Towards Autonomic Microservices

Claudio Guidi

italianaSoftware s.r.l.

Microservice Conference 2020
Bologna, 8-10 September 2020

italianaSoftware

Claudio Guidi

About me

Co-creator and co-leader of the Jolie programming language project.

Jolie is a service oriented programming language which allows for natively programming services.

http://jolie-lang.org

Founder and CEO of italianaSoftware.

The mission of the company is digitalizing business processes increasing their flexibility and resilence through the usage of microservices based distributed systems.

http://italianasoftware.com

Member of the Council of the Microservices-Community.

The Microservices Community is a European-based international community interested in the software paradigm of Microservices.

http://microservices.community

Outline

- Introduction
  Autonomic computing and microservices
- Presentation of a PoC
  I implemented a jolie based simple demo which shows how an autonomic microservice could be implemented
- An architectural proposition
  Autonomic microservices can be built on top of an autonomic enhanced architecture. Here I show an initial proposition for it.
- Conclusions

italianaSoftware

Introduction

Autonomic Computing

Systems manage themselves according to an administrator’s goals.

italianaSoftware

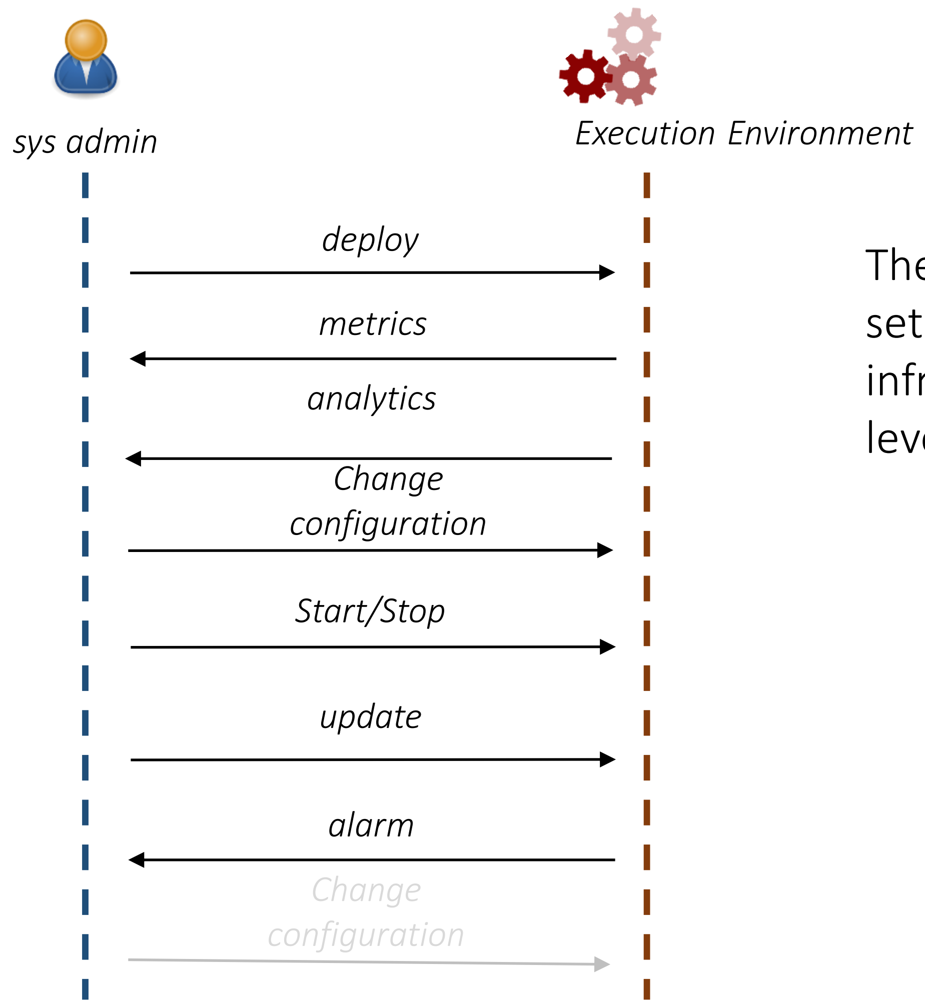
Kephart, Jeffrey & Chess, D.M.. (2003). **The Vision Of Autonomic Computing. Computer**. 36. 41- 50. 10.1109/MC.2003.1160055

Systems manage themselves according to an administrator’s goals. New components integrate as effortlessly as a new cell establishes itself in the human body. These ideas are not science fiction, but elements of the grand challenge to create self-managing computing systems.

Concept	Current Computing (2003)	Autonomic computing
Self-configuration	Corporate data centers have multiple vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmers weeks.	System automatically detects, diagnoses, and repairs localized software and hardware problems
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent systemwide failures.

The management of a running system

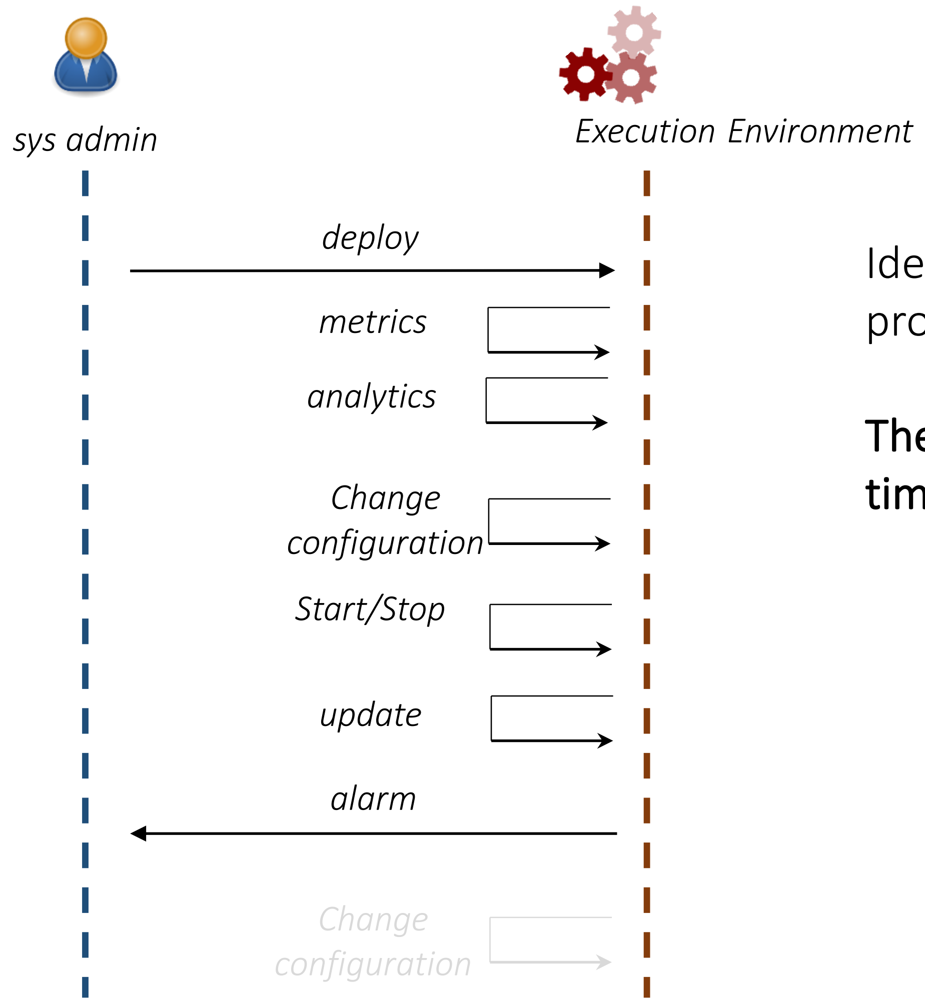
The key point



The management of a running system can be seen as a continuous set of interactions between the sys admin and the target infrastructure in order to keep the applications running with a high level of quality targeting the business requirements.

The autonomic scenario

A lot of interactions between admin and the execution environment are automatically managed by the system itself



Ideally, an autonomic system is able to self-configure, self-heal, self-protect and self-organize itself depending on its status.

The main target is reducing human interactions in order to reducing time cost and increasing efficiency.

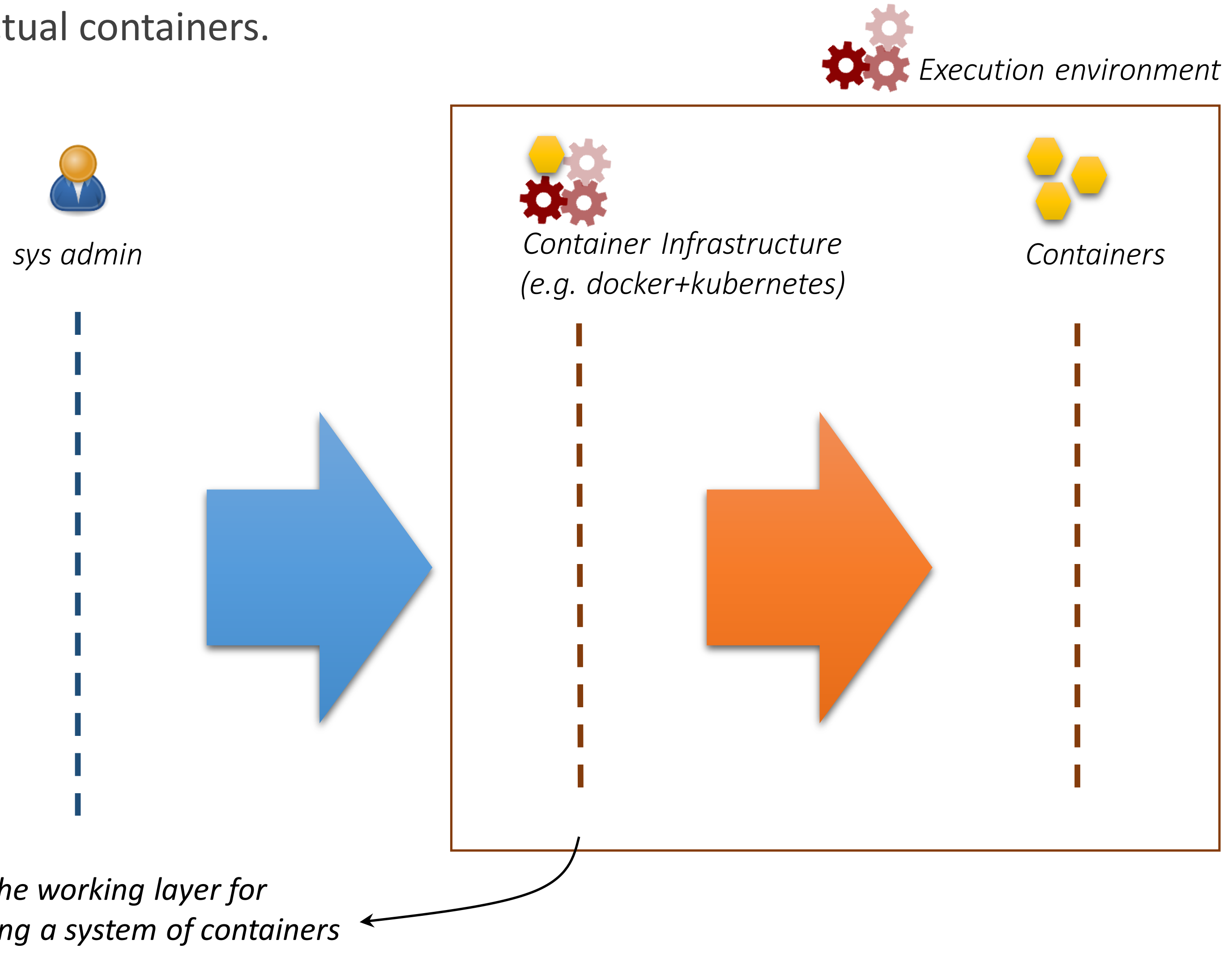
Autonomic computing and containerization

The execution environment can be seen as the composition of a specialized container management infrastructure and the actual containers.

Containerization enables component abstraction to containers and changes the rules on how a system is managed today. **A system is just a set of interacting containers**.

Sys admins can automatize a lot of operations by configuring the orchestration platform.

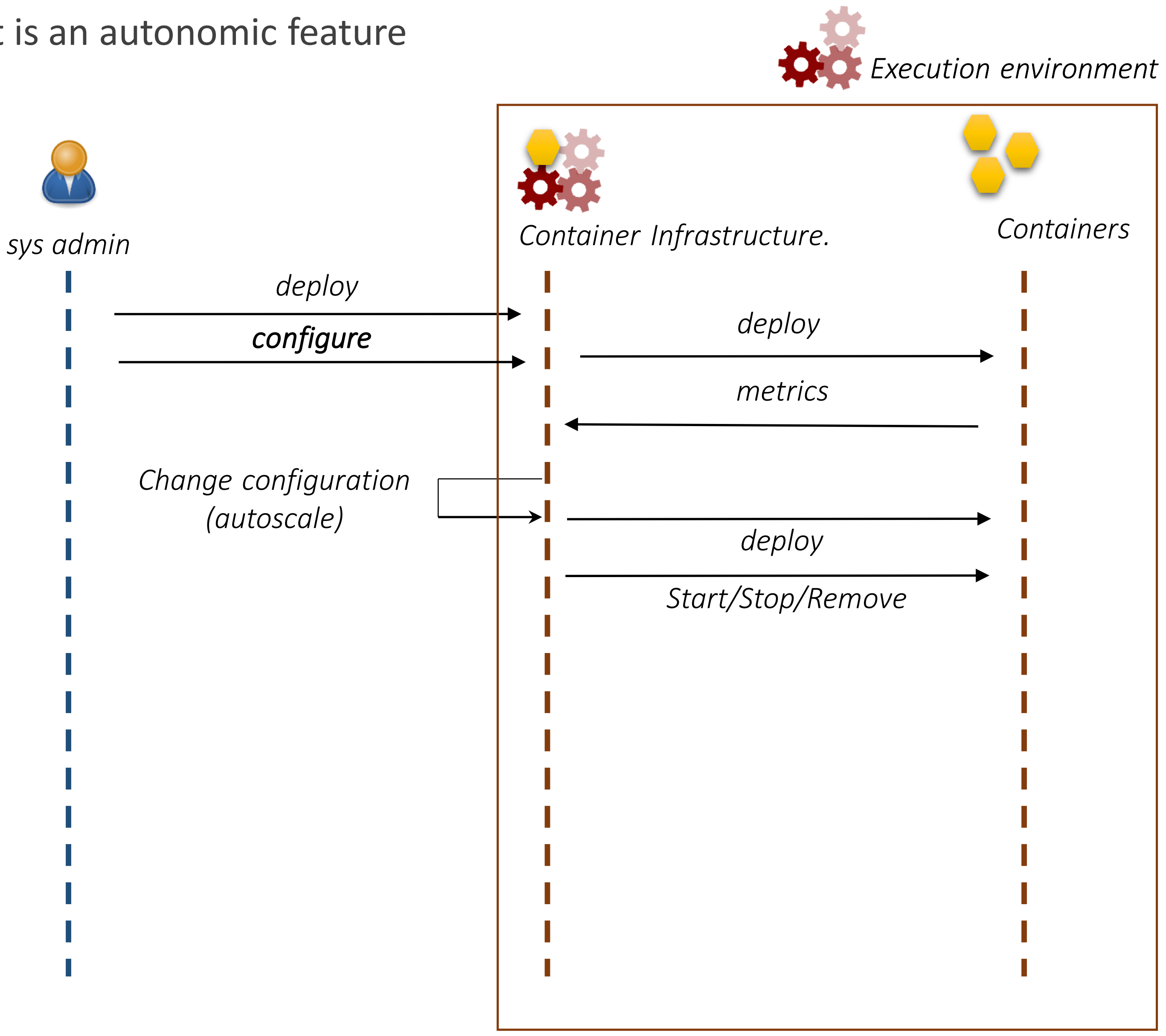
Containerization is **massifying** the development of distributed applications. **<u>The development of an application must be easy and agnostic with respect the target infrastructure.</u>**



Execution environment
sys admin
Container Infrastructure (e.g. docker+kubernetes)
Containers
This is the working layer for managing a system of containers

Autoscaling

It is an autonomic feature

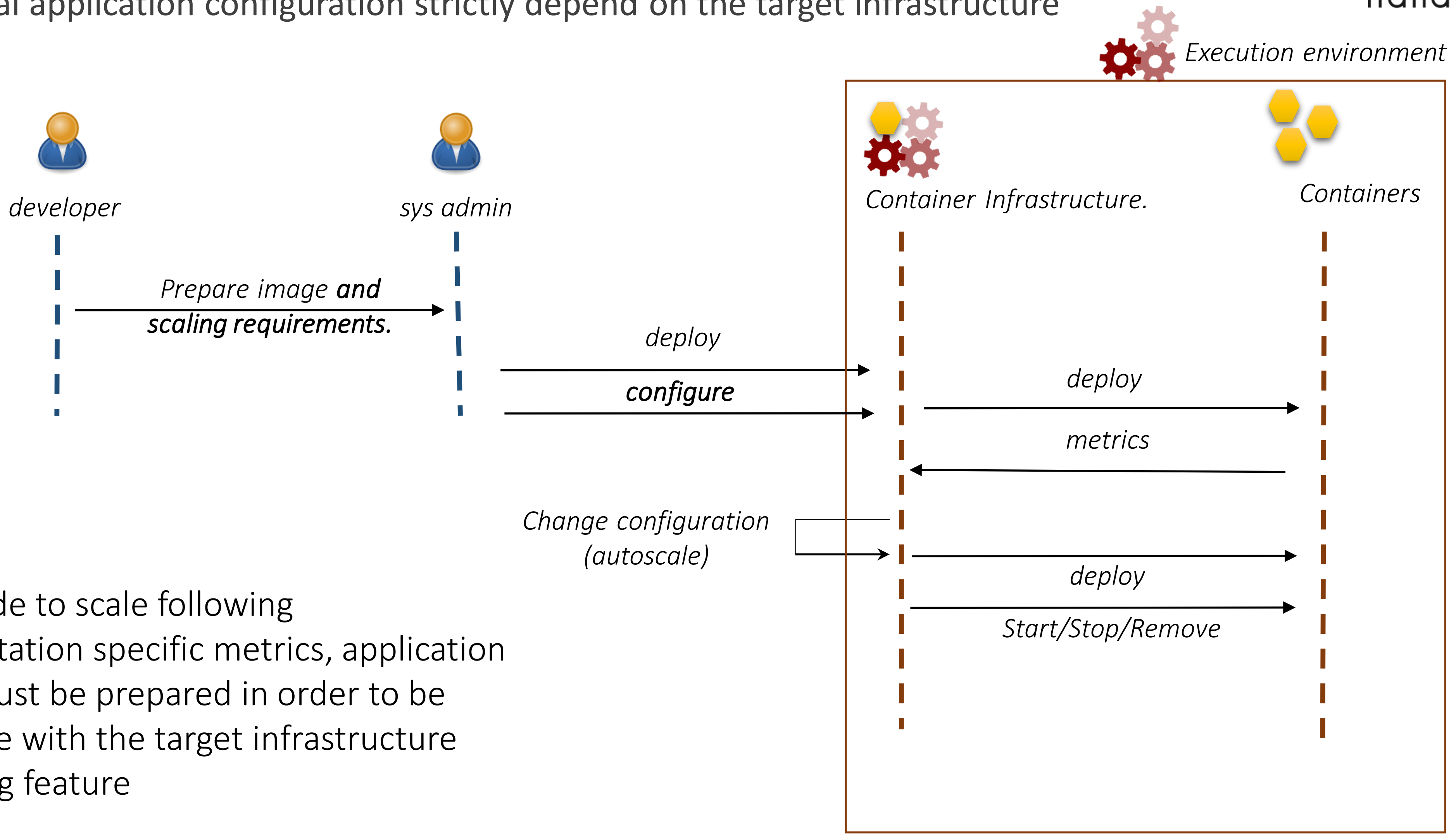


The **autoscaling** is maybe the first autonomic feature a containerization system can provide. As an example, Kubernetes is able to autoscale a system depending on some metrics extracted from single components.

Containers are just components that cannot play any actions for changing their own structure. They are manipulated by the infrastructure.

Preparing an autoscaling application

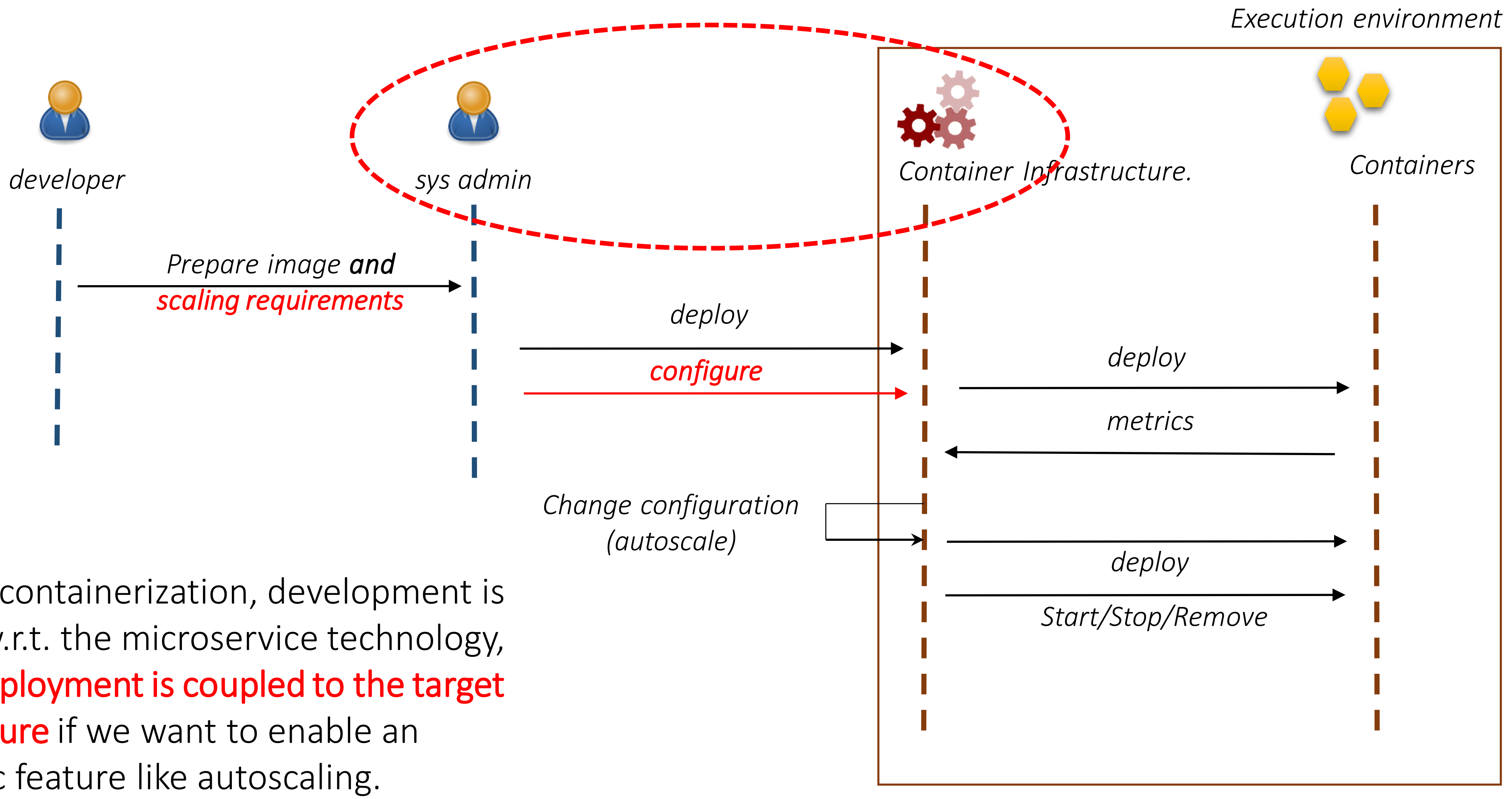
The final application configuration strictly depend on the target infrastructure



If we decide to scale following implementation specific metrics, application metrics must be prepared in order to be compatible with the target infrastructure autoscaling feature

Preparing an autoscaling application

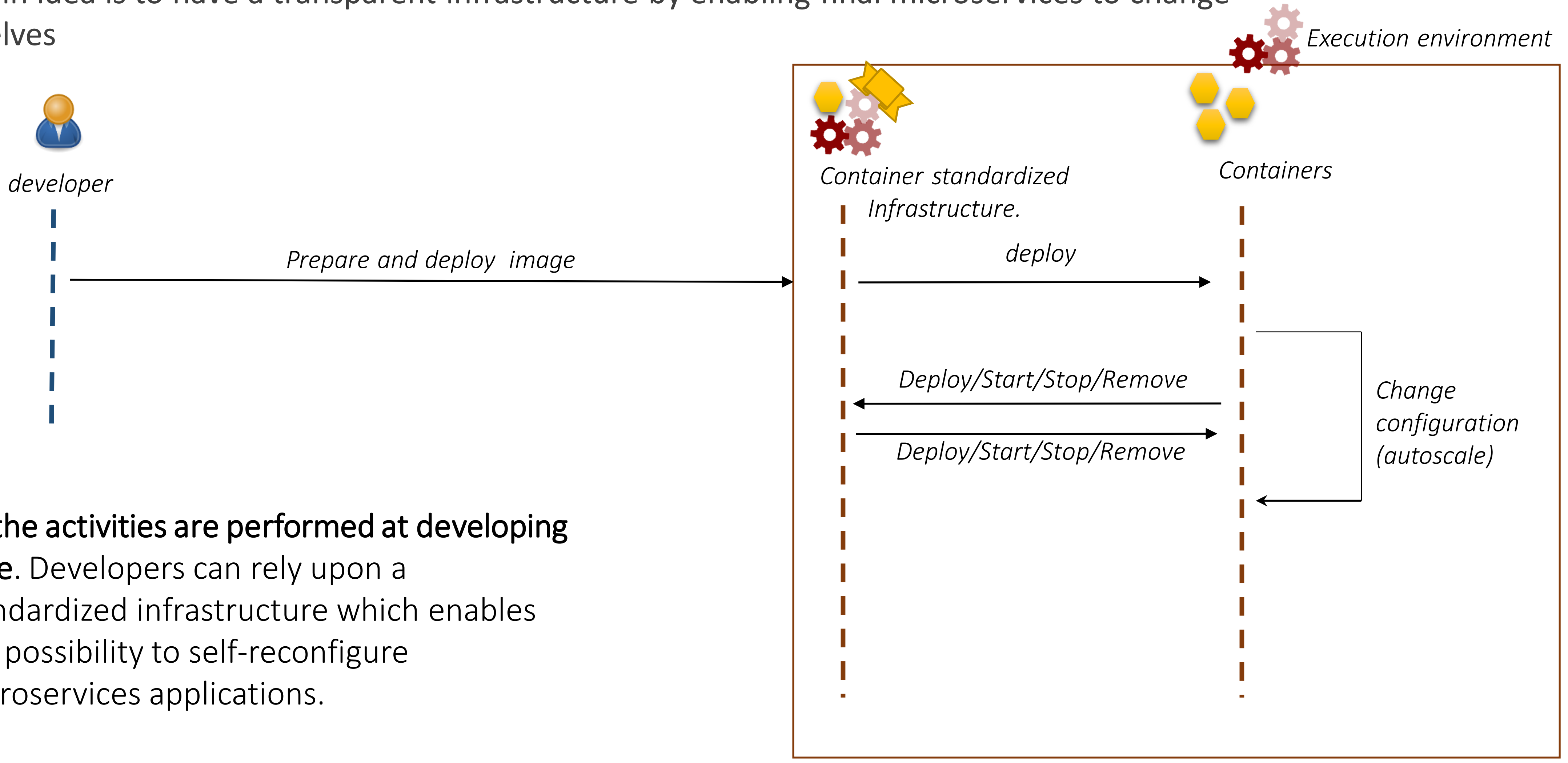
The final application configuration strictly depend on the target infrastructure



Thanks to containerization, development is agnostic w.r.t. the microservice technology, but the deployment is coupled to the target infrastructure if we want to enable an autonomic feature like autoscaling.

Autonomic microservices

The main idea is to have a transparent infrastructure by enabling final microservices to change themselves



All the activities are performed at developing time. Developers can rely upon a standardized infrastructure which enables the possibility to self-reconfigure microservices applications.

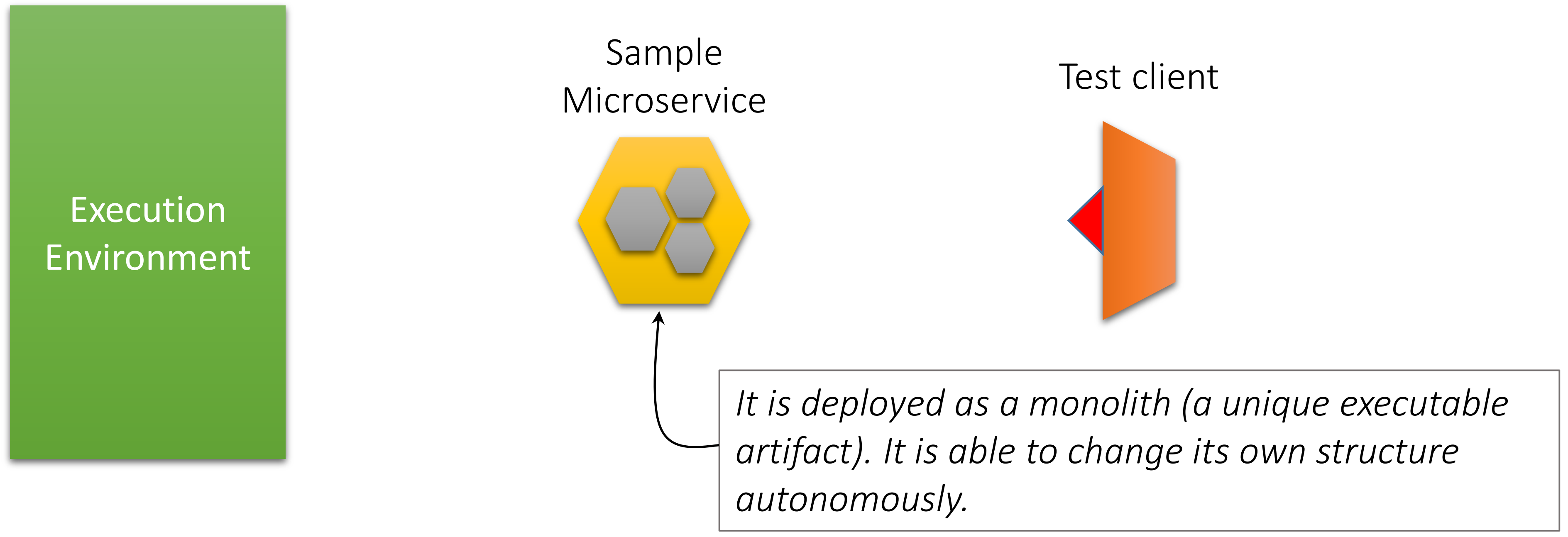
Presentation of a PoC

The PoC

A proof of concept demo developed with jolie

The code of the PoC is available at https://github.com/klag/autonomic-microservices

The main idea is to deploy a monolithic microservice which is able to request to scale one of its components in order to scale depending on some internal metrics.



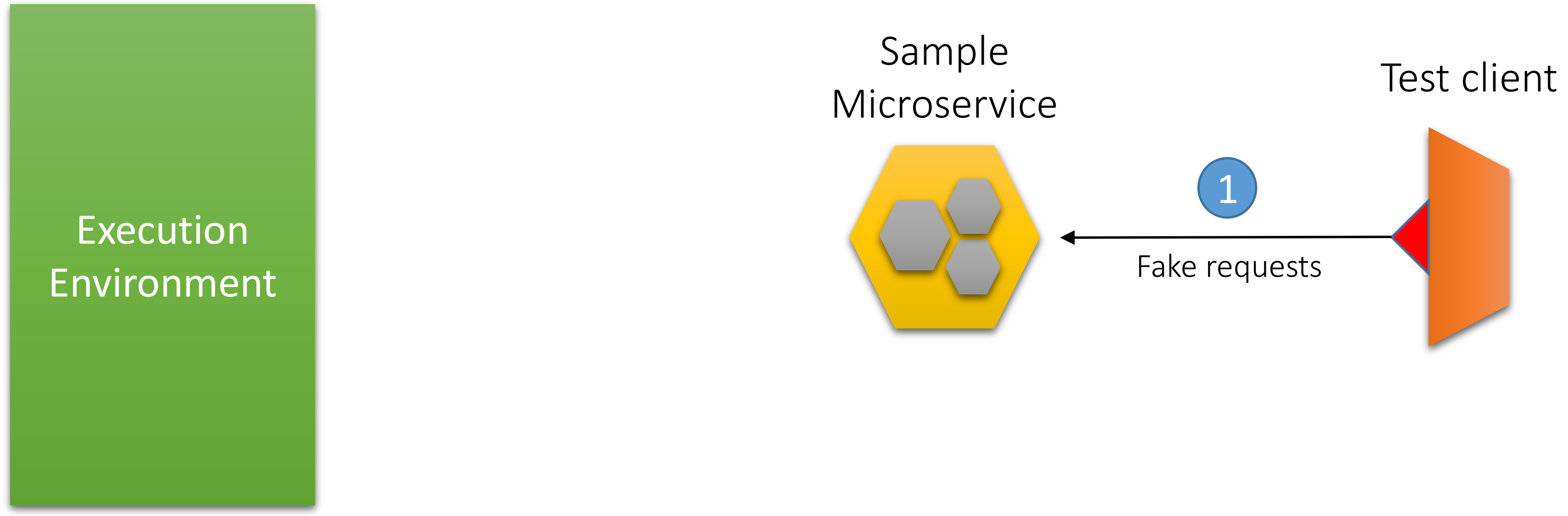
It is deployed as a monolith (a unique executable artifact). It is able to change its own structure autonomously.

The PoC: the main picture

A proof of concept demo developed with jolie

The code of the PoC is available at https://github.com/klag/autonomic-microservices

The client sends a bulk of requests to the sample microservice. When the microservice detects an increasing delay time in its responses, start an interaction with the Execution Environment in order to scale one of its internal components.

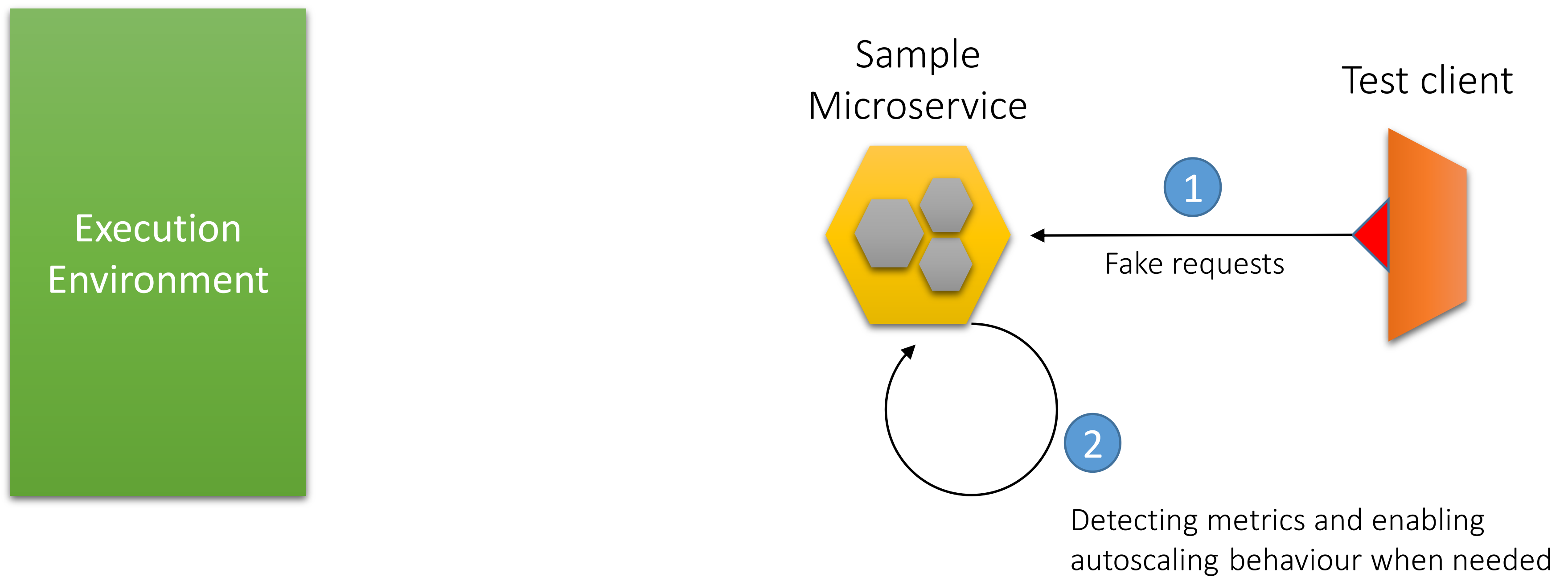


The PoC: the main picture

A proof of concept demo developed with jolie

The code of the PoC is available at https://github.com/klag/autonomic-microservices

The client sends a bulk of requests to the sample microservice. When the microservice detects an increasing delay time in its responses, start an interaction with the Execution Environment in order to scale one of its internal components.

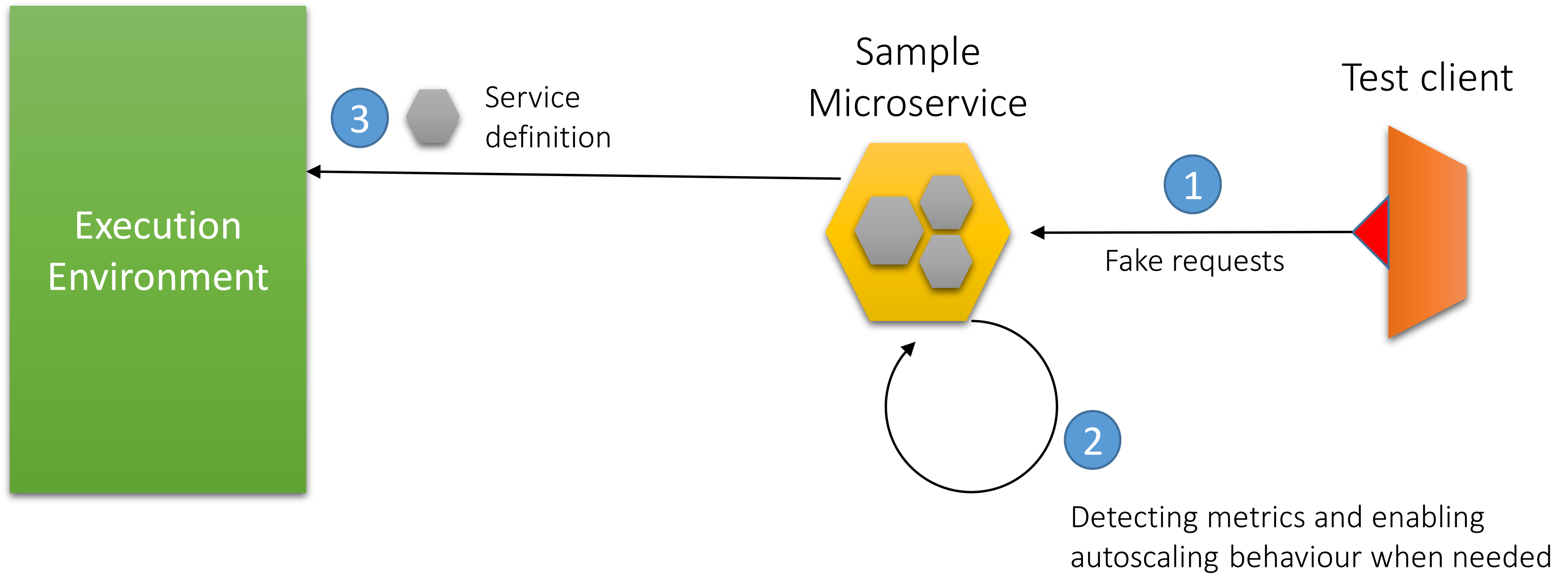


The PoC: the main picture

A proof of concept demo developed with jolie

The code of the PoC is available at https://github.com/klag/autonomic-microservices

The client sends a bulk of requests to the sample microservice. When the microservice detects an increasing delay time in its responses, start an interaction with the Execution Environment in order to scale one of its internal components.

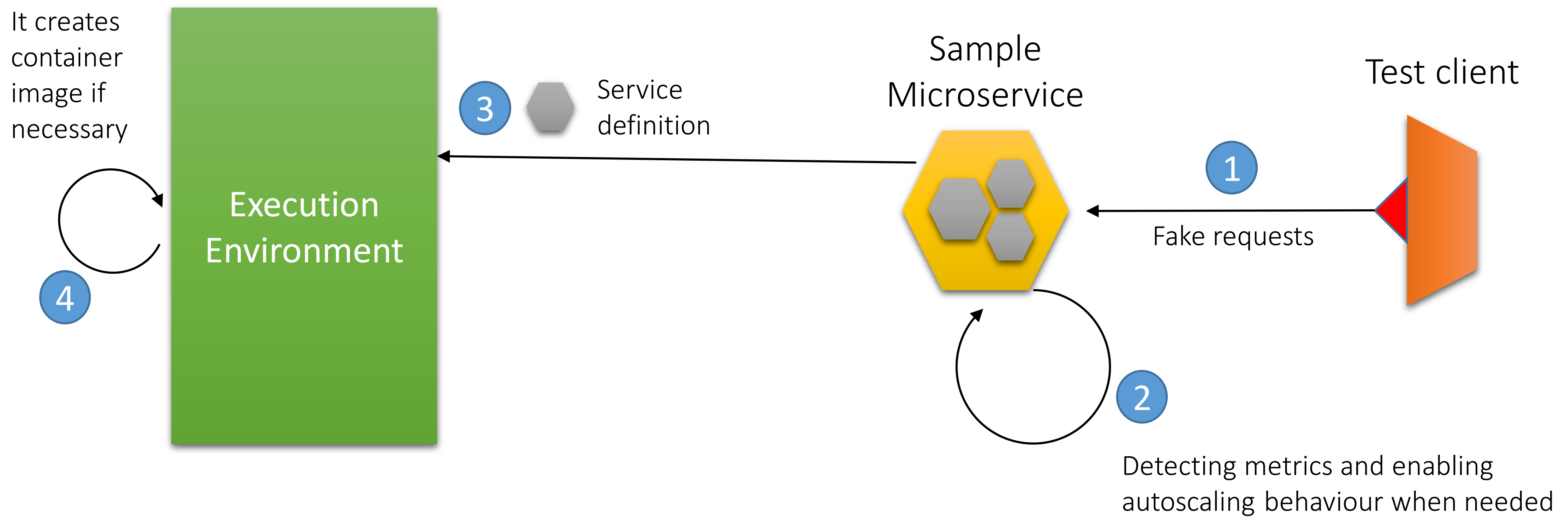


The PoC: the main picture

A proof of concept demo developed with jolie

The code of the PoC is available at https://github.com/klag/autonomic-microservices

The client sends a bulk of requests to the sample microservice. When the microservice detects an increasing delay time in its responses, start an interaction with the Execution Environment in order to scale one of its internal components.

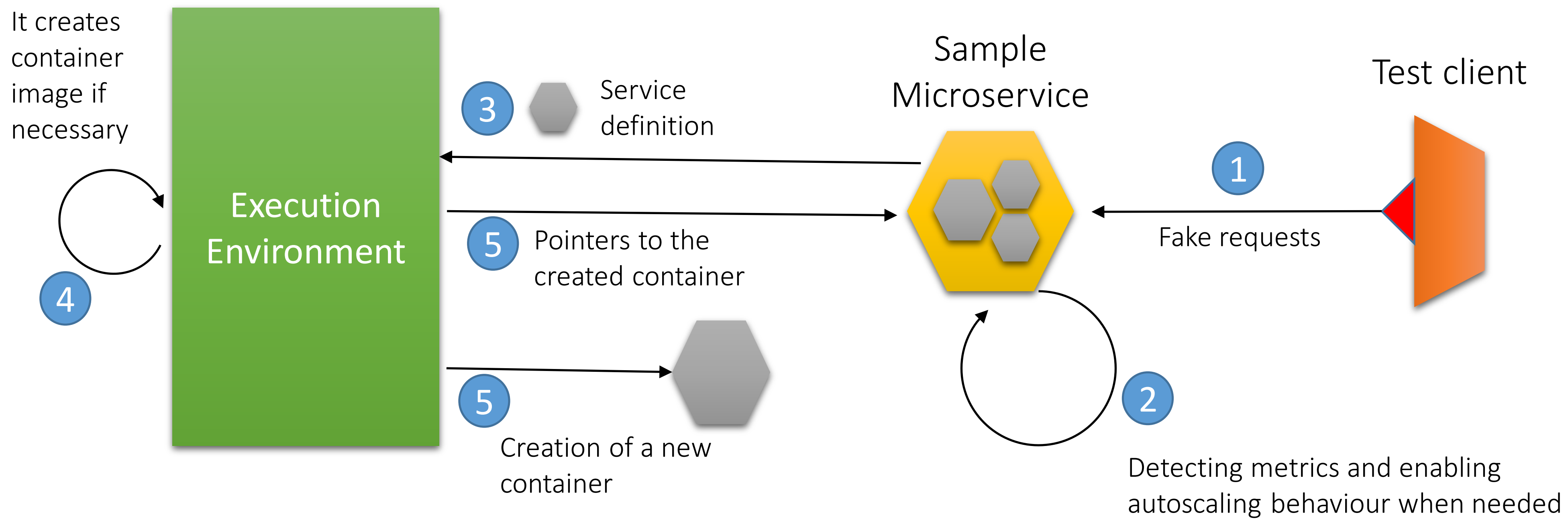

It creates container image if necessary
4
Execution Environment
3
Service definition
Sample Microservice
Test client
1
Fake requests
2
Detecting metrics and enabling autoscaling behaviour when needed

The PoC: the main picture

A proof of concept demo developed with jolie

The code of the PoC is available at https://github.com/klag/autonomic-microservices

The client sends a bulk of requests to the sample microservice. When the microservice detects an increasing delay time in its responses, start an interaction with the Execution Environment in order to scale one of its internal components.

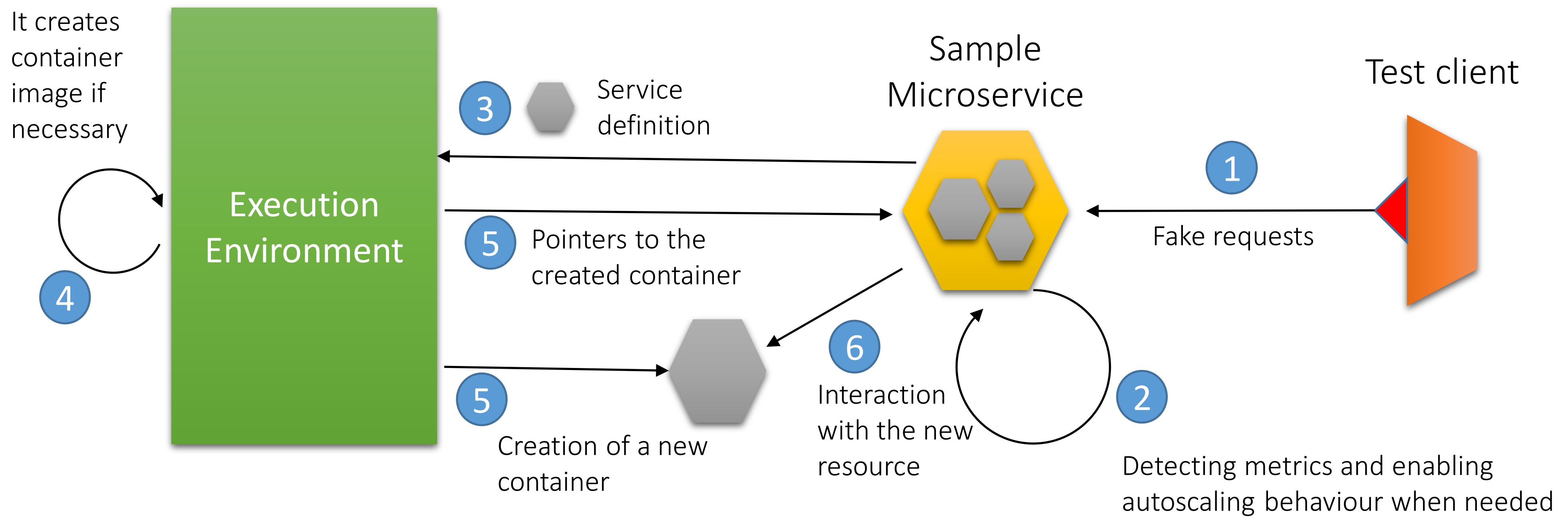

It creates container image if necessary
4
Execution Environment
3
Service definition
5
Pointers to the created container
5
Creation of a new container
Sample Microservice
Test client
1
Fake requests
2
Detecting metrics and enabling autoscaling behaviour when needed

The PoC: the main picture

A proof of concept demo developed with jolie

The code of the PoC is available at https://github.com/klag/autonomic-microservices

The client sends a bulk of requests to the sample microservice. When the microservice detects an increasing delay time in its responses, start an interaction with the Execution Environment in order to scale one of its internal components.

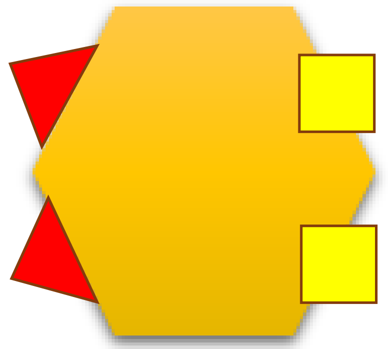

It creates container image if necessary
4
Execution Environment
3
Service definition
5
Pointers to the created container
5
Creation of a new container
6
Interaction with the new resource
Sample Microservice
2
Detecting metrics and enabling autoscaling behaviour when needed
1
Fake requests
Test client

Jolie background

I exploited some specific features of the Jolie programming language

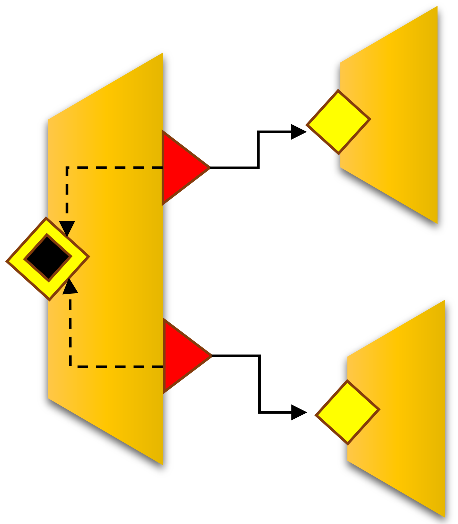
italianaSoftware

Service Orientation



Thinking in services
Jolie crystallises the programming concepts of service-oriented computing as linguistic constructs. The basic building blocks of software are not objects or functions, but rather services that can be relocated and replicated as needed. A composition of services is a service.
https://www.jolie-lang.org/

Aggregation and Couriers



Architectural primitives
In Jolie a service can play the role of aggregator, merging the inputPorts of other services into its own ones.

A courier is a process which is run into an aggregator before forwarding an incoming message to an aggregated service.

https://jolielang.gitbook.io/docs/language-tools-and-standard-library/architectural-composition/aggregation

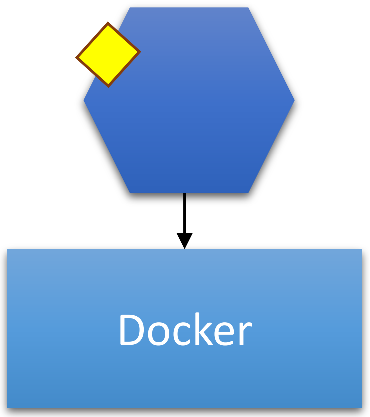
Embedding

Deploying as a monolith
Jolie services can embed other jolie services. A system of services can be deployed as a monolith within a unique executable artifact.

A service can be packed within an archive file whose extension is .jap

https://*jolielang.gitbook.io/docs/language-tools-and-standard-library/architectural-composition/embedding*

Jocker

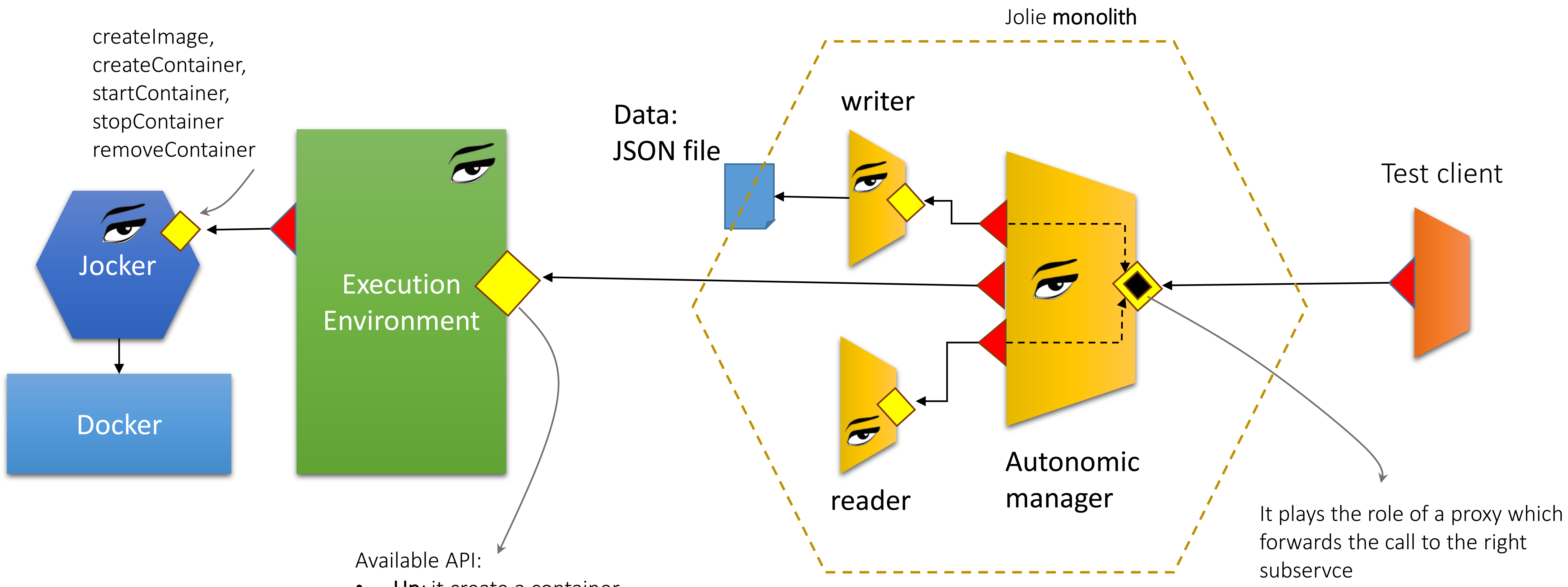


Orchestrating Docker
Jocker is an experimental project made in Jolie which offers a Jolie wrapper to Docker API. Jocker is a container itself, and It allows to orchestrate a docker server by using Jolie calls.

https://jolielang.gitbook.io/docs/language-tools-and-standard-library/containerization/docker/jocker

The PoC: the architecture

I exploited some specific features of the Jolie programming language



Available API:

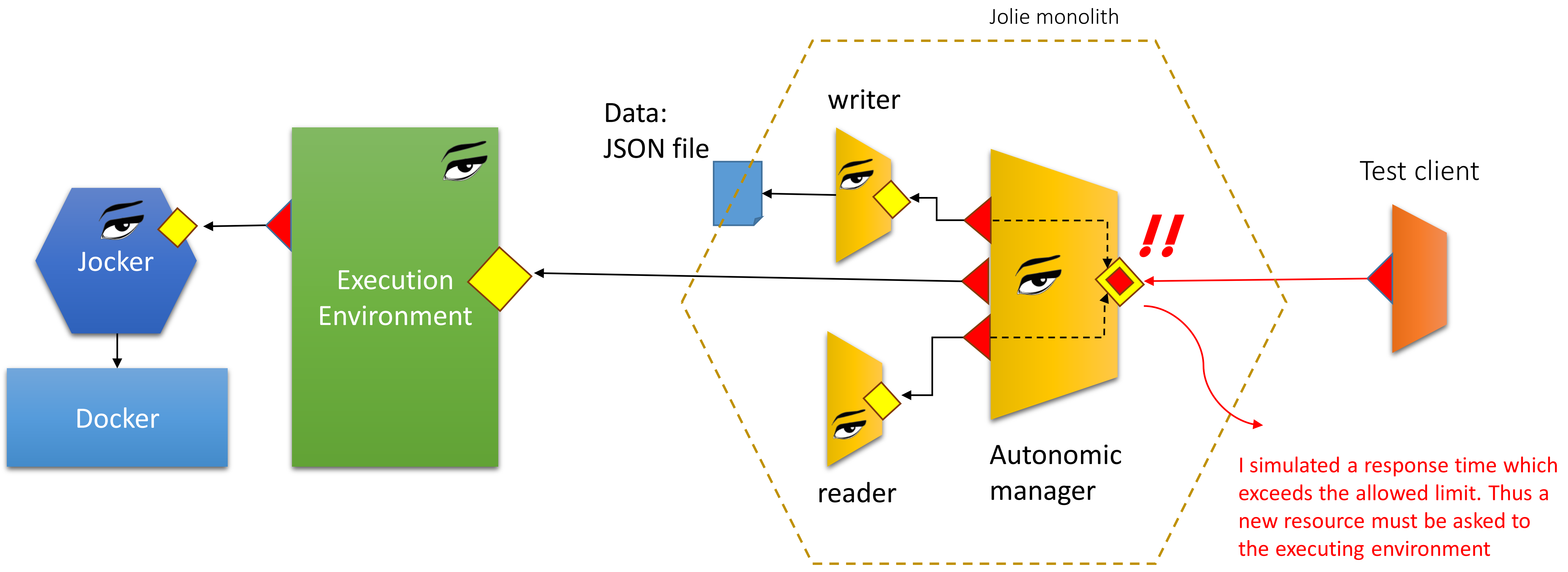
- **Up**: it create a container
- **Down**: it removes a container

Developed in Jolie

Scaling the reader

The reader can be auto scaled when the responding time exceeds a limit

italianaSoftware



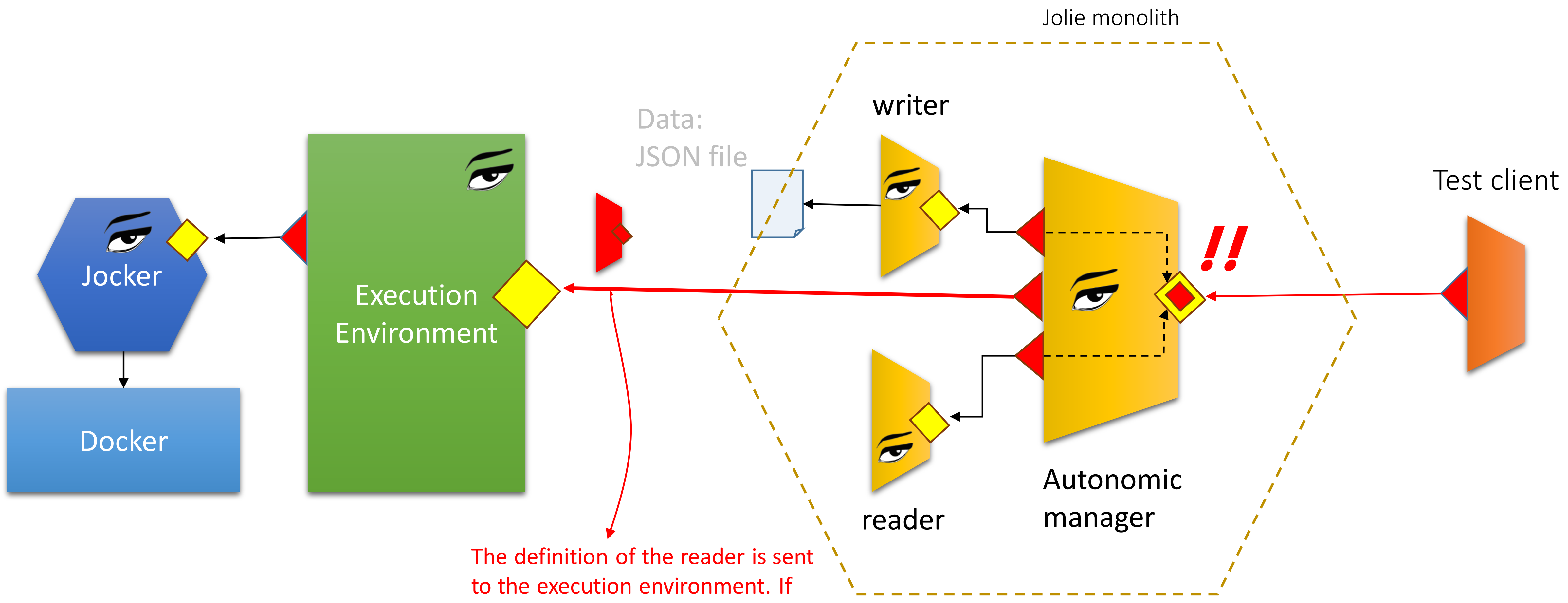
I simulated a response time which exceeds the allowed limit. Thus a new resource must be asked to the executing environment

Developed in Jolie

Scaling the reader

A definition of the reader is sent to the execution environment

italianaSoftware

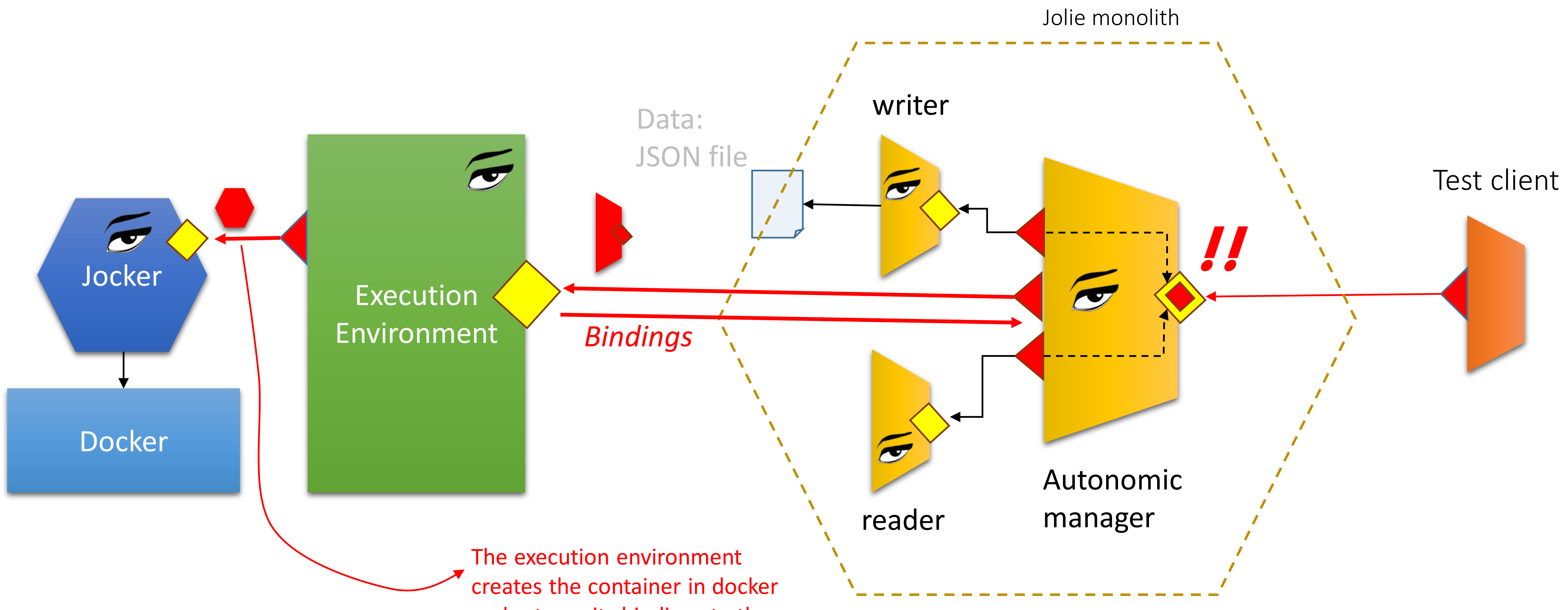


The definition of the reader is sent to the execution environment. If the image does not exist, the service requests for the creation of the image, then create and starts the container

Developed in Jolie

Scaling the reader

A new docker container for the reader is requested.

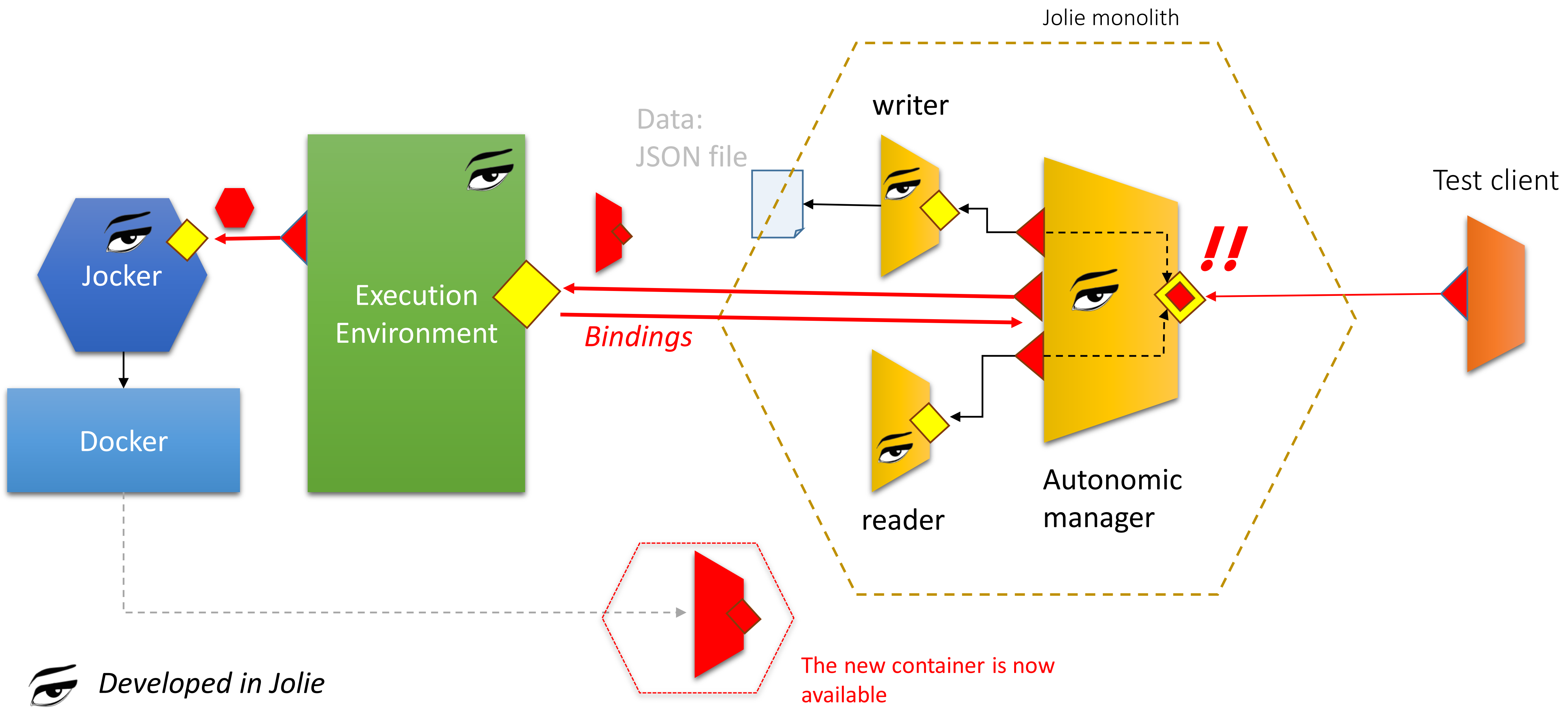


The execution environment creates the container in docker and returns its bindings to the microservice

Developed in Jolie

Scaling the reader

A new container for the reader is created.



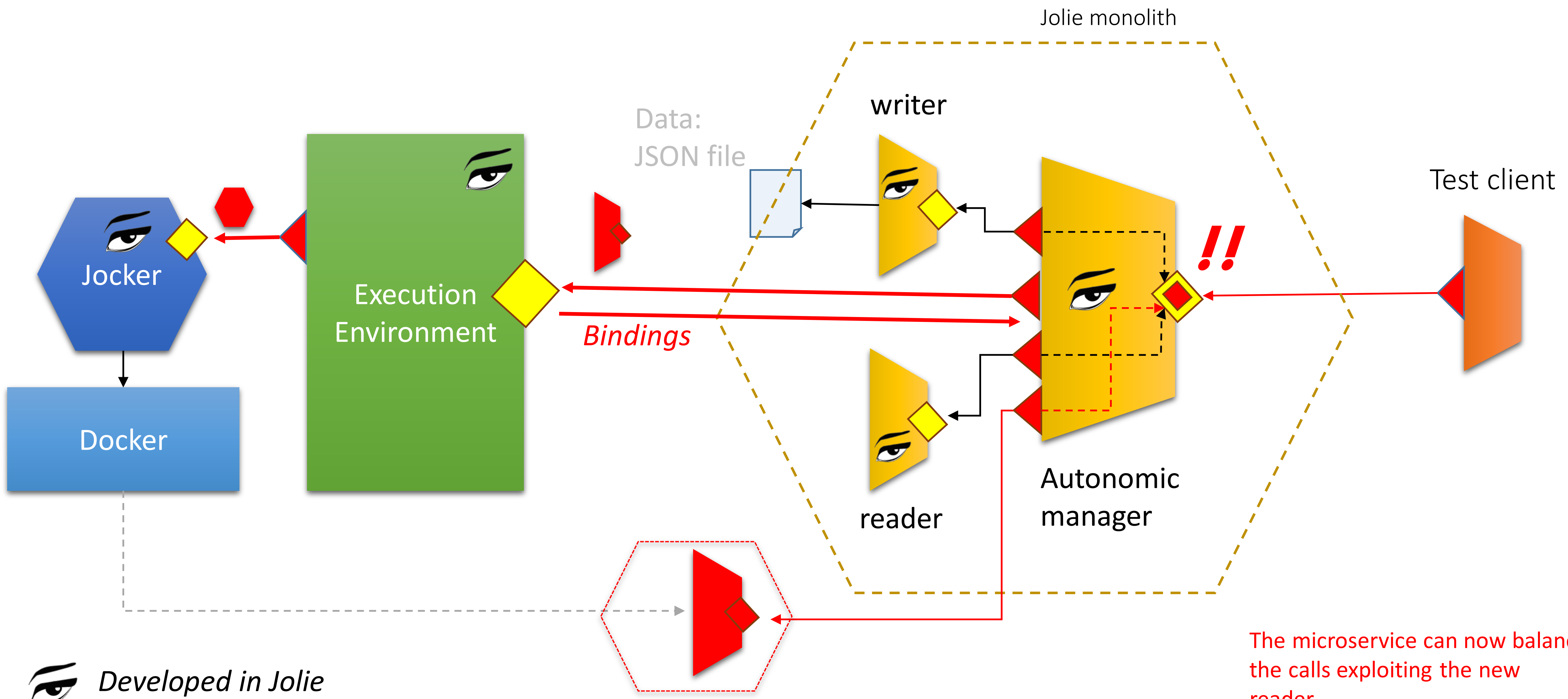
Developed in Jolie

italianaSoftware

Scaling the reader

The service starts to balance the calls also to the new readers

italianaSoftware



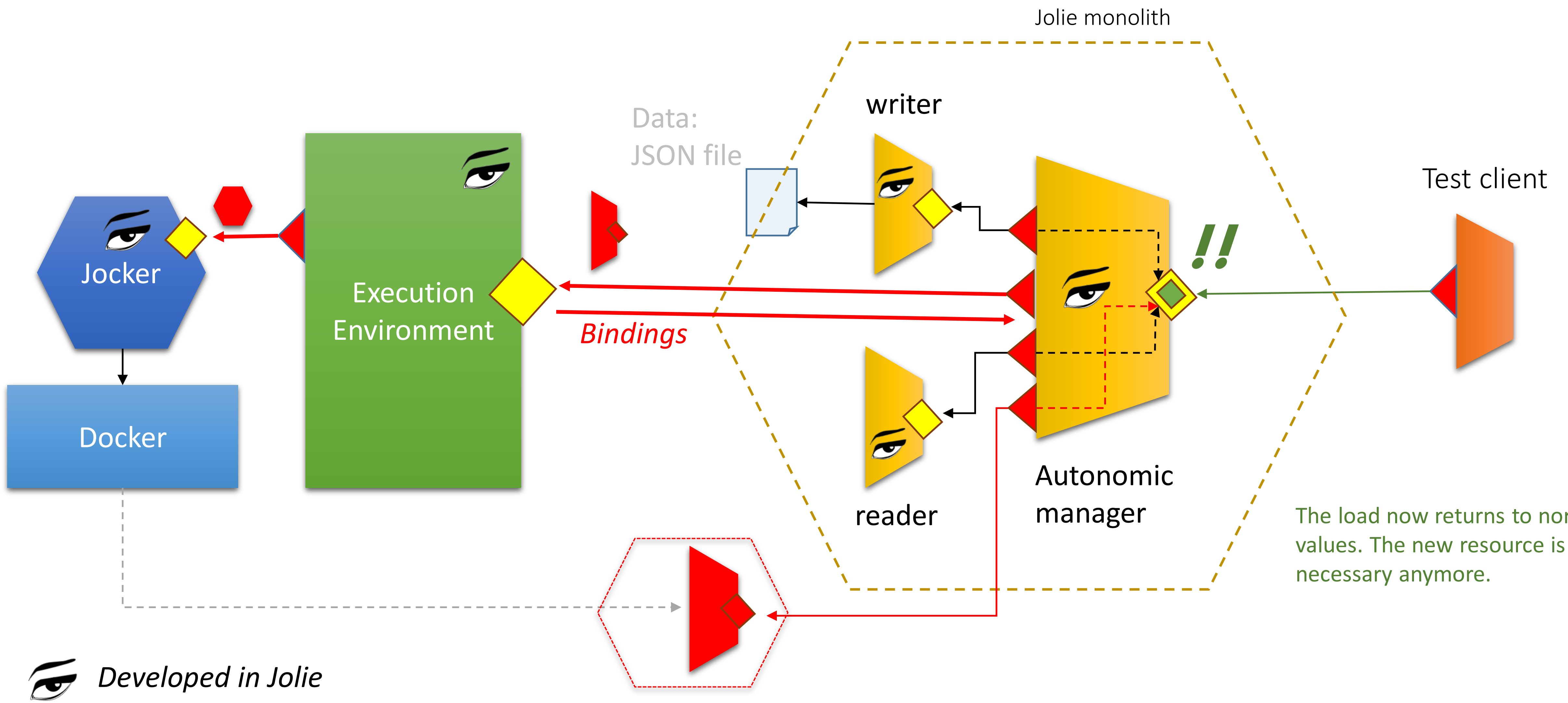
The microservice can now balance the calls exploiting the new reader.

Developed in Jolie

Scaling the reader

The load returns to normal

italianaSoftware

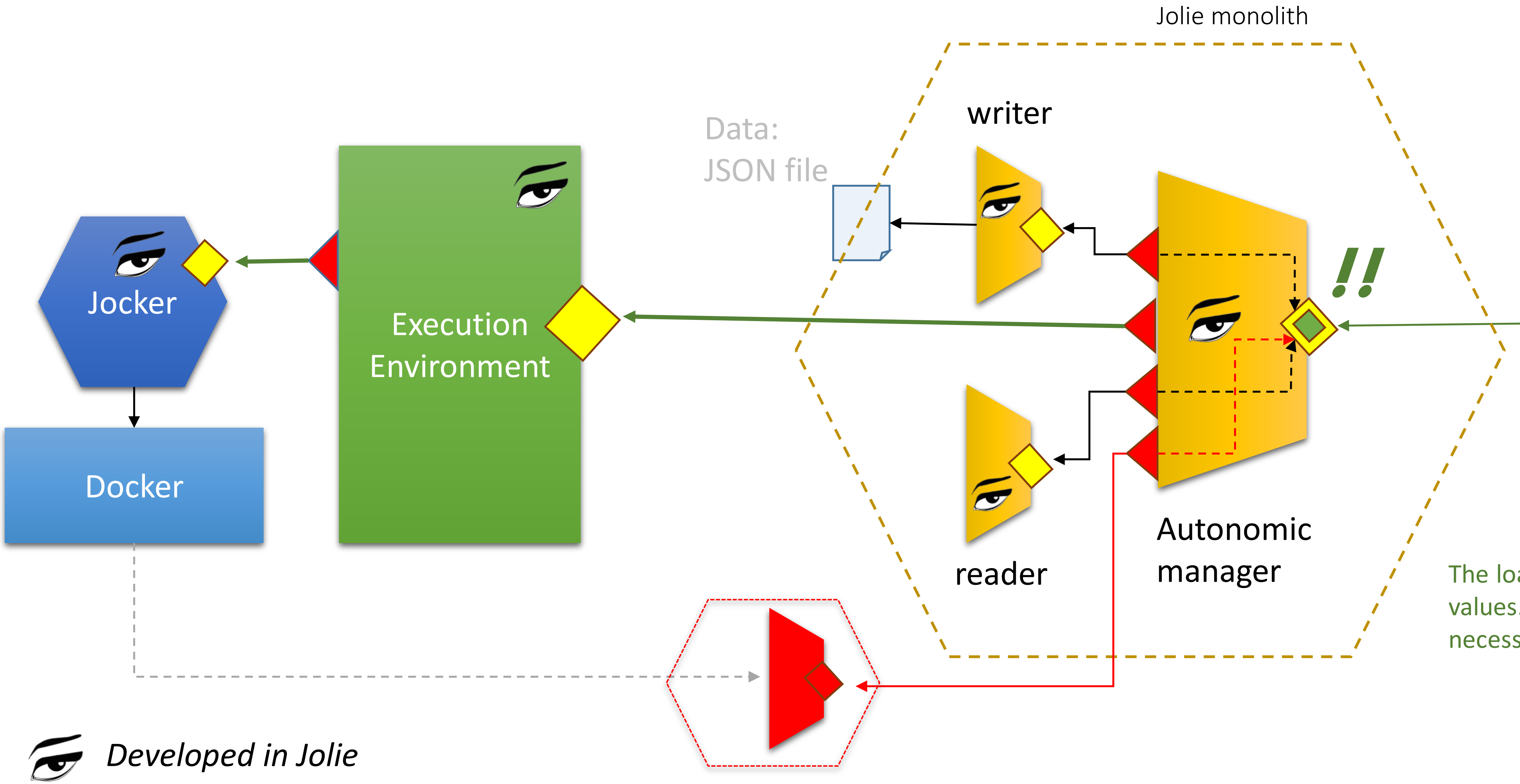


The load now returns to normal values. The new resource is not necessary anymore.

Developed in Jolie

Scaling the reader

The service asks for reducing the resources

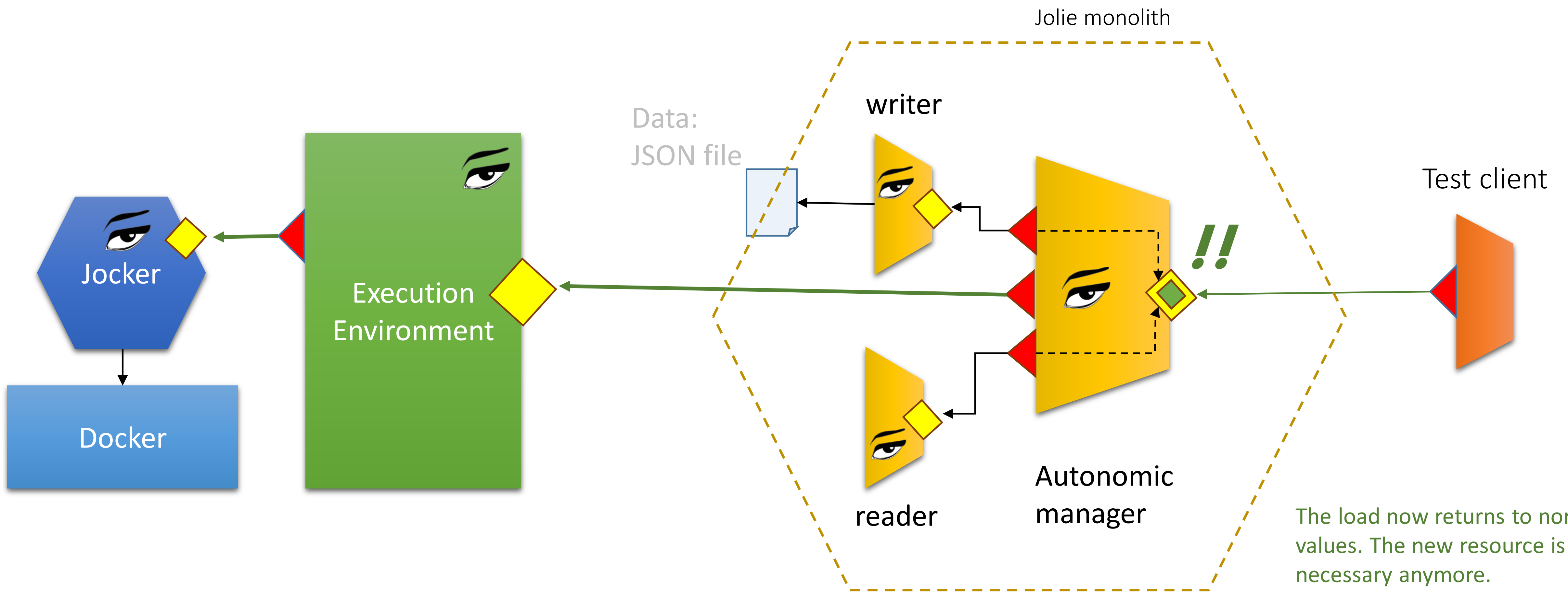


The load now returns to normal values. The new resource is not necessary anymore.

Developed in Jolie

Scaling the reader

The new container is removed

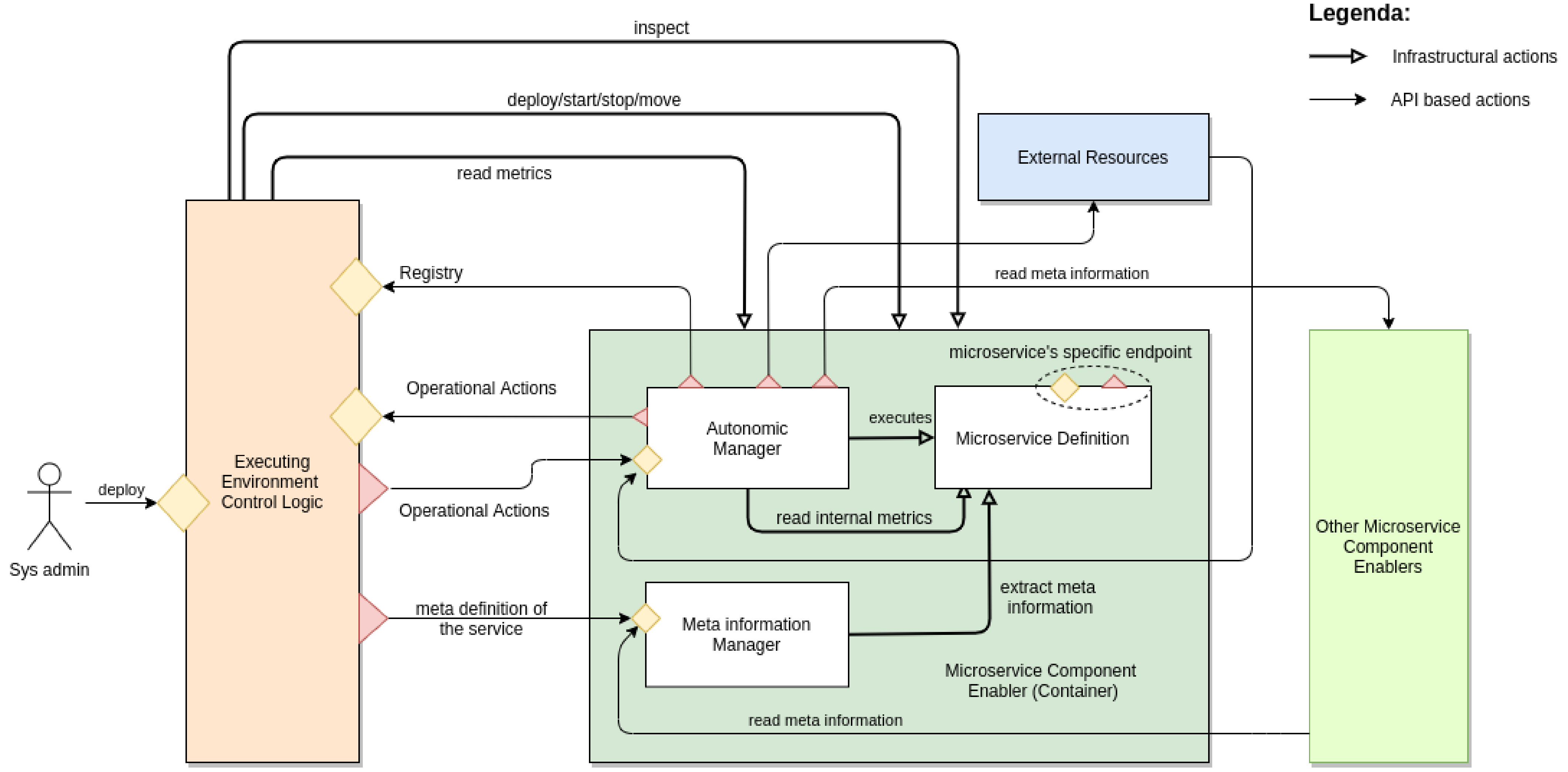


The load now returns to normal values. The new resource is not necessary anymore.

Developed in Jolie

An architectural proposition

A first proposition for a standardized architecture



Important points

The standardization of an architecture for autonomic microservices is very challenging. There are several open points to be taken into account.

- **Resource registry and meta information:** which meta information must be collected at system level and what can be managed by the single services?
- **Standardization of the API and protocols:** all the API and the interaction protocols must be standardized
- **Security**: which are all the security aspects that must be taken into account? Are there new aspects to be considered?
- **Component visibility**: a new component created by an autonomic microservice could be private or public. Component visibility should be correctly modelled.
- ...

Conclusions

Conclusions

Triggerring a discussion within the microservices community

Autonomic microservices are very challenging. This talk aims at introducing the topic within the community in order to analyze its aspects.

Objectives of this talk:

- Triggerring a discussion about microservices and autonomic computing within the microservices community
- Hopefully, starting a stable discussion group within the community about this topic which involves both people from the academy and industry

Thanks