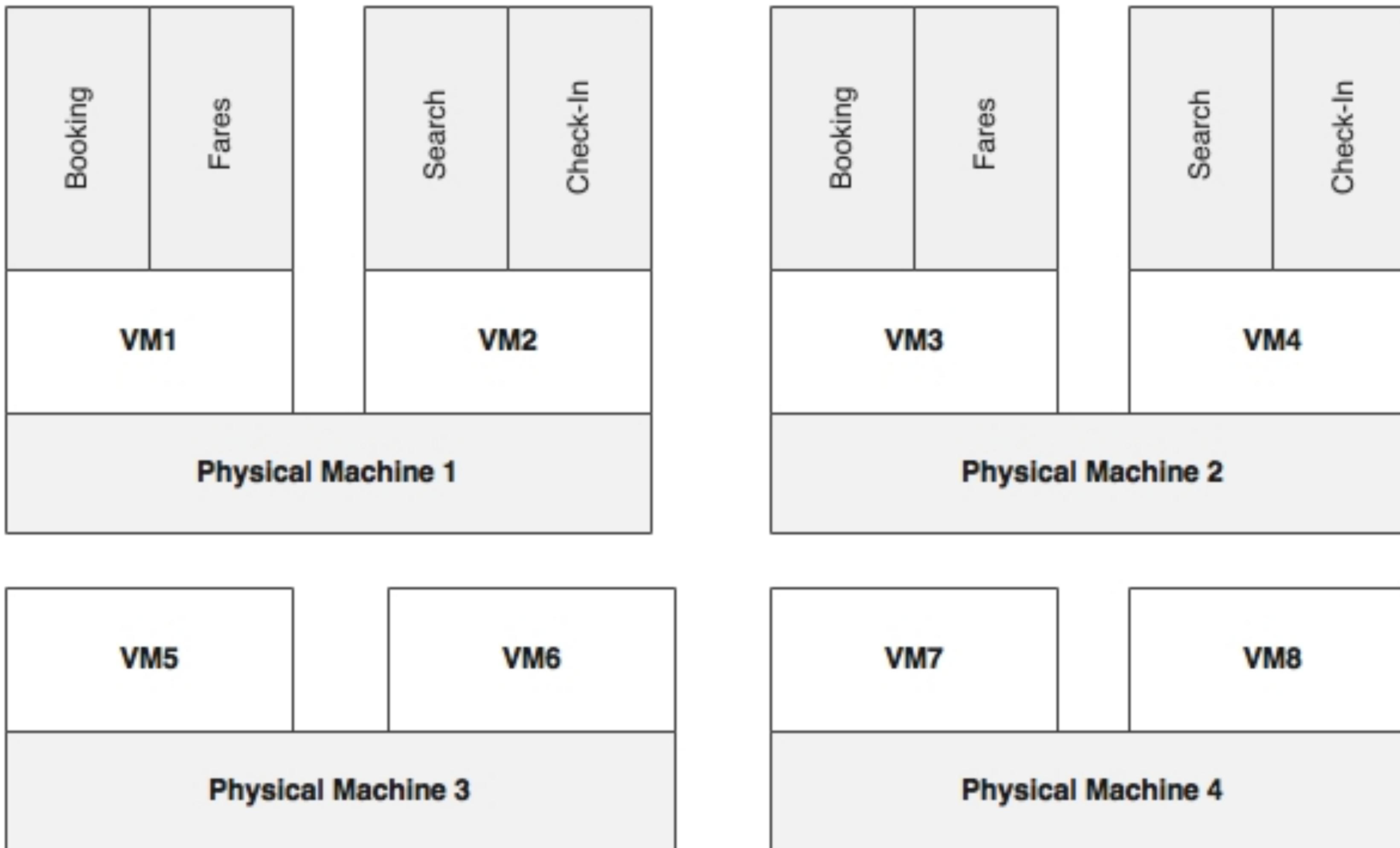
- How to enhance microservices developed before?
- We need a component to monitor certain performance metrics and trigger autoscaling
- We will call this component the **life cycle manager**

The capabilities required for an autoscaling system

- Metrics Collector: responsible for collecting metrics from all service instances
- Scaling policies: sets of rules indicating when to scale up and scale down
- Decision Engine: responsible for making decisions based on the aggregated metrics and scaling policies
- Deployment Rules: DpE decide which parameters to consider when deploying services
- Deployment Engine: can start or stop MS instances or update the registry by altering the health states of services

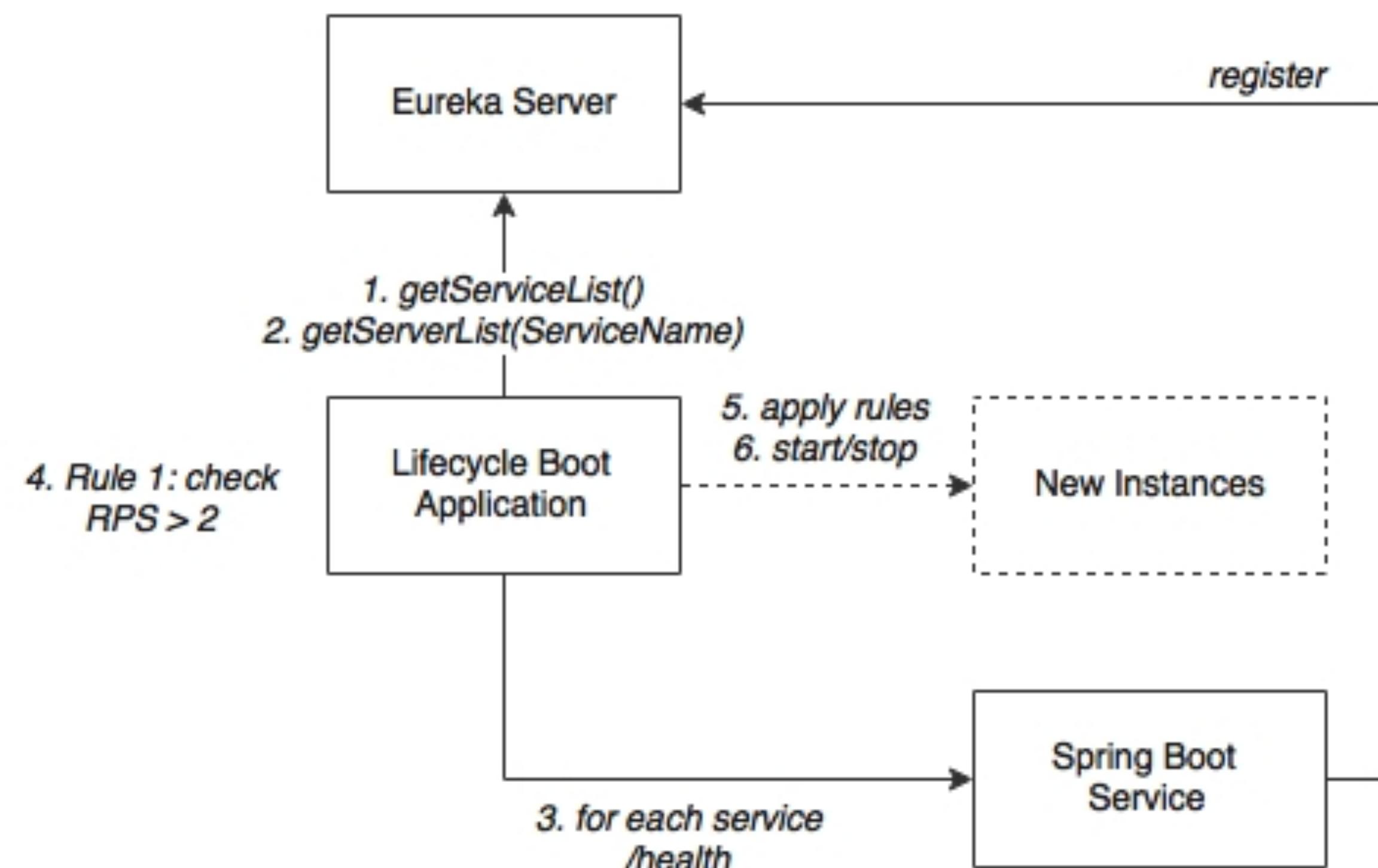


Understanding the deployment topology



Understanding the execution flow

- The Spring Boot service represents microservices such as Search, Book, Fares, and Check-in, at startup automatically register endpoint details to the Eureka registry
- With the collected metrics information, the life cycle manager executes a list of policies and derives decisions on whether to scale up or scale down instances
- In the case of shutdown, it connects to the server using an actuator endpoint and calls the shutdown service to gracefully shut down an instance
- In the case of starting a new instance, the deployment engine of the life cycle manager uses the scaling rules and decides where to start the new instance and what parameters are to be used when starting the instance



Understanding the execution flow

- In this example, we will use TPM (Transactions Per Minute) or RPM (Requests Per Minute) as sampler metrics for decision making
- If the Search service has more than 10 TPM, then it will spin up a new Search service instance
- Similarly, if the TPM is below 2, one of the instances will be shut down and released back to the pool

When starting a new instance, the following policies will be applied

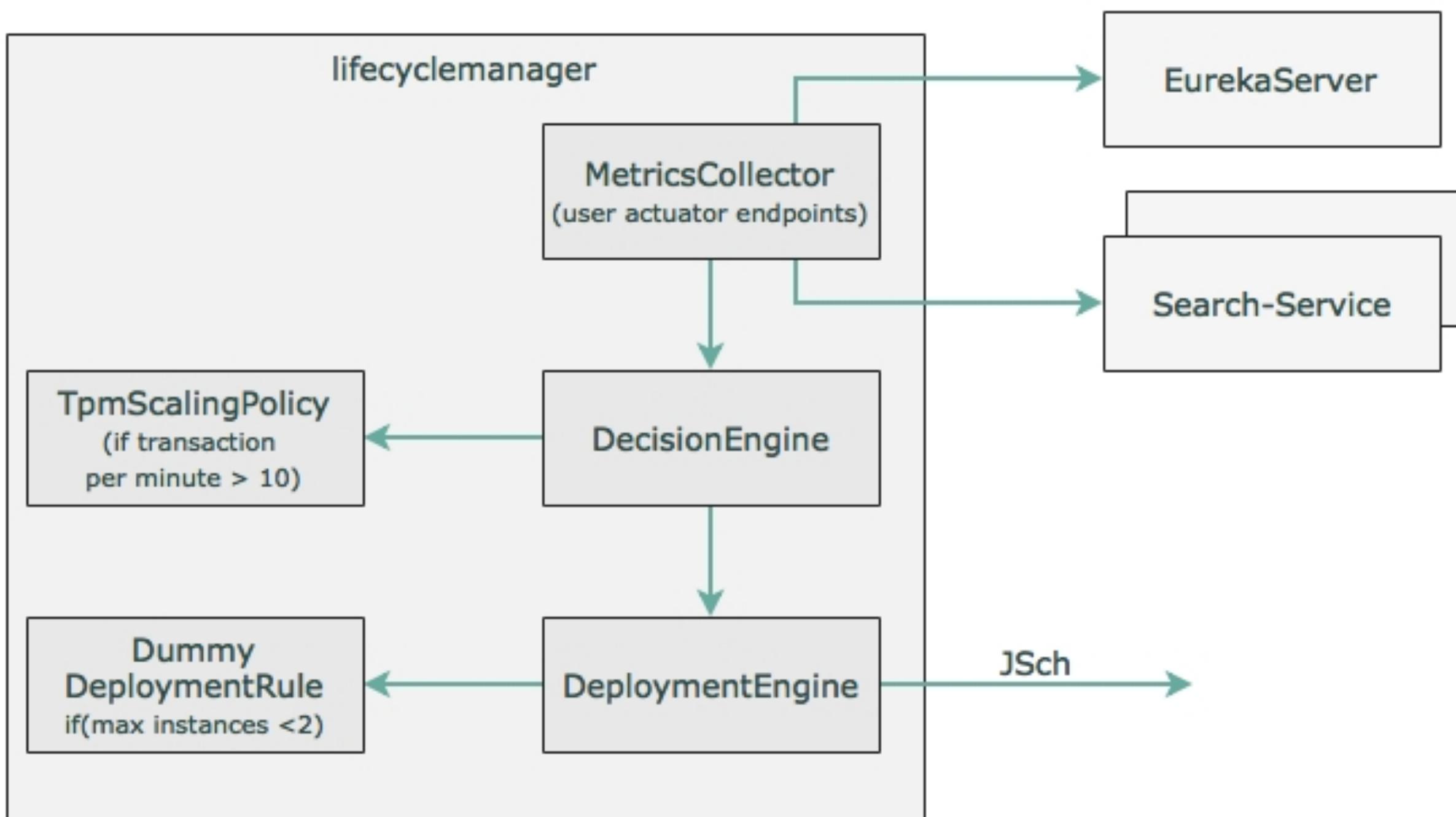
- The number of service instances at any point should be a minimum of 1 and a maximum of 4
 - This also means that at least one service instance will always be up and running
- A scaling group is defined in such a way that a new instance is created on a VM that is on a different physical machine
 - This will ensure that the services run across different physical machines

Homework 7

- How a simple life cycle manager is implemented?

A walkthrough of the life cycle manager code

- Create a new Spring Boot application and name it **lecture7.lifecyclemanager**
- The flowchart for this example is as shown in the following diagram



The screenshot shows the file structure of the **lecture7.lifecyclemanager** project in IntelliJ IDEA:

- src/main/java**: Contains packages for **com.brownfied.pss.lcm**, **com.brownfied.pss.lcm.decisionengine**, **com.brownfied.pss.lcm.deploymentengine**, **com.brownfied.pss.lcm.deploymentrules**, **com.brownfied.pss.lcm.metricscollector**, and **com.brownfied.pss.lcm.scalingpolicy**. Each package contains Java source files for Application.java, DecisionEngine.java, DeploymentEngine.java, DeploymentRule.java, DeploymentRules.java, DummyDeploymentRule.java, MetricsCollector.java, ScalingPolicies.java, ScalingPolicy.java, and TpmScalingPolicy.java.
- src/main/resources**: Contains bootstrap.properties.
- src/test/java**: Contains test Java source files.
- JRE System Library [JavaSE-1.8]**: Maven dependency for Java SE 1.8.
- Maven Dependencies**: Maven dependency reports.
- bin**: Maven generated binary files.
- src**: Maven generated source files.
- target**: Maven generated target files.
- mvnw**: Maven wrapper script.
- mvnw.cmd**: Maven wrapper script for Windows.
- pom.xml**: Maven project configuration file.

A walkthrough of the life cycle manager code

- Create a **MetricsCollector** class with the following method
- At the startup of the Spring Boot application, this method will be invoked using **CommandLineRunner**

```
public void start(){
    while(true){
        eurekaClient.getServices().forEach(service -> {
            System.out.println("printing service " + service);
            Map metrics = restTemplate.getForObject("http://"+service+"/metrics",Map.class);
            decisionEngine.execute(service, metrics);
        });
    }
}
```

- The preceding method looks for the services registered in the Eureka server and gets all the instances
 - In the real world, rather than polling, the instances should publish metrics to a common place, where metrics aggregation will happen

A walkthrough of the life cycle manager code

- The following **DecisionEngine** code accepts the metric and applies certain scaling policies to determine whether the service requires scaling up or not

```
public boolean execute(String serviceId, Map metrics){  
    if(scalingPolicies.getPolicy(serviceId).execute(serviceId, metrics)){  
        return deploymentEngine.scaleUp(deploymentRules.getDeploymentRules(serviceId), serviceId);  
    }  
    return false;  
}
```

A walkthrough of the life cycle manager code

- Based on the service ID, the policies that are related to the services will be picked up and applied. In this case, a minimal **Transactions Per Minute** scaling policy is implemented in **TpmScalingPolicy**

```
public class TpmScalingPolicy implements ScalingPolicy {  
    public boolean execute(String serviceId, Map metrics){  
        if(metrics.containsKey("gauge.servo.tpm")){  
            Double tpm = (Double) metrics.get("gauge.servo.tpm");  
            System.out.println("gauge.servo.tpm " + tpm);  
            return (tpm > 10);  
        }  
        return false;  
    }  
}
```

A walkthrough of the life cycle manager code

- If the policy returns true, **DecisionEngine** then invokes **DeploymentEngine** to spin up another instance
- **DeploymentEngine** makes use of **DeploymentRules** to decide how to execute scaling
- The rules can enforce the number of min and max instances, in which region or machine the new instance has to be started, the resources required for the new instance, and so on
- **DummyDeploymentRule** simply makes sure the max instance is not more than 2

A walkthrough of the life cycle manager code

- **DeploymentEngine**, in this case, uses the JSch (**Java Secure Channel**) library from JCraft to SSH to the destination server and start the service. This requires the following additional Maven dependency

```
<dependency>
    <groupId>com.jcraf</groupId>
    <artifactId>jsch</artifactId>
    <version>0.1.53</version>
</dependency>
```

A walkthrough of the life cycle manager code

- The current SSH implementation is kept simple enough as we will change this in future chapters
- In this example, **DeploymentEngine** sends the following command over the SSH library on the target machine:

```
String command ="java -jar -Dserver.port=8091 ./work/  
codebox/chapter6/chapter6.search/target/search-1.0.jar";
```

- Integration with **Nexus** happens from the target machine using Linux scripts with Nexus CLI or using curl
 - In this example, we will not explore Nexus

A walkthrough of the life cycle manager code

- The next step is to change the Search microservice to expose a new gauge for TPM
 - We have to change all the microservices developed earlier to submit this additional metric.
 - We will only examine Search in this homework, but in order to complete it, all the services have to be updated
- In order to get the **gauge.servo.tpm** metrics, we have to add **TPMCounter** to all the microservices

```
class TPMCounter {  
    LongAdder count;  
    Calendar expiry = null;  
  
    TPMCounter(){  
        reset();  
    }  
  
    void reset (){  
        count = new LongAdder();  
        expiry = Calendar.getInstance();  
        expiry.add(Calendar.MINUTE, 1);  
    }  
  
    boolean isExpired(){  
        return Calendar.getInstance().after(expiry);  
    }  
  
    void increment(){  
        if(isExpired()){  
            reset();  
        }  
        count.increment();  
    }  
}
```

A walkthrough of the life cycle manager code

- The following code needs to be added to **SearchController** to set the **tpm** value:

```
class SearchRestController {  
    TPMCounter tpm = new TPMCounter();  
    @Autowired  
    GaugeService gaugeService;  
    //other code
```

- The following code is from the get REST endpoint (the search method) of **SearchRestController**, which submits the **tpm** value as a gauge to the actuator endpoint:

```
tpm.increment();  
gaugeService.submit("tpm", tpm.count.intValue());
```

Running the life cycle manager

- Edit `DeploymentEngine.java` and update the password to reflect the machine's password, as follows. This is required for the SSH connection: `session.setPassword("rajeshrv");`
- Build all the projects by running Maven from the root folder via the following command:

```
mvn -Dmaven.test.skip=true clean install
```

- Then, run RabbitMQ, as follows: `./rabbitmq-server`
- Ensure that the Config server is pointing to the right configuration repository. We need to add a property file for the life cycle manager.
- Run the following commands from the respective project folders:

```
java -jar target/config-server-0.0.1-SNAPSHOT.jar
```

```
java -jar target/eureka-server-0.0.1-SNAPSHOT.jar
```

```
java -jar target/lifecycle-manager-0.0.1-SNAPSHOT.jar
```

```
java -jar target/search-1.0.jar
```

```
java -jar target/search-apigateway-1.0.jar
```

```
java -jar target/website-1.0.jar
```

Running the life cycle manager

- Once all the services are started, open a browser window and load <http://localhost:8001>
- Execute the flight search 11 times, one after the other, within a minute. This will trigger the decision engine to instantiate another instance of the Search microservice.
- Open the Eureka console (<http://localhost:8761>) and watch for a second **SEARCH-SERVICE**. Once the server is started

Instances currently registered with Eureka				
Application	AMIs	Availability Zones	Status	
LIFECYCLE-MANAGER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.0.106:lifecycle-manager-service:9090	
SEARCH-APIGATEWAY	n/a (1)	(1)	UP (1) - 192.168.0.106:search-apigateway:8095	
SEARCH-SERVICE	n/a (2)	(2)	UP (2) - 192.168.0.106:search-service:8091 , 192.168.0.106:search-service:8090	
TEST-CLIENT	n/a (1)	(1)	UP (1) - 192.168.0.106:test-client:8001	