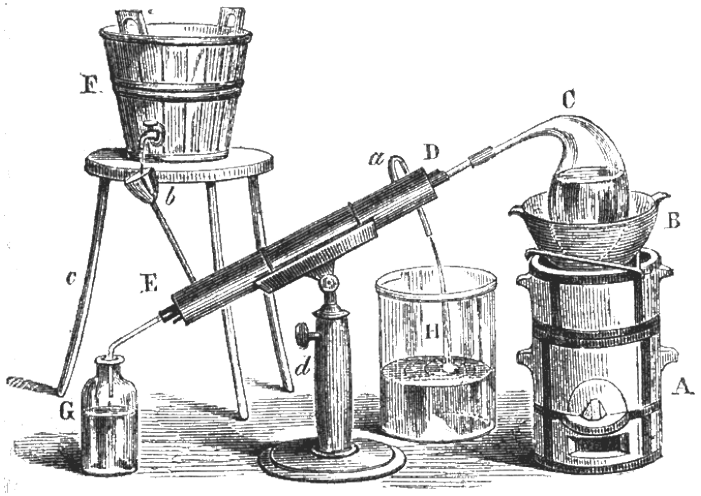
Odense, 26/10/2017

A linguistic approach to microservices

Claudio Guidi, italianaSoftware

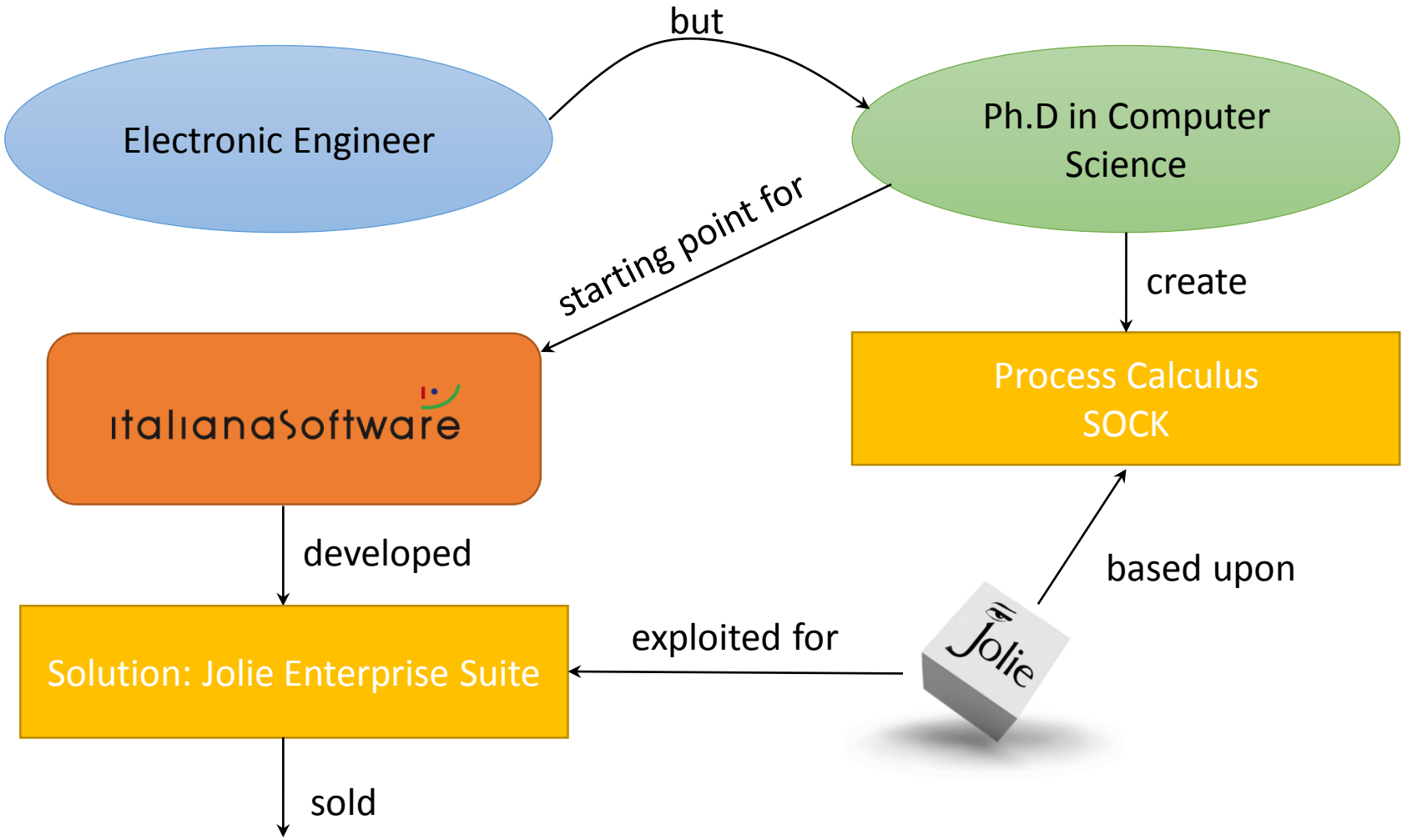


Distilling the core concepts of microservices into a programming language

italianaSoftware

imola
project partners

About me



italianaSoftware

imola

About italianaSoftware

italianaSoftware

Is part of

gruppoimola

imola sensiblelogic

Microservices centered
Product and technology centered
Target customers: manufacturing area

Consultancy centered
Finance area (banks and assurance)
Strong experience in SOA

Based in Imola

A linguistic approach to microservices

Interesting… but why?

italianaSoftware

imola

The nature of computational resources is going to change

A machine

The computational resource so far

Naturally not distributed.

Processes focused on **computation.**

Functions, procedures and objects**.**

A cloud of machines

The reference computational resource in the next years

Naturally distributed.

Processes focused on **communication**.

Services, microservices**,** nanoservices, **…**

italianaSoftware

imola

Containers are going to replace the idea of machine we know so far

Containerization
(cloud independence)

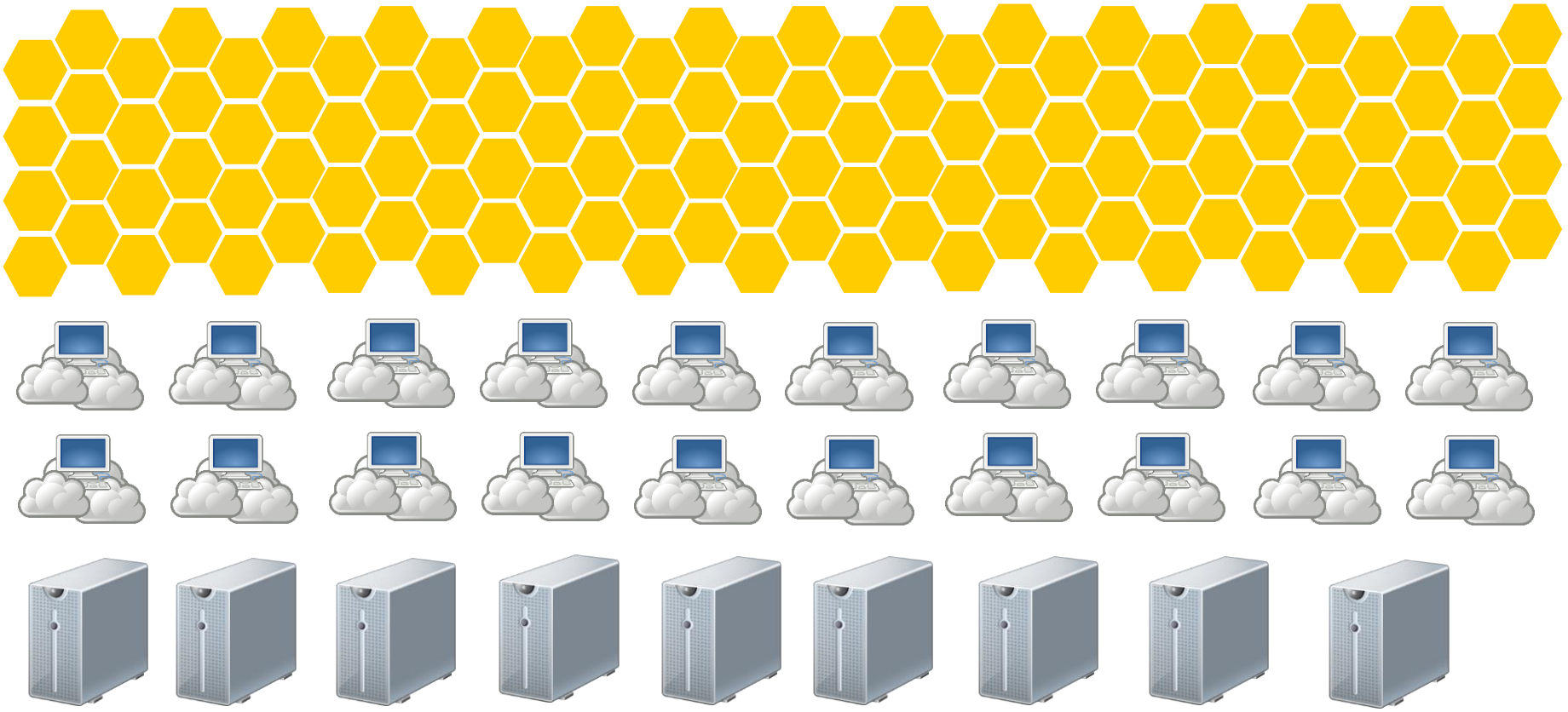
Virtualization
(hardware independence)

Real Computational Resource

italianaSoftware

imola

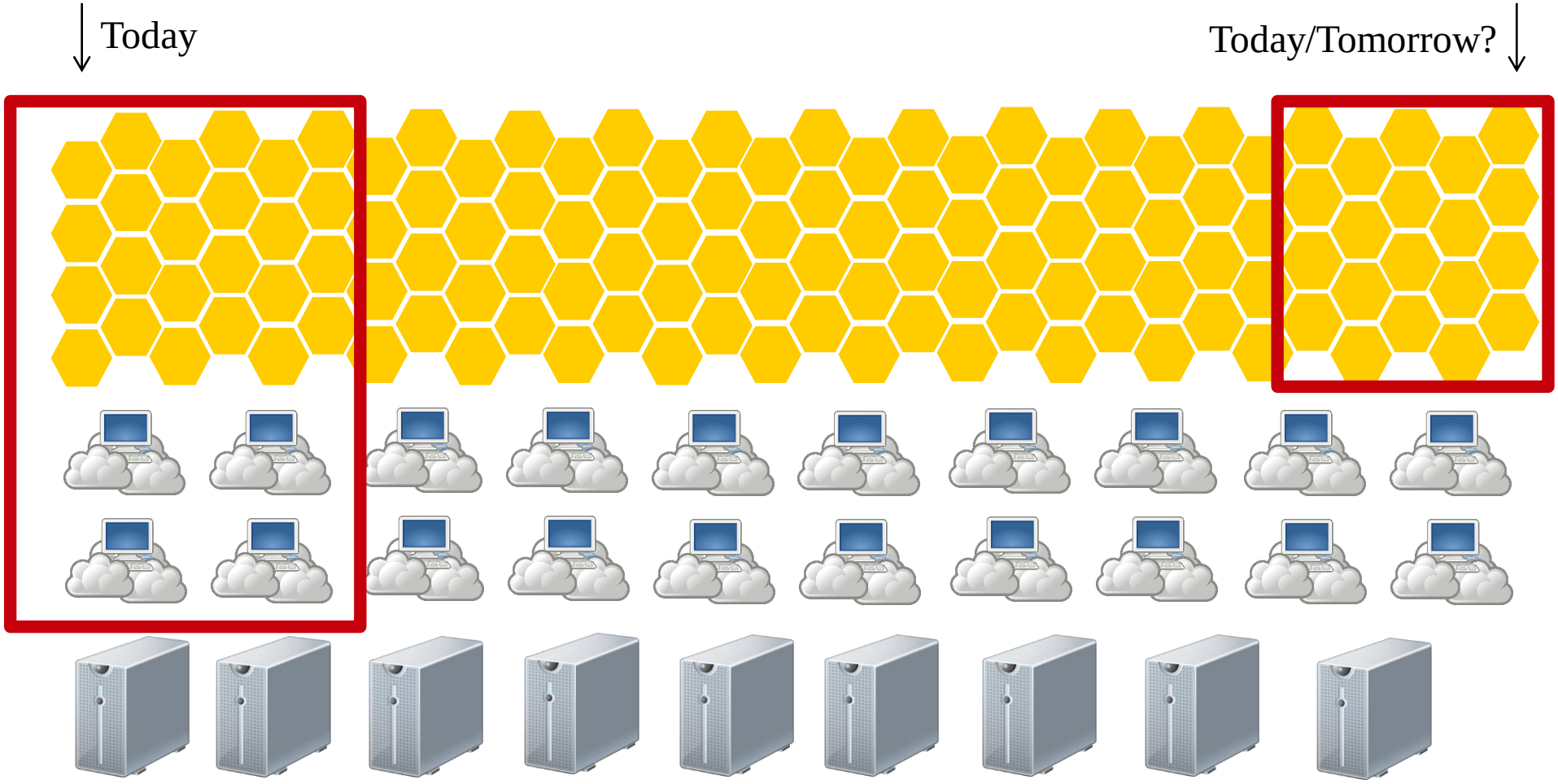
Layers of abstraction



italianaSoftware

imola

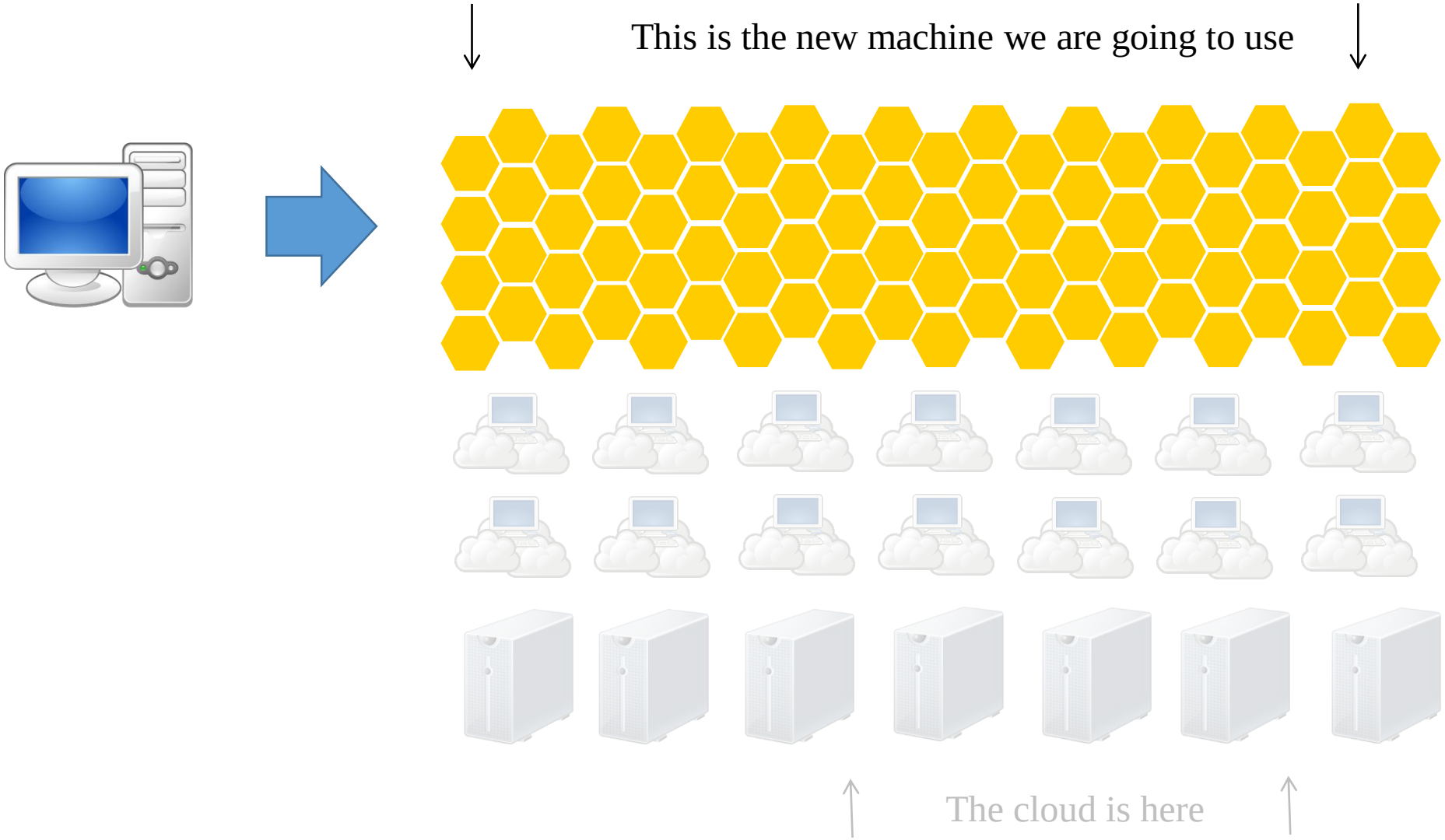
How long are we seeing behind the surface?



italianaSoftware

imola

The new machine

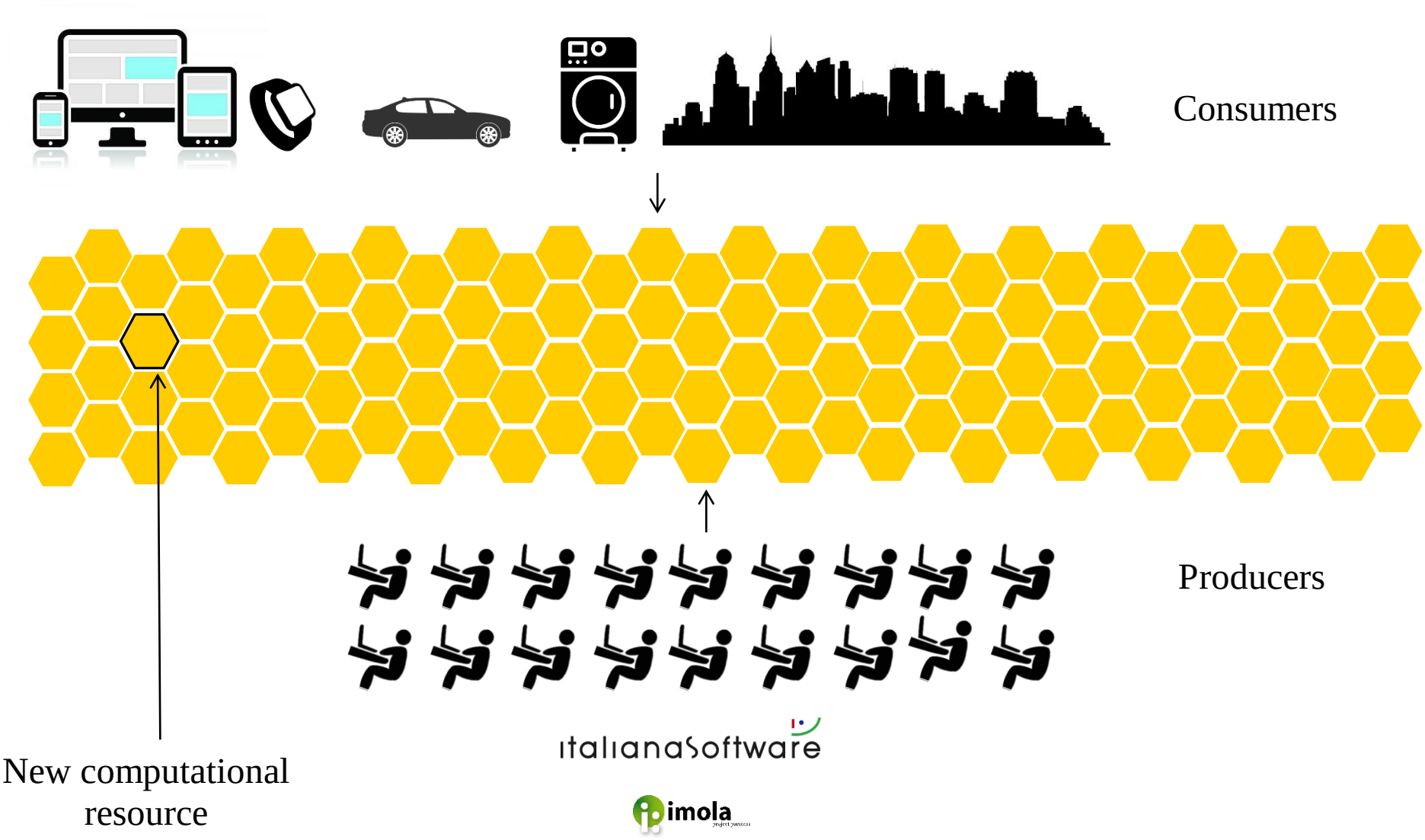


italianaSoftware

imola

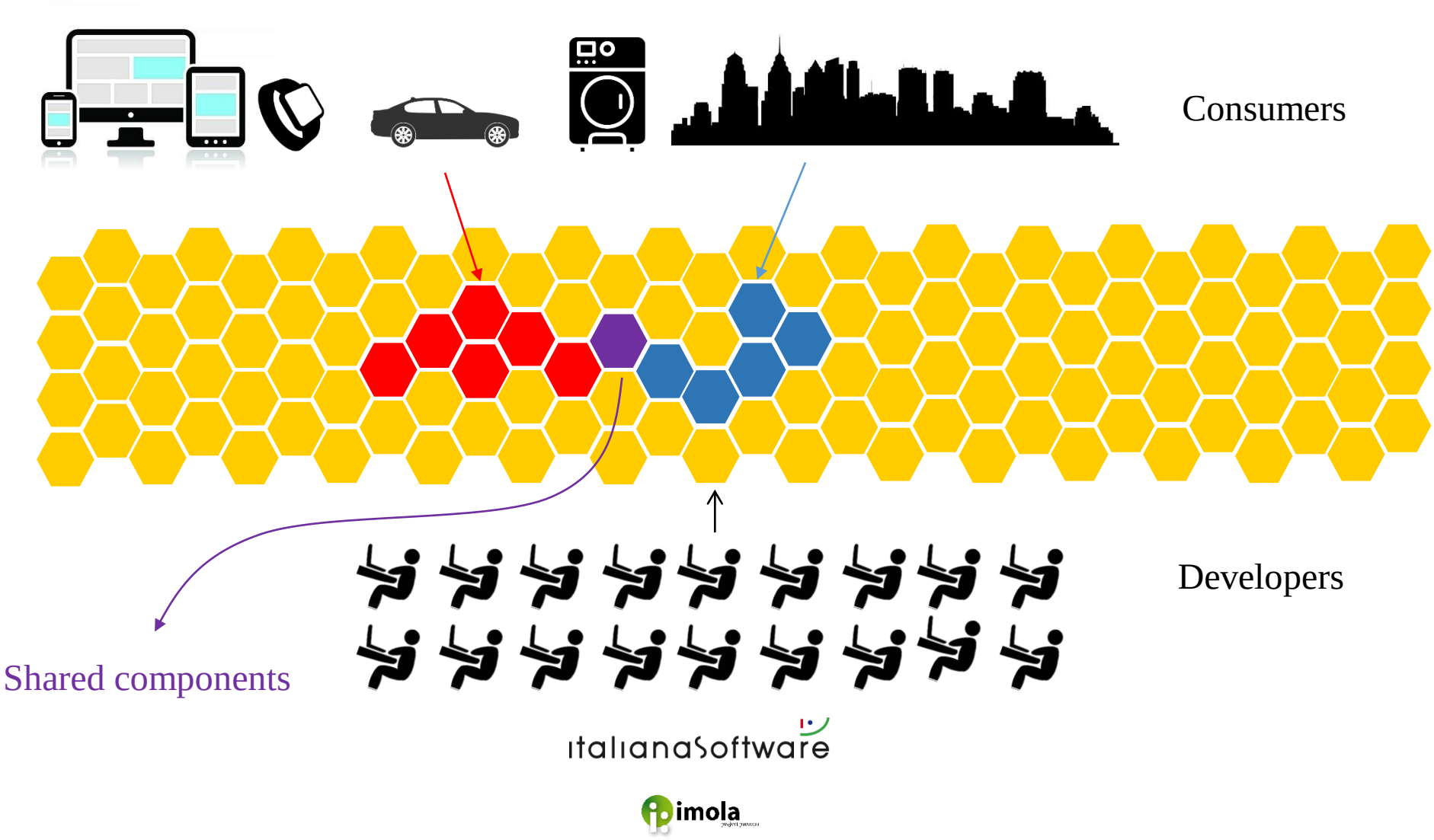
What does it imply?

Which are the short terms and the long terms consequences of such a scenario?



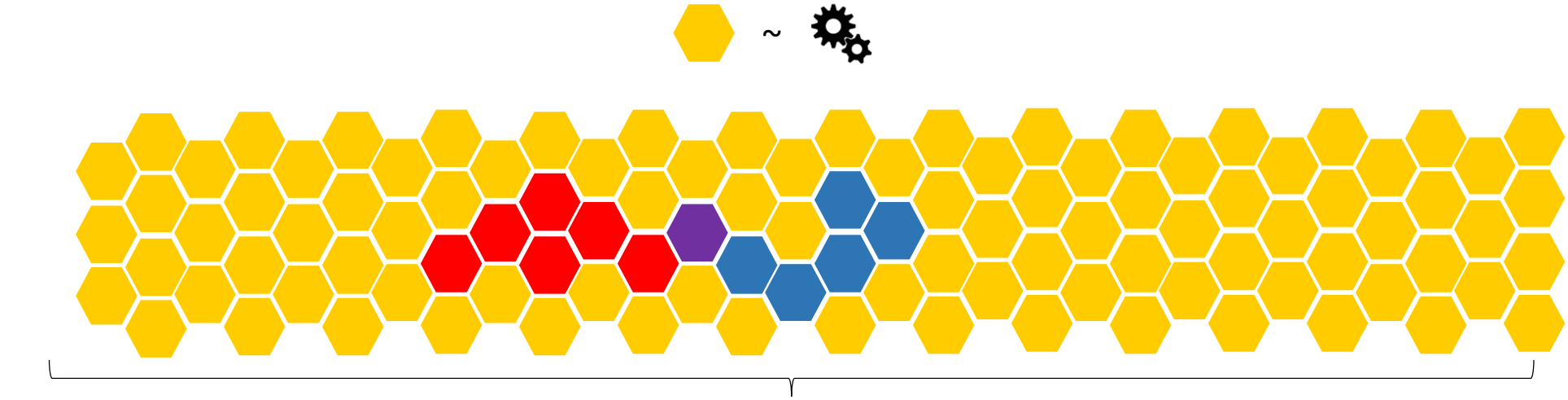
The applications

The applications will be obtained as a composition of existing components



Computational resource
~
Software functionality

Ideally, the concept of computational resource will be more and more interchangeable with the concept of software component.



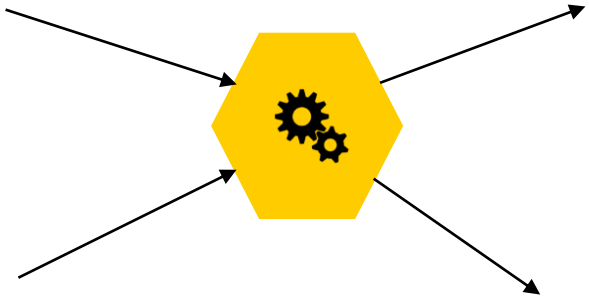
≠

In the classical idea where a computational resource is a computer machine, the software is concretely installed into the machine and it is strongly tight to the technical features of the machine

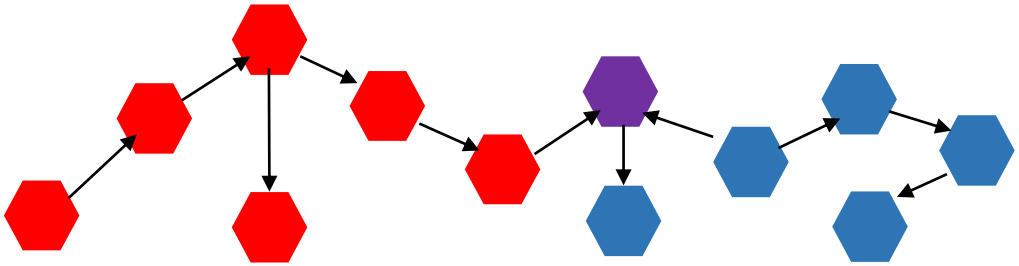
In the new machine the metrics of the system will be managed by the cloud platform provider as a all which will optimize the loads, by relocating resources or replicating them. The software functionlities will be actually decoupled from the available technical resources.

Communication is more important than computation

Communication and coordination will be more important than computation



The management of the connections and the dependencies among the components is the key point!



italianaSoftware

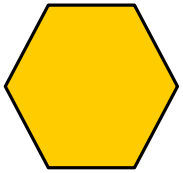
imola

Containers today

italianaSoftware

imola

Core concepts of a container



System Level

- Lighter than Virtual Machines
- Requires a software layer installed in the hosting machine, ex: Docker

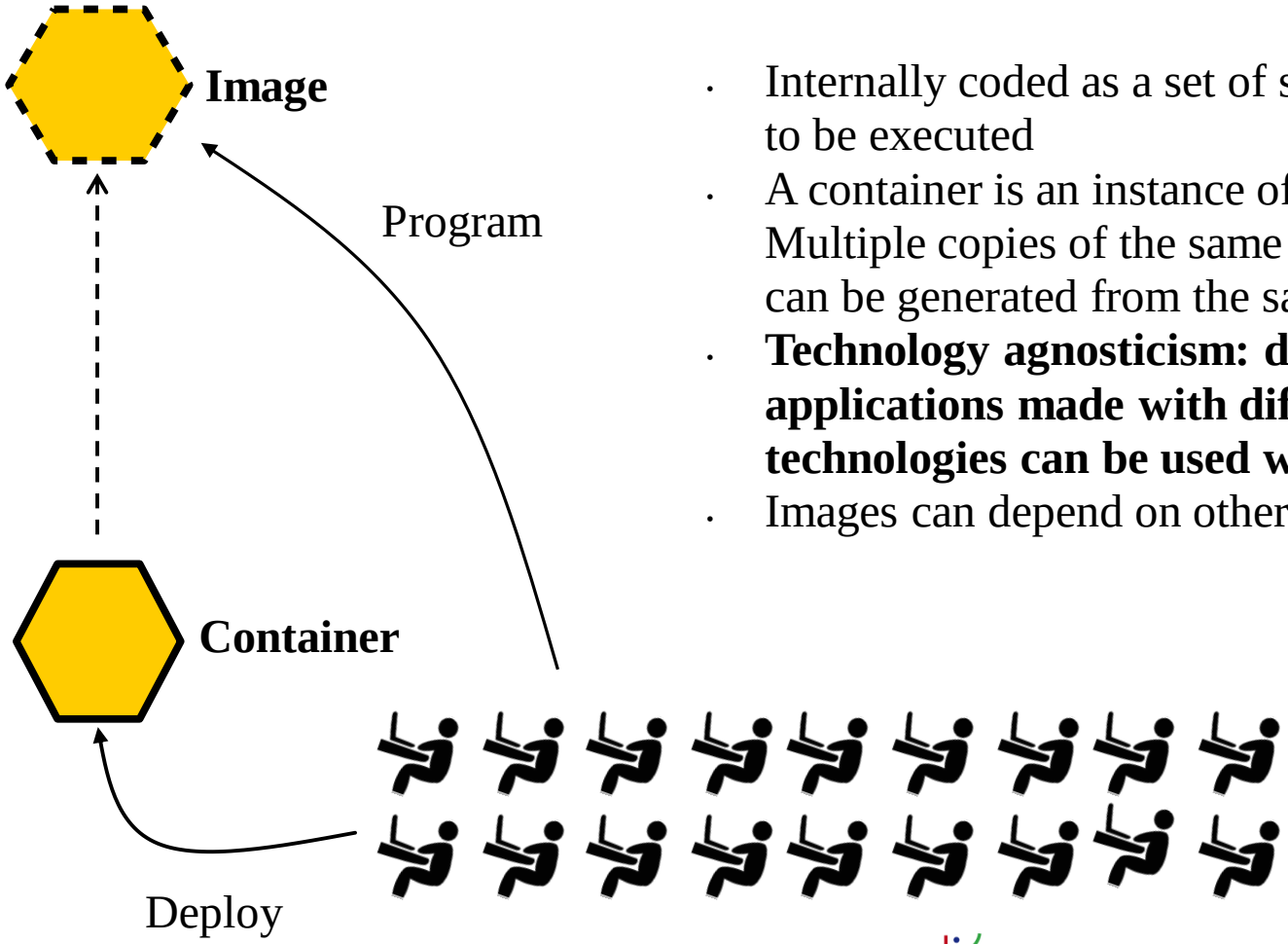
Application Level

- Instantiated starting from an image
- Usually requires connections with other containers
- Remotely controlled status: start/stop/pause/destroy/…

italianaSoftware

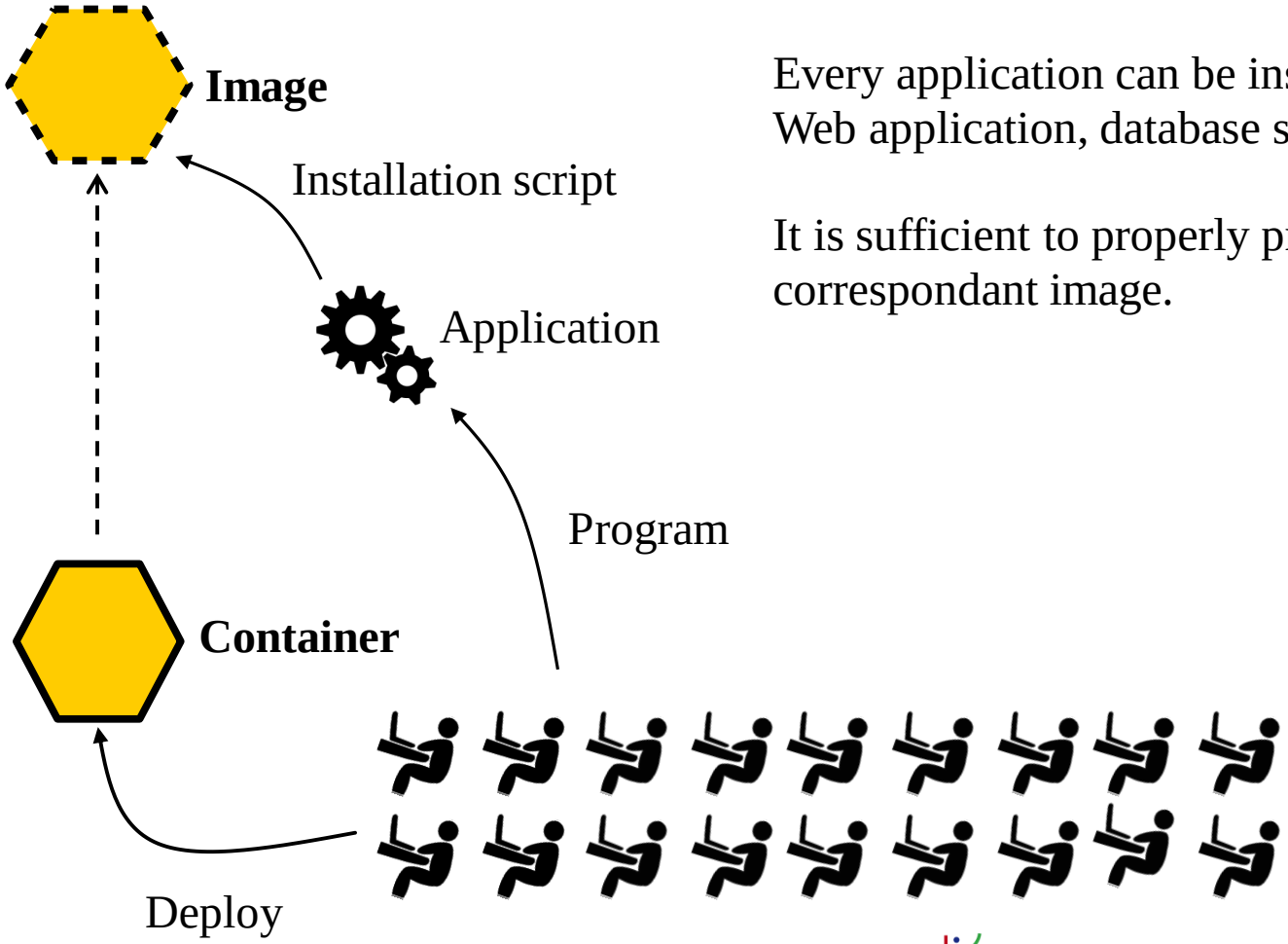
imola

Images



- Internally coded as a set of script to be executed
- A container is an instance of an image. Multiple copies of the same container can be generated from the same image.
- **Technology agnosticism: different applications made with different technologies can be used within a container**
- Images can depend on other images

Inside a container



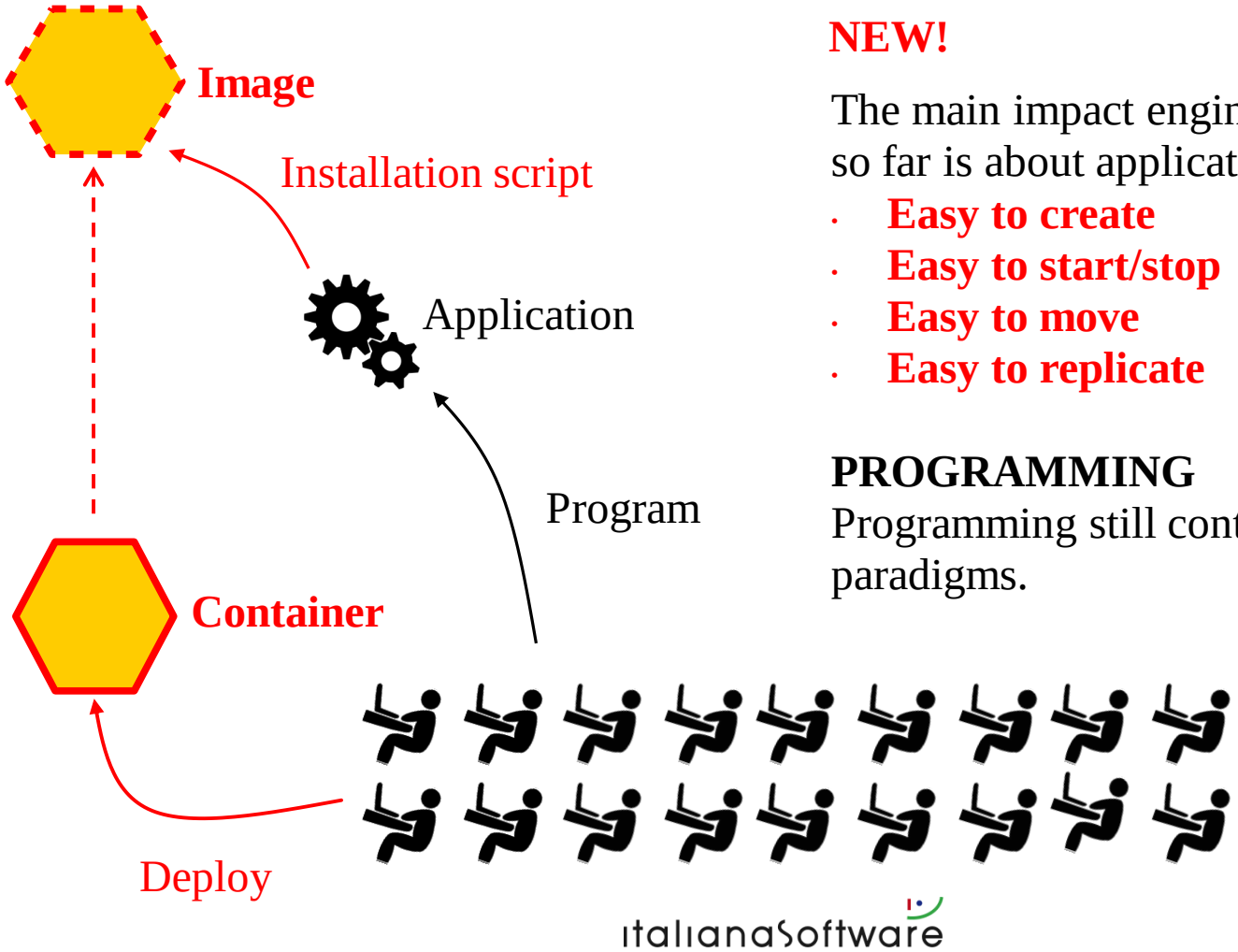
Every application can be installed within a container.
Web application, database servers, load balancers, etc.

It is sufficient to properly prepare the correspondant image.

italianaSoftware

imola

Deploying



NEW!

The main impact engineers are facing so far is about application deployment.

- **Easy to create**
- **Easy to start/stop**
- **Easy to move**
- **Easy to replicate**

PROGRAMMING

Programming still continue to exploit traditional paradigms.

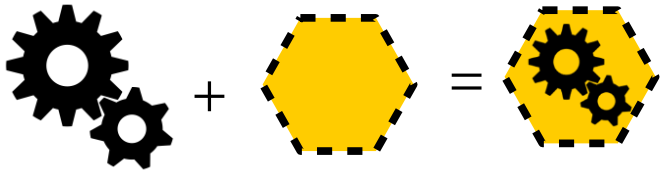
italianaSoftware

imola

What's programming?

Remember: communication and **coordination** are the main features to deal with

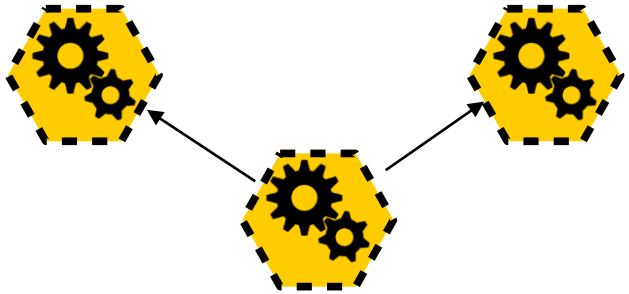
Depending on the technologies there are different languages, different tools, different methods



Two aspects:

1. **Single component** programming
    1. Communication endpoints
    2. Communication protocols
    3. Communication primitives
    4. Activity flow
    5. Fault handling

2. **Architecture** programming.
    1. Component composition
    2. Message routing
    3. Protocol transformation
    4. Asynchronous vs synchronous communication

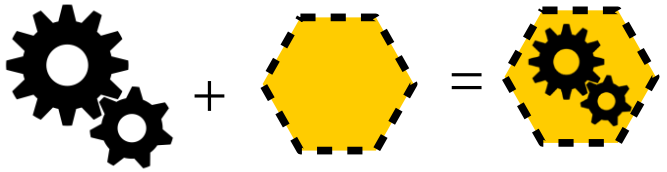


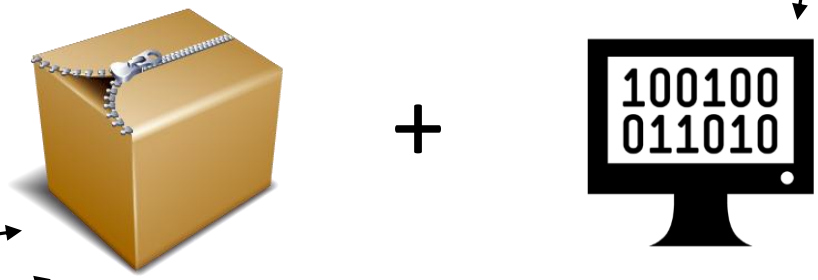
Single component programming

Access endpoints for providing functionalities
- Communication protocols
- Interfaces of the functionalities

Other components dependencies
- Communication protocols
- Interfaces of the functionalities

Business logics
- Actual coding of the provided functionalities



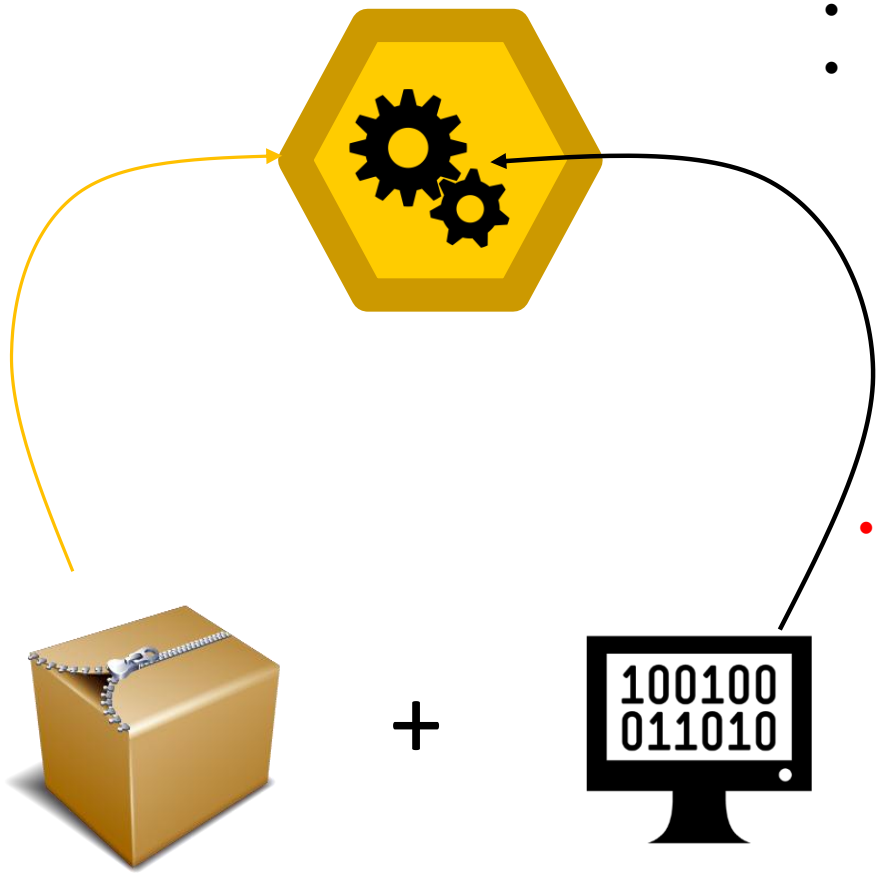


Libraries and Frameworks + Programming Languages

Frameworks and languages

Communication and coordinationfeatures are usually demanded to the usage of frameworks and libraries

- Several tools and frameworks
- They change very often.
- They require upgrades which could have impacts on the code.
- Specific skills required by the developers
- No actual standards for defining interfaces
- Usually structurally designed for specific protocols (ex: http)



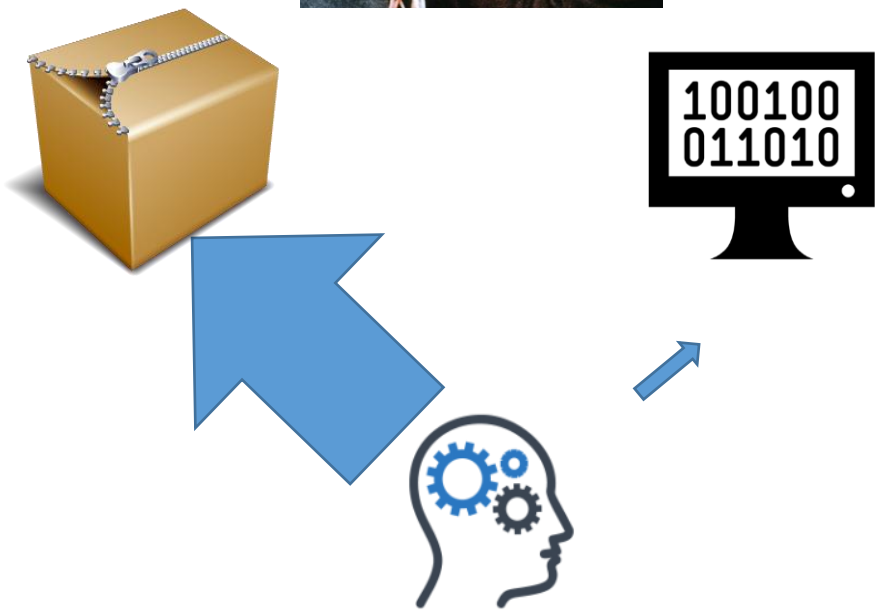
Libraries and Frameworks + Programming Languages

- The syntax is quite stable
- Once trained, a developer has all the important knowledge to be productive

- **Unfortunately**, the code related to the communication activities **is strictly bound** to the chosen framework and libraries.

Programming languages are irrelevant

The knowledge required to manage frameworks and libraries **eats up** the knowledge required to manage the programming language **which become quite irrelevant**



Knowledge is money

- More than one
- They change often
- High dinamicity

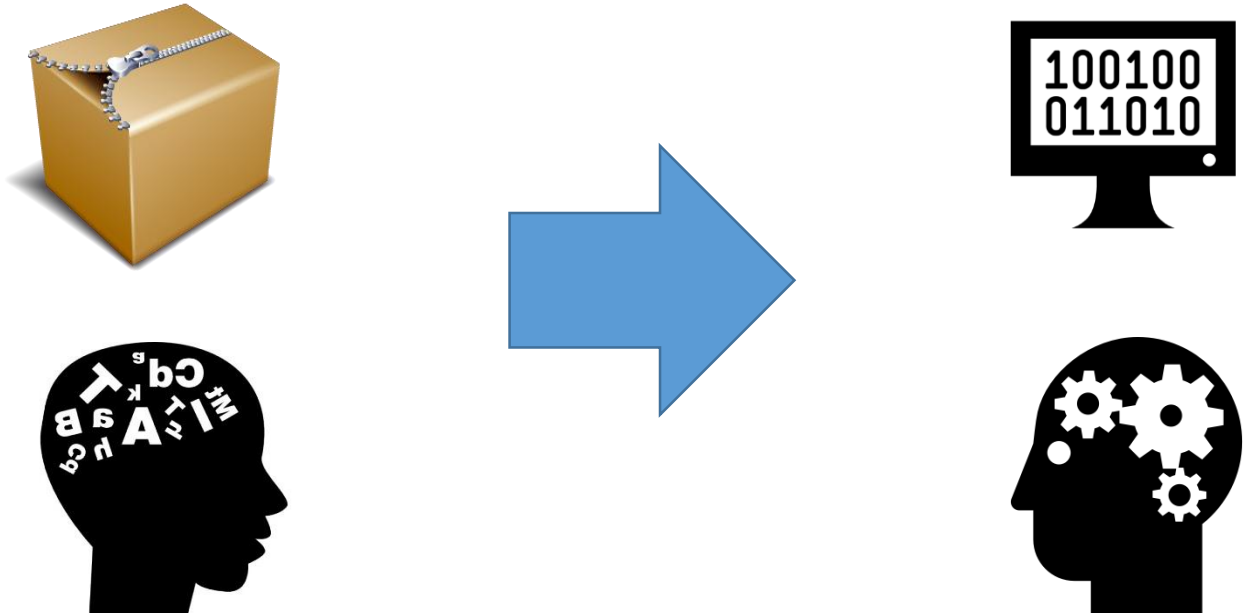
- Syntax usually stable in the long terms. Backward compatibility usually guaranteed

italianaSoftware

imola

How to reduce the knowledge?

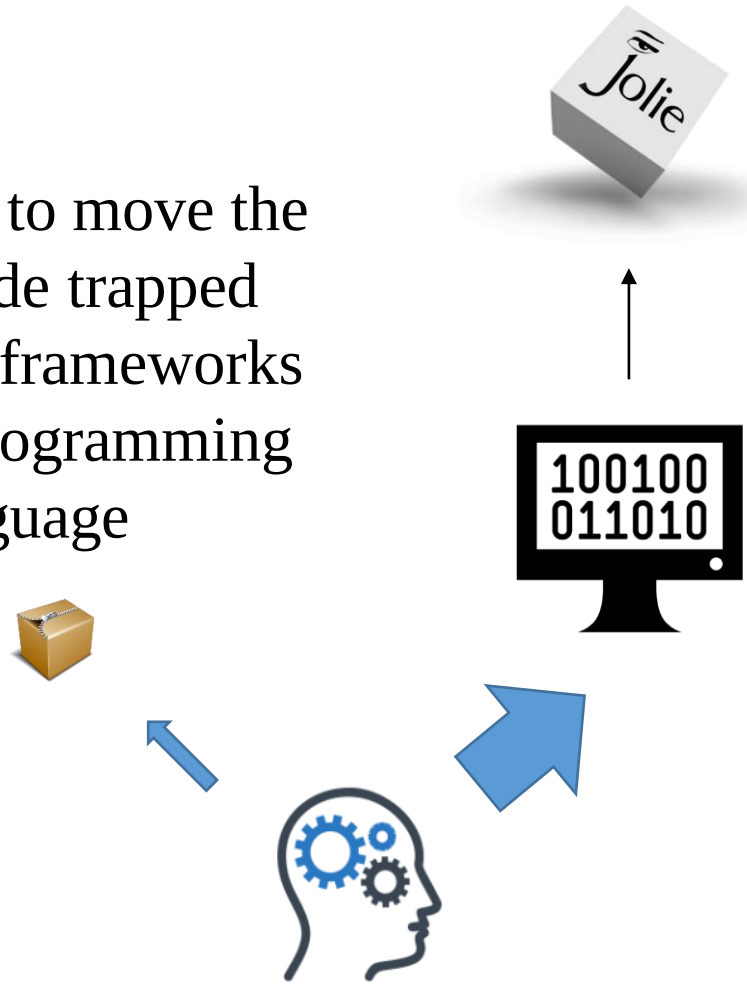
Reducing the required knowleldge by moving the programming concepts hidden in the frameworks into a stable syntax of a programming language



A new programming language

Our choice is to introduce a new programming language.
Jolie represents our solution.

Our aim is to move the knowlegde trapped within the frameworks into the programming language



Jolie

http://www.jolie-lang.org

italianaSoftware

imola

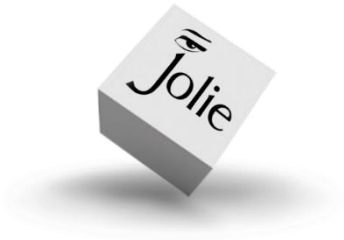
Once upon a time...

SOA standard specifications analysis (WSDL, SOAP, WS-BPEL,…)

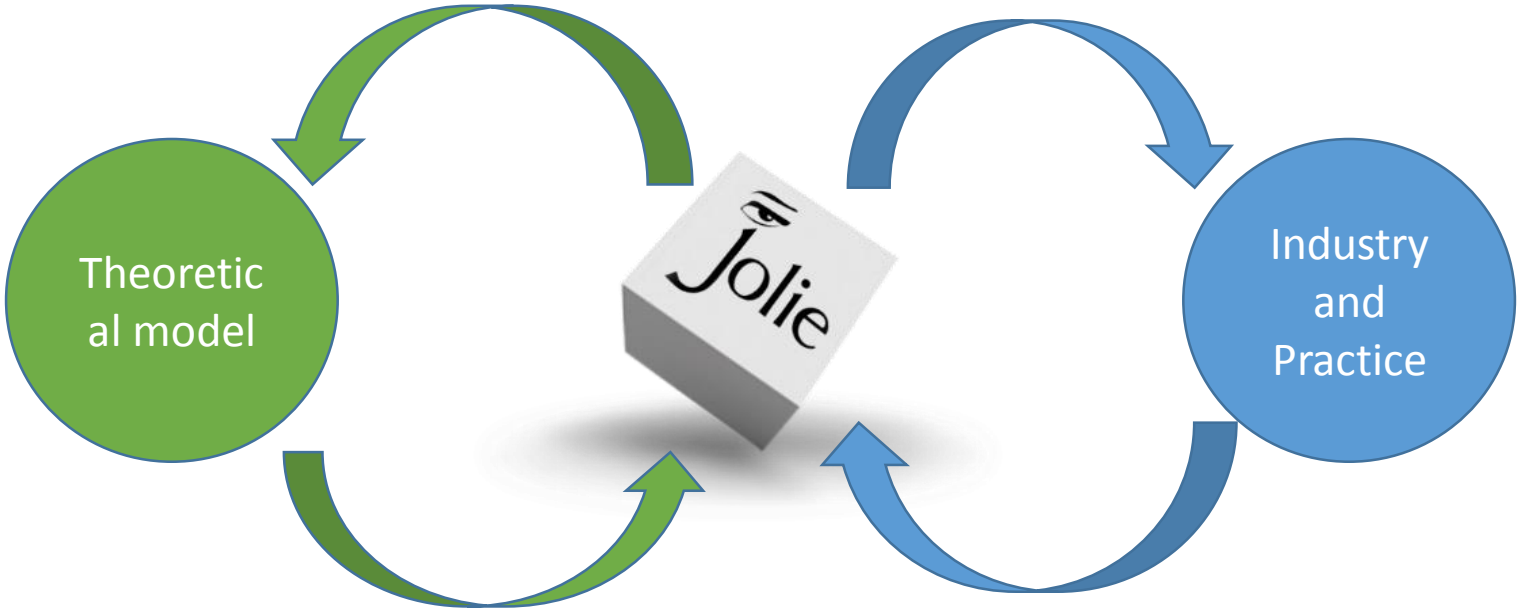
Development of a Milner's CCS based process calculus (SOCK).
Definition of operational semantics

Development of a language based (Jolie) and of its interpreter engine

Exploitation in industry and real life



Action model



italianaSoftware

imola

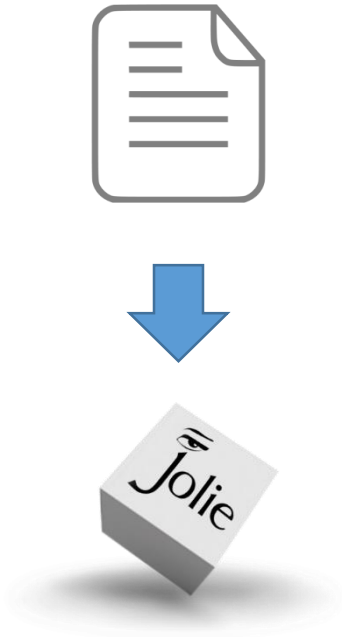
Jolie is interpreted

A jolie service is described by a script
which is interpreted at runtime by the jolie engine.

No compilation required.

The engine is developed in Java

We call **microservice** the running instance of a service



Services

The main idea behind Jolie is that

A service is a single unit of programmable software.

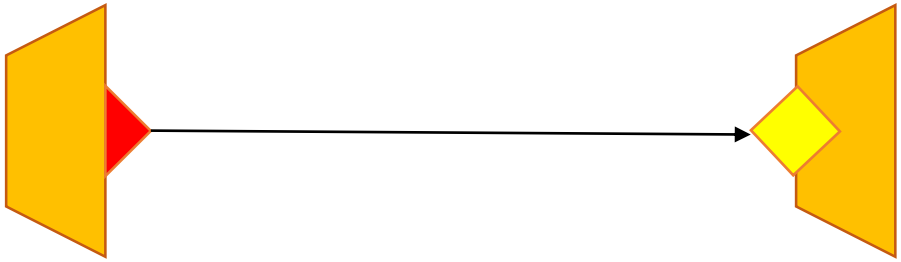
It is not obtained as a customization or a specialization of an existing server, but it is the only possible programmable entity.

We usually exploit a trapezoid for representing a service.

The linguistic constructs for communication

outputPort OPName {
Location: ‘’socket://230.230.230.230:8000’’
Protocol: sodep
Interfaces: MyInterface
}

inputPort IPName {
Location: ‘’socket://230.230.230.230:8000’’
Protocol: sodep
Interfaces: MyInterface
}

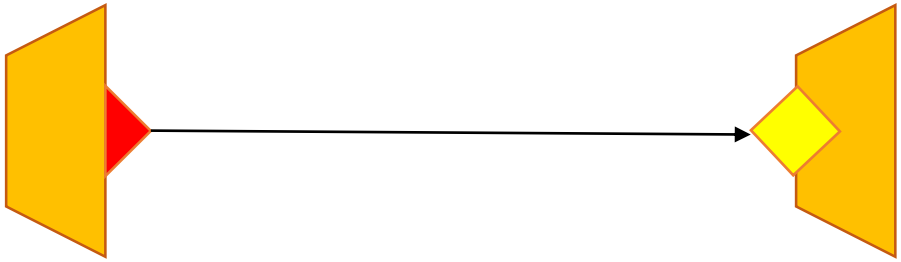


Listener endpoints and sender endpoints are a specific construct in the language. They are called ports: **inputPorts** and **outputPorts**.

Where

outputPort OPName {
Location: ‘’socket://230.230.230.230:8000’’
Protocol: sodep
Interfaces: MyInterface
}

inputPort IPName {
Location: ‘’socket://230.230.230.230:8000’’
Protocol: sodep
Interfaces: MyInterface
}

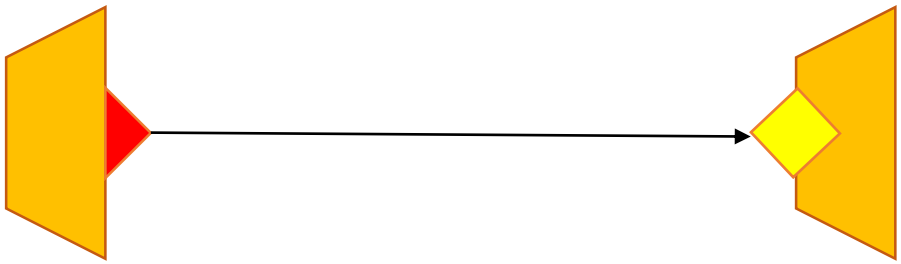


The location represents the place, in the space of the Internet, where the service can be reached.

How

```
outputPort OPName {
Location: ‘‘socket://230.230.230.230:8000’’
Protocol: sodep
Interfaces: MyInterface
}
```

```
inputPort IPName {
Location: ‘‘socket://230.230.230.230:8000’’
Protocol: sodep
Interfaces: MyInterface
}
```

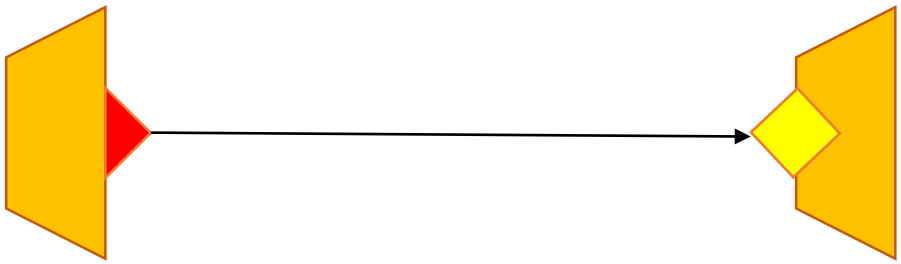


The protocol specifies the application protocol to be used. Sodep is a protocol released with Jolie but also http, http/json, http/soap and https.

What

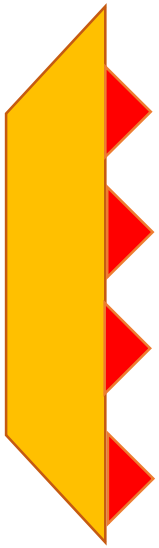
outputPort OPName {
Location: ‘’socket://230.230.230.230:8000’’
Protocol: sodep
Interfaces: MyInterface
}

inputPort IPName {
Location: ‘’socket://230.230.230.230:8000’’
Protocol: sodep
Interfaces: MyInterface
}

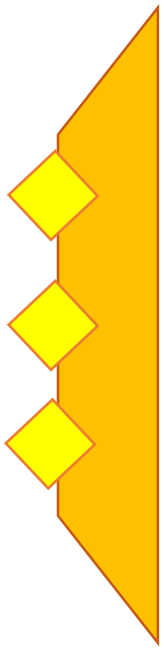


The interface expresses all the available operations provided by a service.

More than one port are possible



A service can invoke different services on different ports.

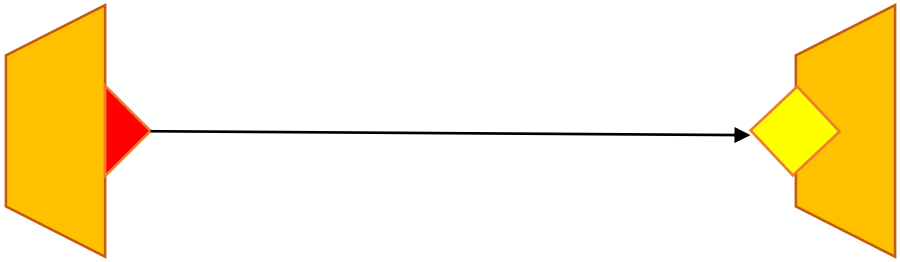


A service can provide its functionalities on more than one port. Different protocols and locations are possible for a service.

The interface definition is part of the language

```
outputPort OPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}
```

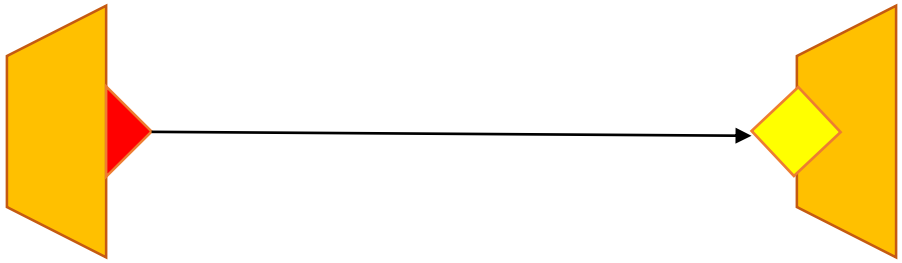
```
inputPort IPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}
```



```
type TestRRequest: void {
        .field: string
}

Interface MyInterface {
RequestResponse:
                testRR( TestRequest )( string )
OneWay:
                testOW( TestRequest )
}
```

Communication primitives

OneWay: asynchronous

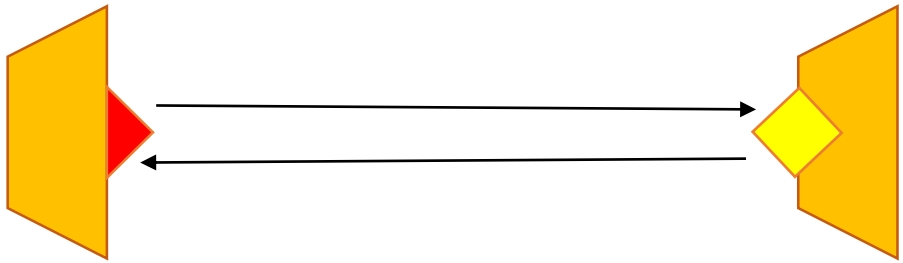


```
outputPort OPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}

main {
    request.field = ''hello world!''
    testOW@OPName( request );
    …other activities…
}
```

```
inputPort IPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}

main {
    testOW( request );
    …other activities…
}
```

Communication primitives

RequestResponse: synchronous



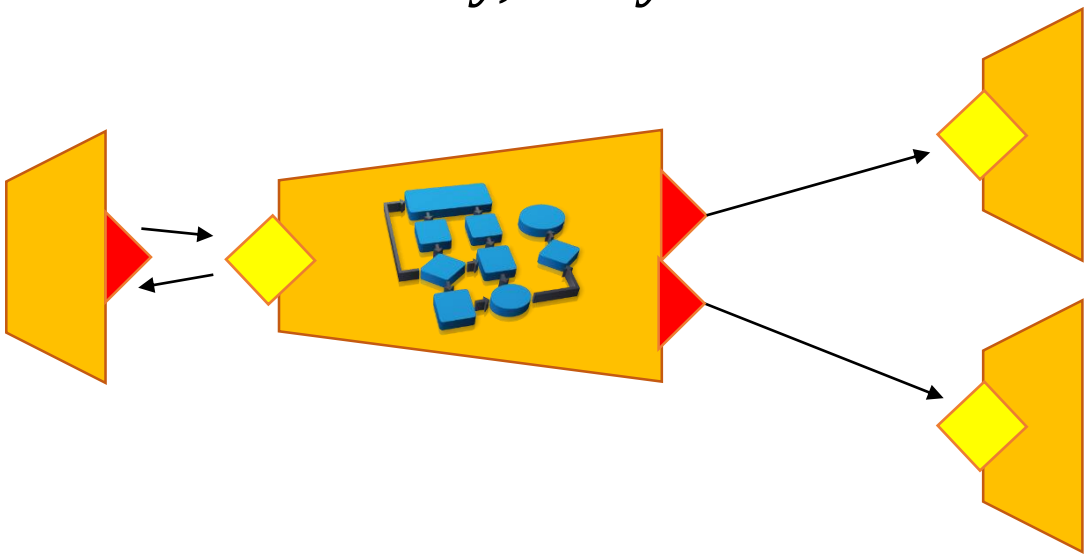
```
outputPort OPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}

main {
    request.field = ''hello world!''
    testRR@OPName( request )( response );
    …other activities…
}
```

```
inputPort IPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}

main {
    testRR( request )( response ) {
    …other activities…
    }
}
```

The behaviour

The behaviour of a service is represented by a **workflow**.

Potentially, every service is an orchestrator.



inputPort OPName2 {
Location: "socket://230.230.230.230:8001"
Protocol: sodep
Interfaces: MyInterface2
}

inputPort OPName3 {
Location: "socket://230.230.230.230:8002"
Protocol: sodep
Interfaces: MyInterface3
}

Parallel, sequence, external choice

```
main {
    testRR( request )( response ) {
            ....
            test2RR@OpName2( ... )( ... )
            |
             test3RR@OpName3( ... )( ... )
    }
}
```

A behaviour structured as a workflow permits to focus on the message flows instead of computation

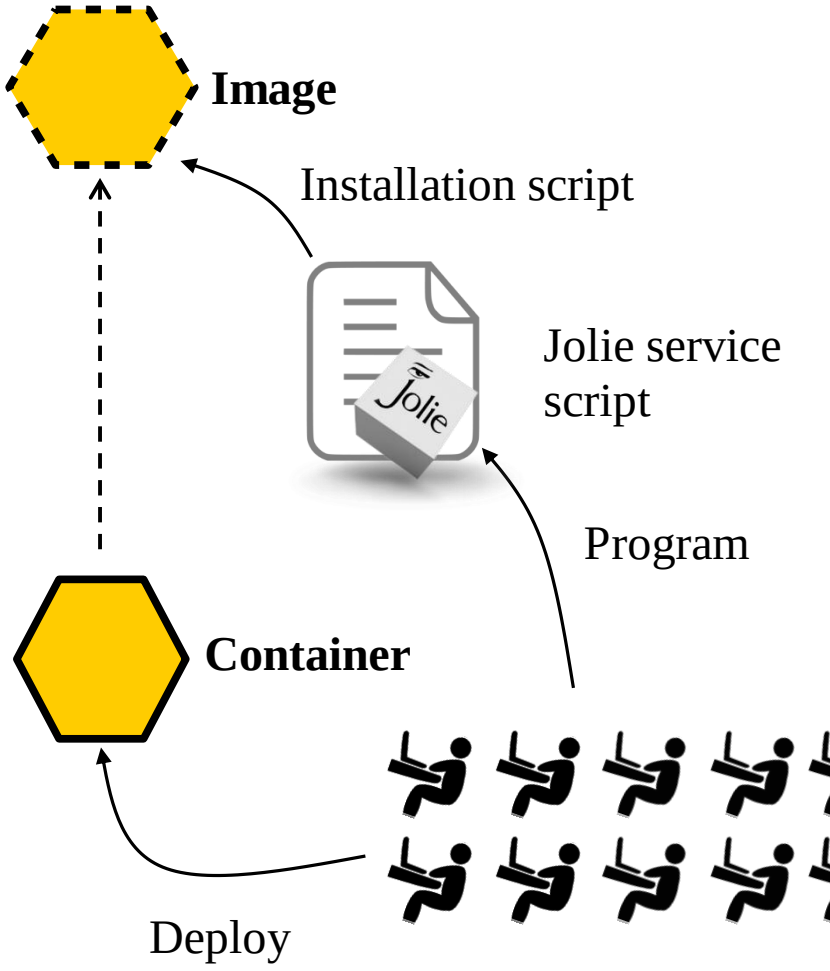
Other

- **Tree format for data and messages**
- **Standard computation constructs**
  - if then
  - for
  - while
- **Other constructs**
  - spawn
  - foreach
  - provide until
  - linkIn and linkOut
- **Fault handling**
  - Termination handlers
  - Compensation handlers
  - Dynamic installation handlers
- **Sessions**
  - Session correlation (correlation sets)
  - Global data and session data
  - Critical sections programming (synchronize)
- **Execution modalities**
  - Single
  - Sequential
  - Concurrent

Important distilled single component programming concepts

Programming Constructs	Benefits
Ports	They permit to define in a standard way endpoints: protocols and available operations are just parameters.
Interface	It permits to formally define the interface of the service without exploiting external tools.
Communication primitives	They permit to define message communication actions (send and receive) independently from protocols and formats.
Workflow behaviour	It permits to define the activities of a service by focusing on the message flow instead of computation. Sessions are instantiated automatically.
…	

If you remember...



PROGRAMMING WITH JOLIE
No specific packaging is required or libraries.
Just prepare the container with the Jolie Engine.

The code is readable.

The syntax is stable.

Less knowledge required.

Common concepts and constructs immediately shared by all the team members.
Everybody can read and understand what a service is doing wothout any additional knowledge.

italianaSoftware

imola

Programming architectures

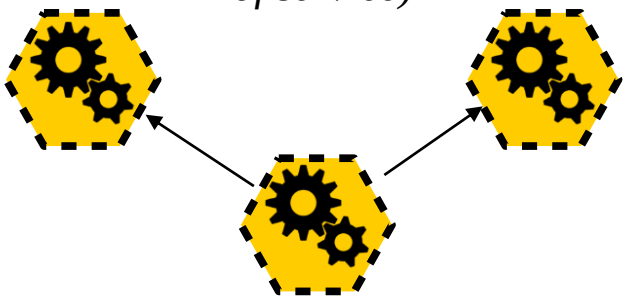
italianaSoftware

imola

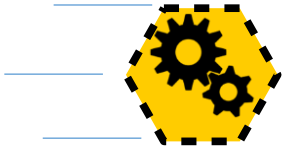
Where is the architecture?

Containers

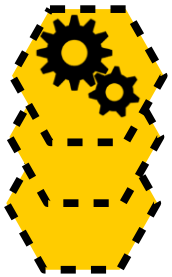
Creating connections
(they must still available at the level of service)



Moving



Scaling



Services

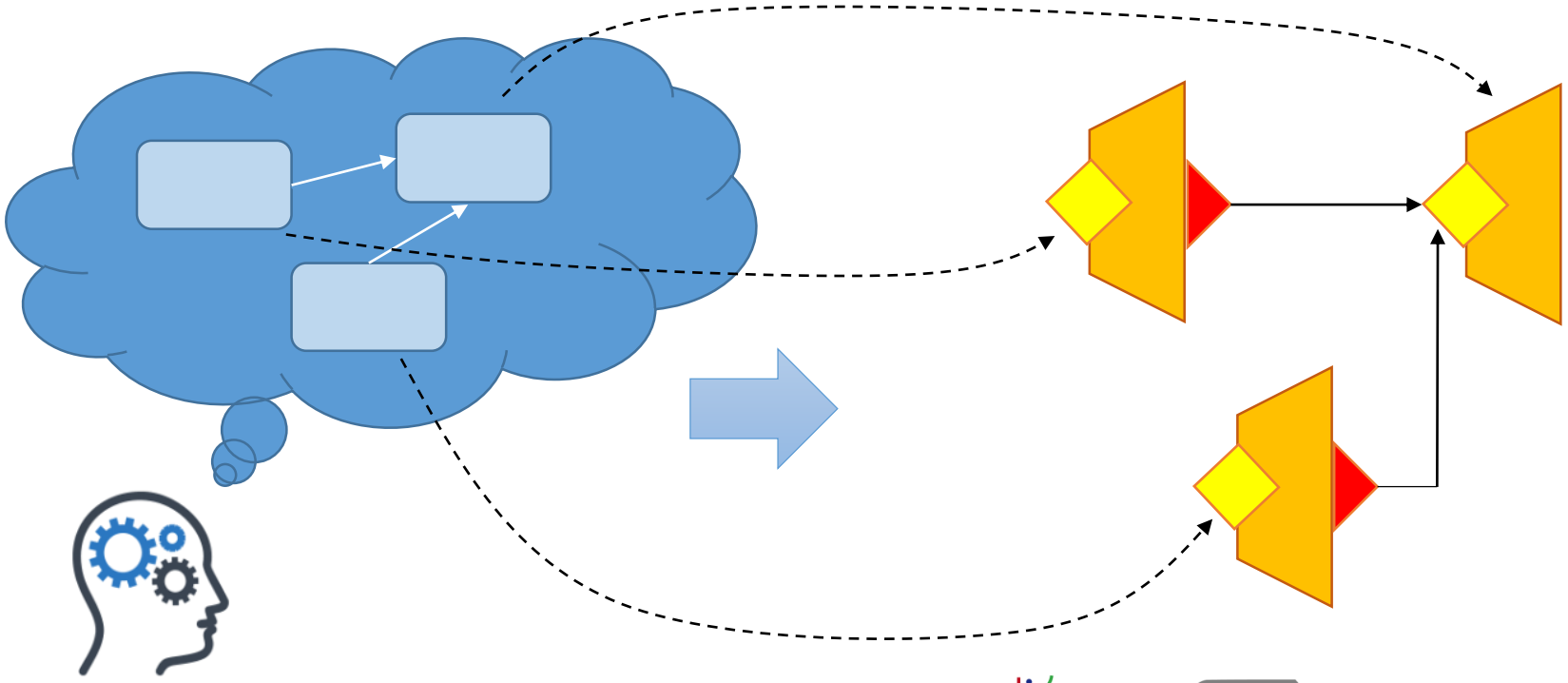
?

italianaSoftware

imola

Programming services means programming architectures

Since the single programmable unit is a service, it does not matter the conceptual designing model you choose for designing your system or your application because ***the result will always be an architecture of services!***



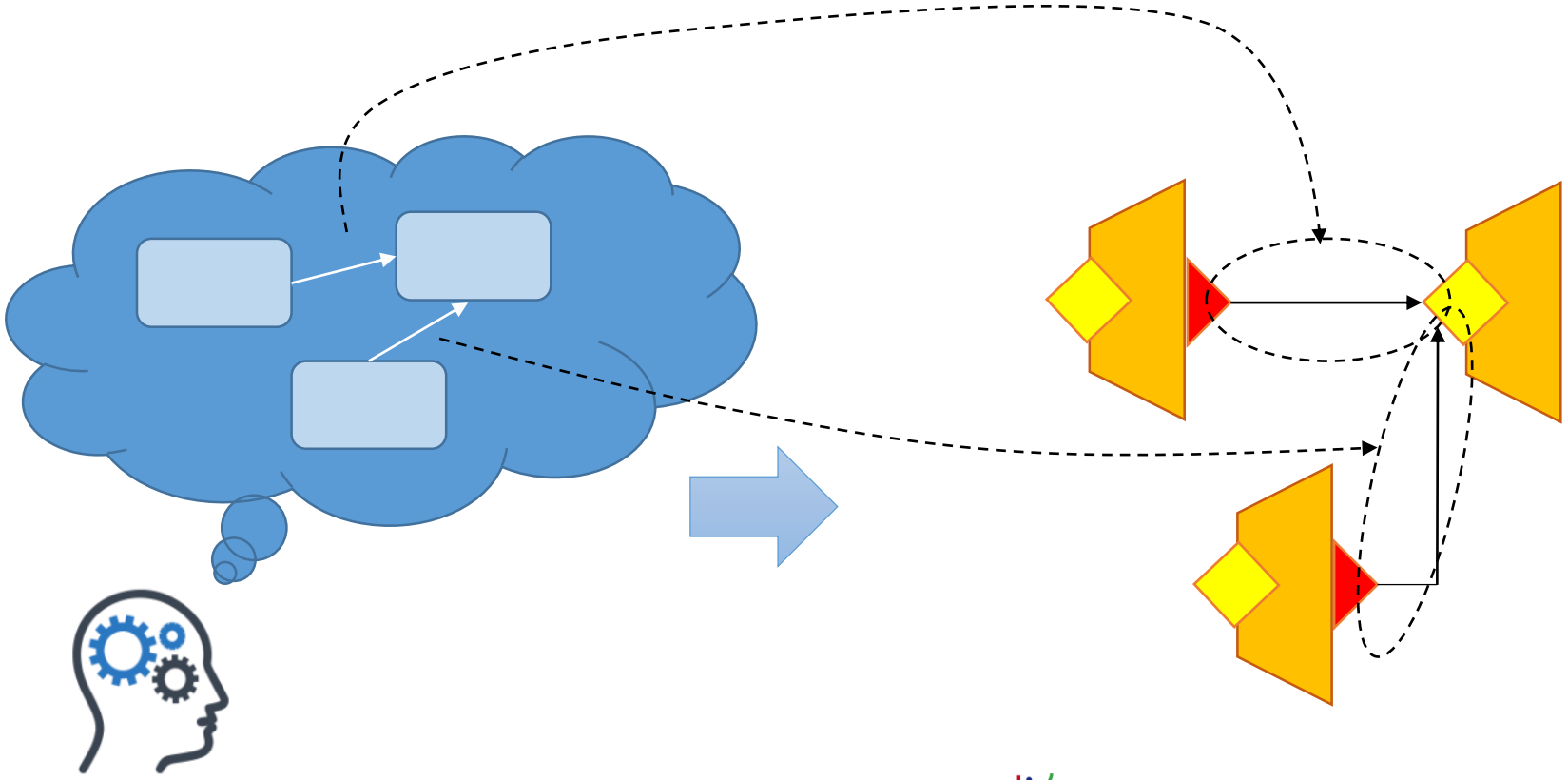
italianaSoftware

imola

Each modelled service has an immediate and direct representation into code

Prommaning services means programming architectures

The relation among entities are directly modelled as service dependencies

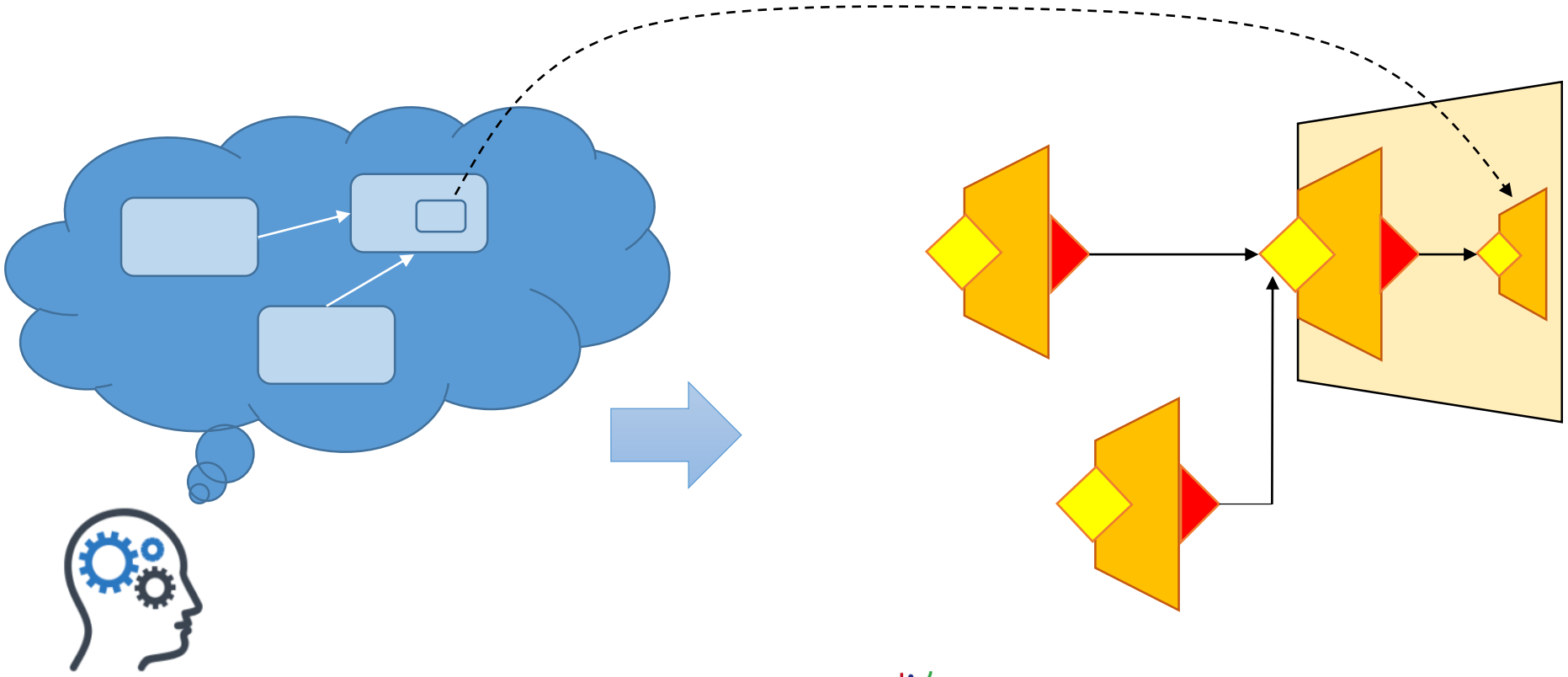


italianaSoftware

imola

Service size is not a problem

You can easily fragment your entities in sub-entities
by keeping the mapping with services because services can be ***EMBEDDED***

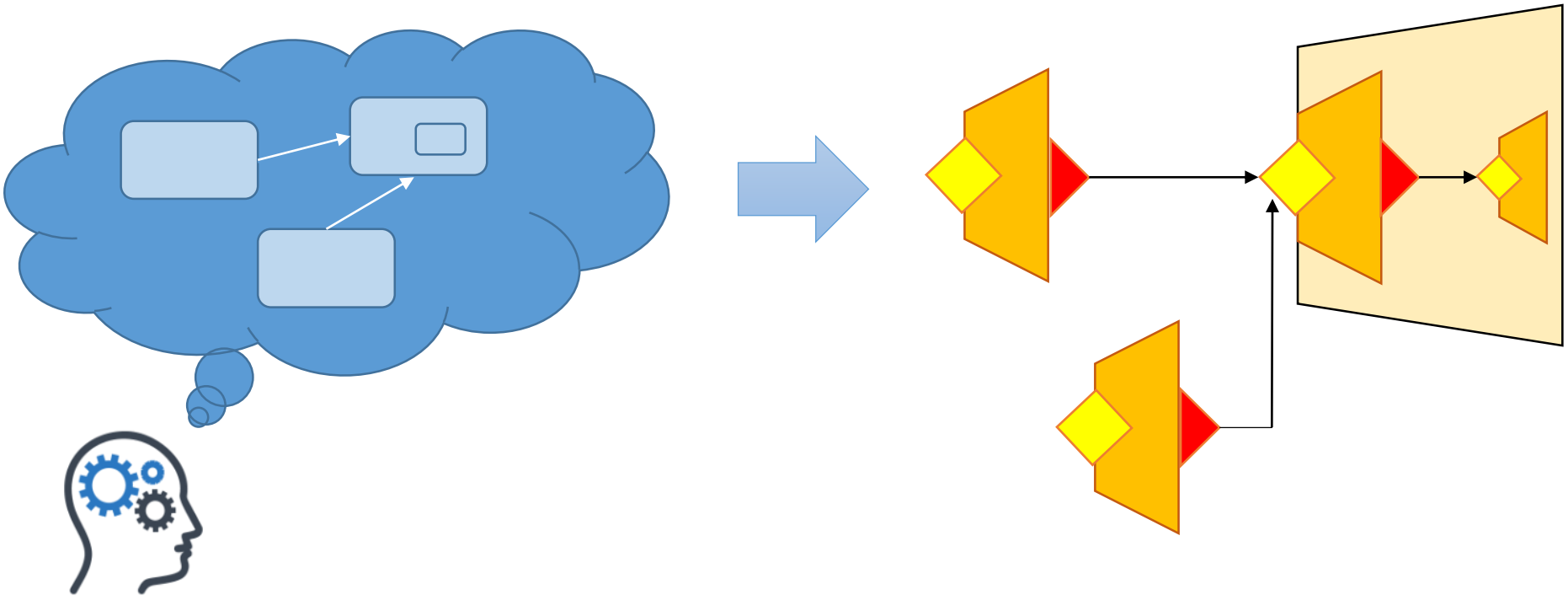


italianaSoftware

imola

Everything you think is an architecture!

Do you remember? We said we want to reduce the knowledge!



italianaSoftware

imola

Jolie

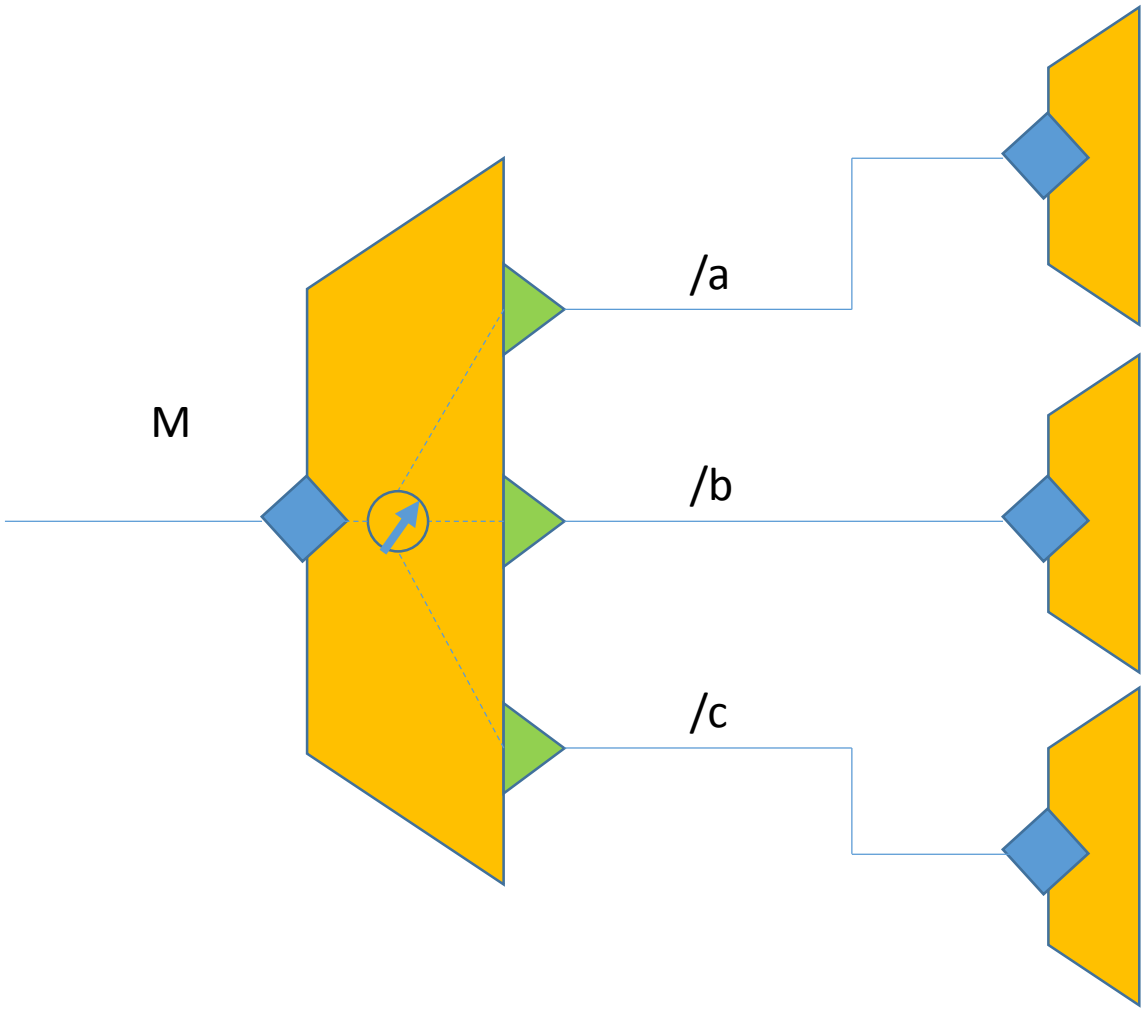
embedded {
...
}

Programming architectures with Jolie

italianaSoftware

imola

Redirection



italianaSoftware

imola

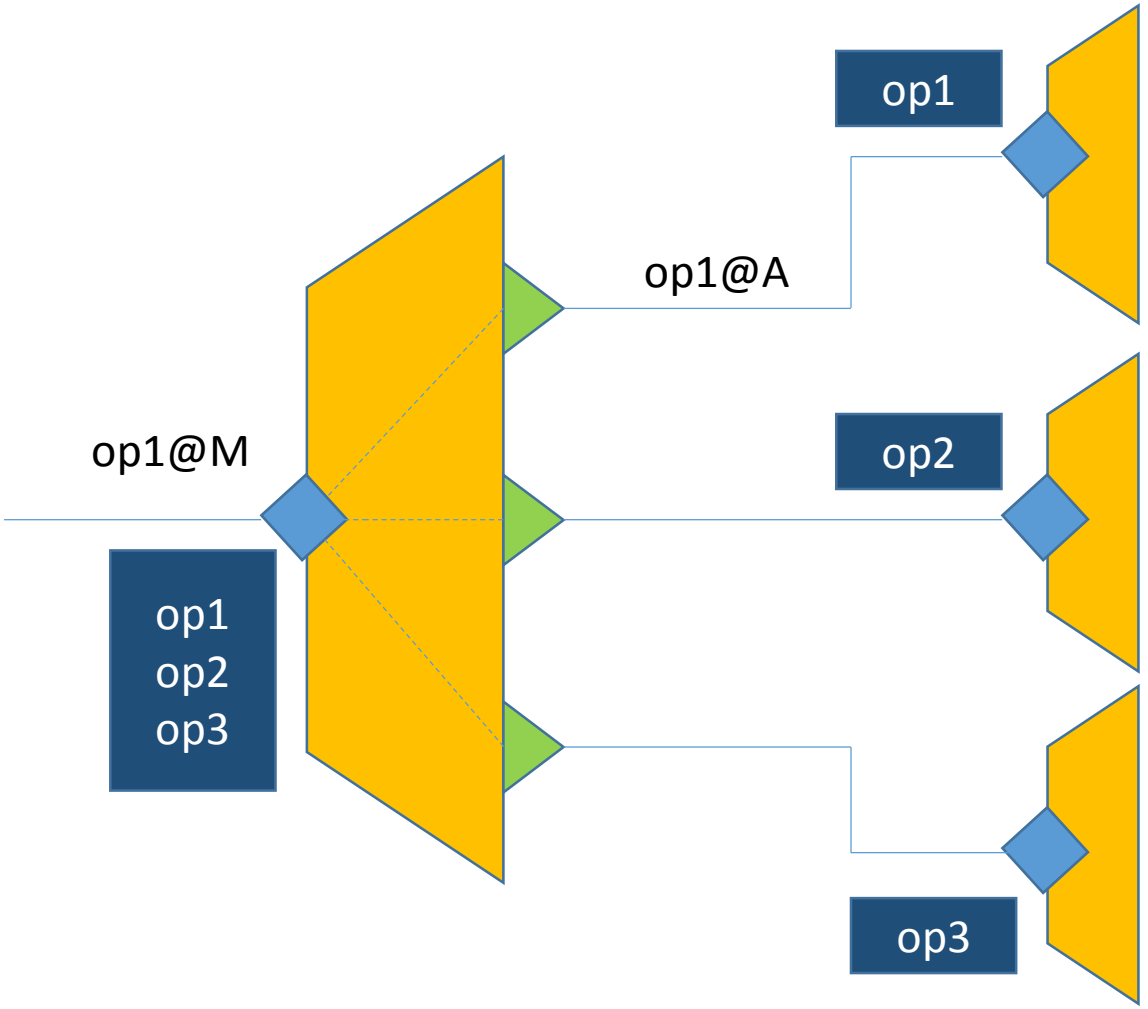
Redirection, the code

```
outputPort SubService {
    Location: "socket://localhost:8001/"
    Protocol: soap
}

outputPort SumService {
    Location: "socket://localhost:8002/"
    Protocol: soap
}

inputPort M {
    Location: "socket://localhost:8000/"
    Protocol: sodep
    Redirects: a => A,
               b => B
}
```

italianaSoftware

imola

Aggregation



italianaSoftware

imola

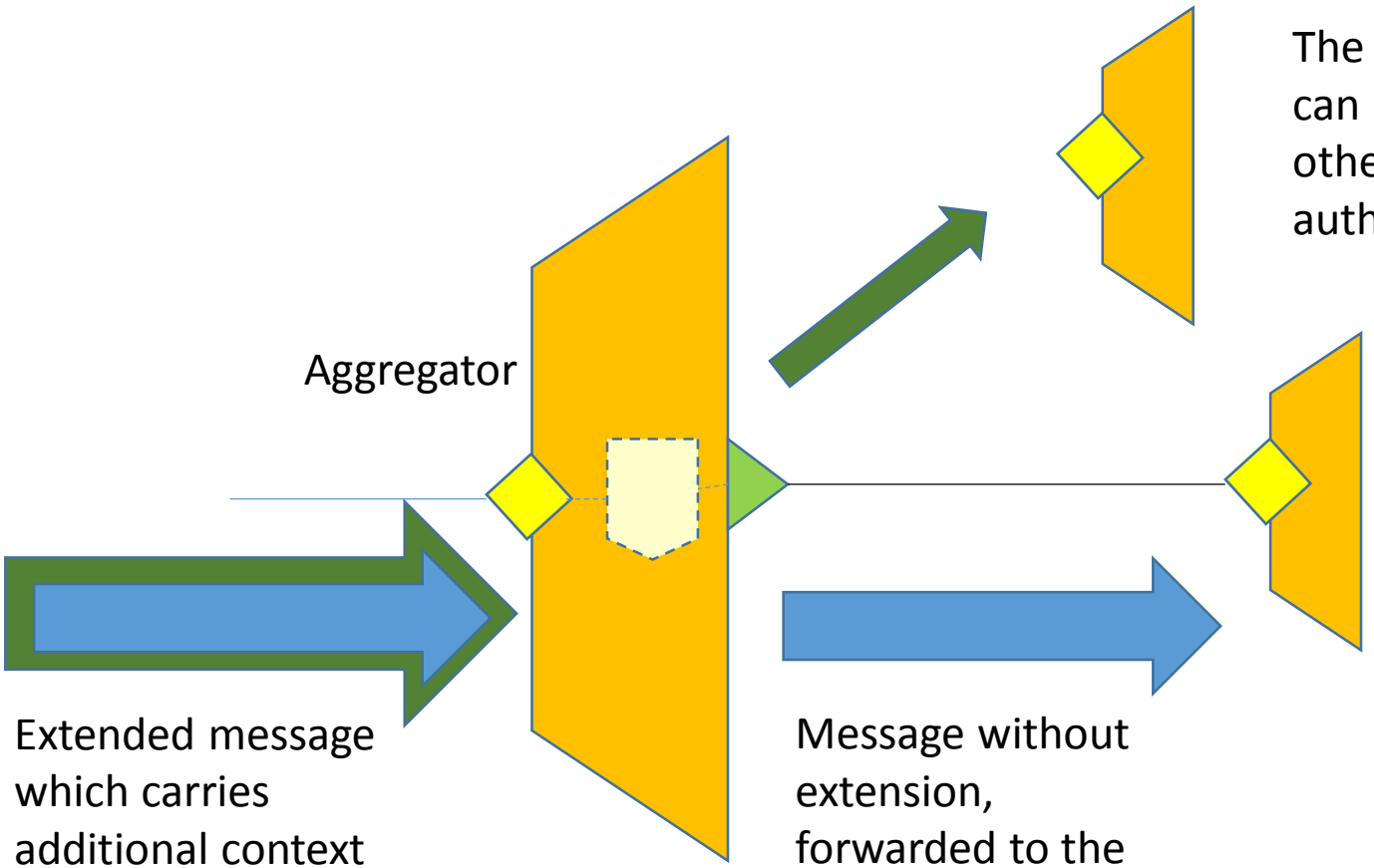
Aggregation, the code

```
outputPort A {
    Location: "socket://localhost:8001/"
    Protocol: soap
}

outputPort B {
    Location: "socket://localhost:8002/"
    Protocol: soap
}

inputPort M {
    Location: "socket://localhost:8000/"
    Protocol: sodep
    Aggregates: A, B
}
```

italianaSoftware

imola

Couriers



The extended information can be used for invoking other services (ex: authentication services)

Aggregated service

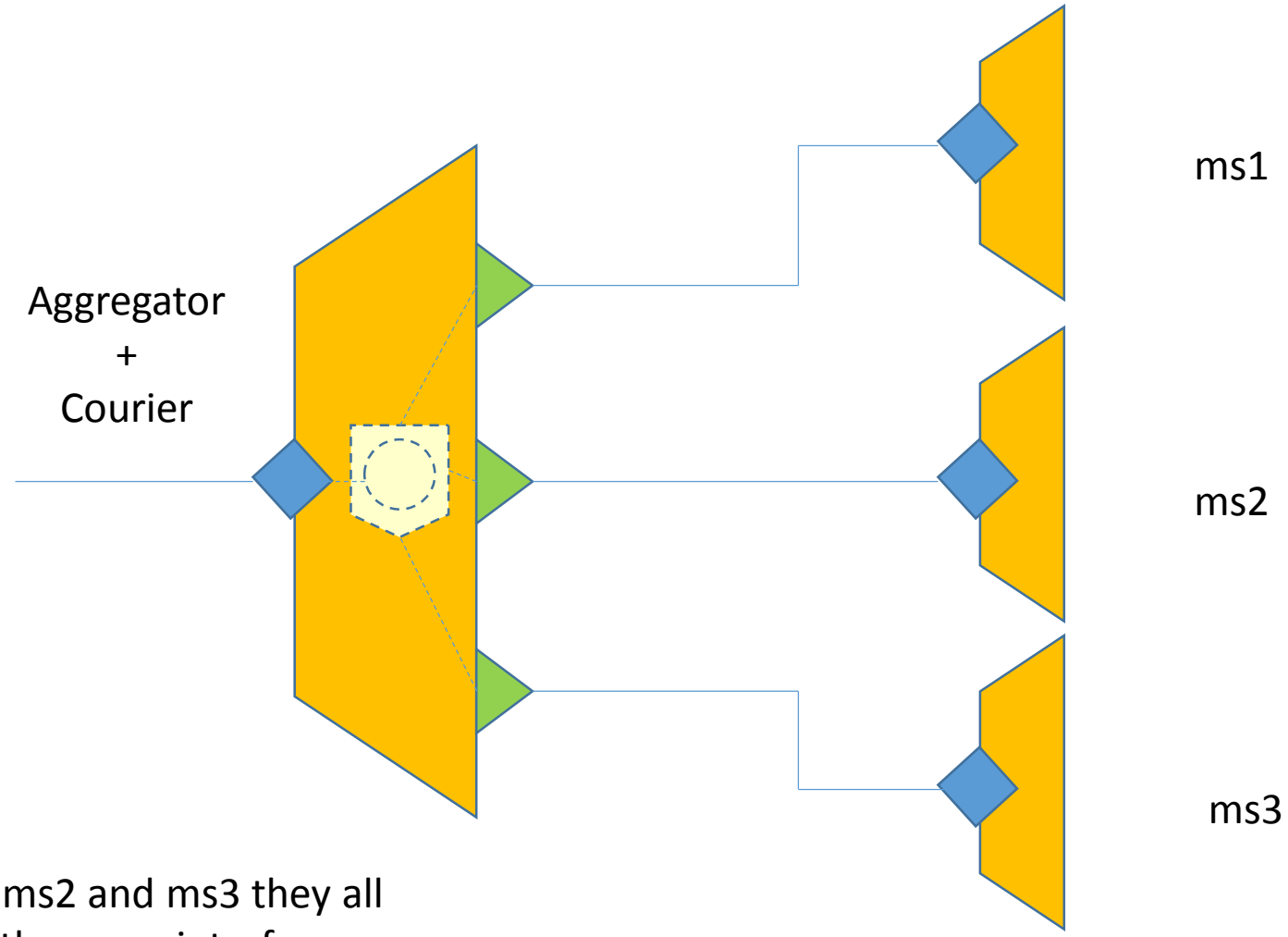
Extended message which carries additional context infos (ex: security tokens)

Message without extension, forwarded to the target service

italianaSoftware

imola

Collection

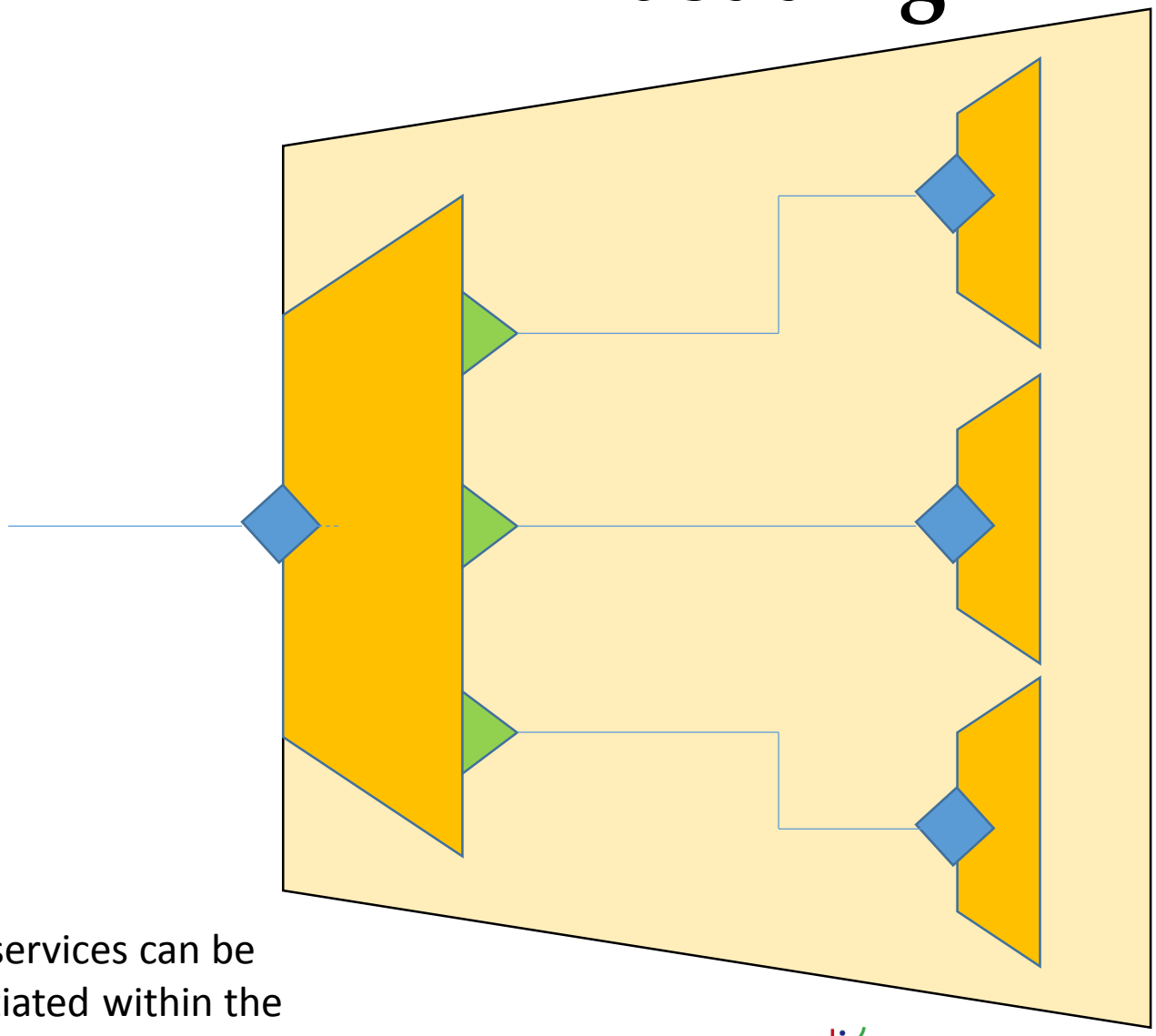


Ms1, ms2 and ms3 they all have the same interface. Forwarding is decided by using the extended information of the message.

italianaSoftware

imola

Embedding



More services can be instantiated within the same microservice

italianaSoftware

imola

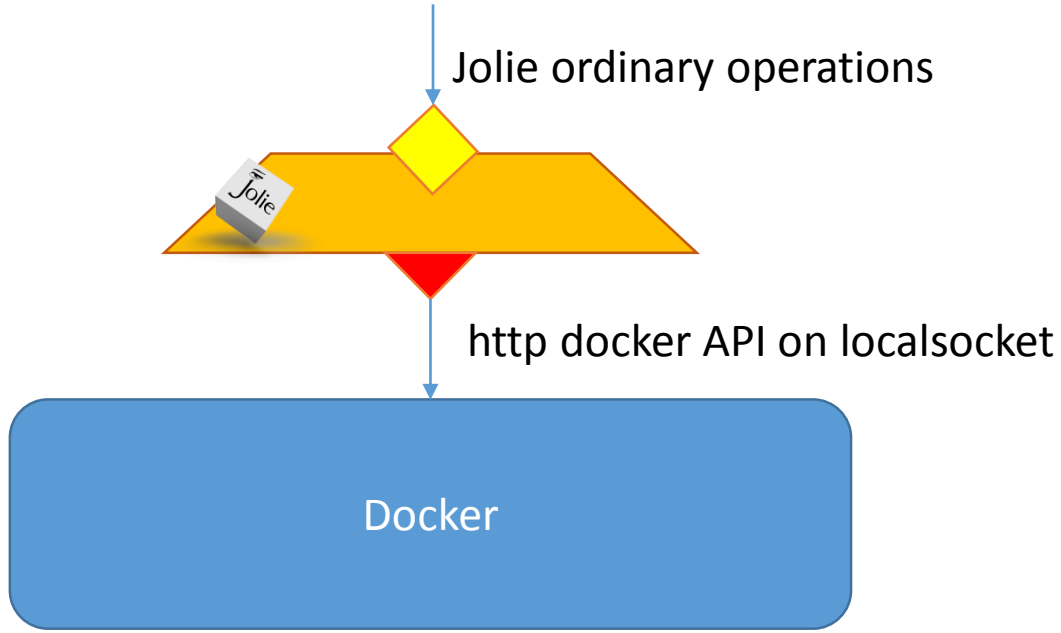
Jolie & Docker

italianaSoftware

imola

Jocker

docker pull jolielang/jocker

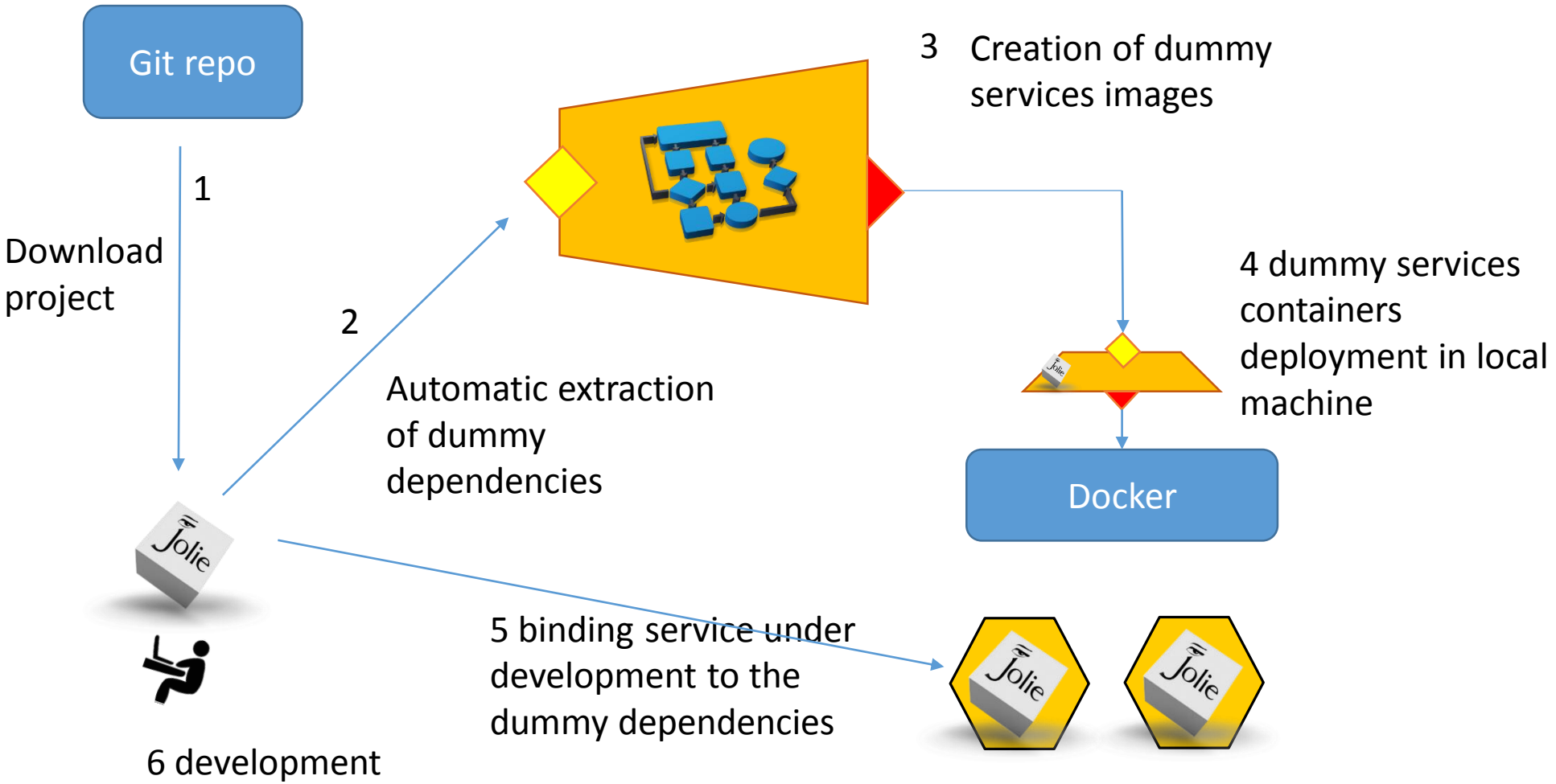


italianaSoftware

imola

Developing

(experimental)

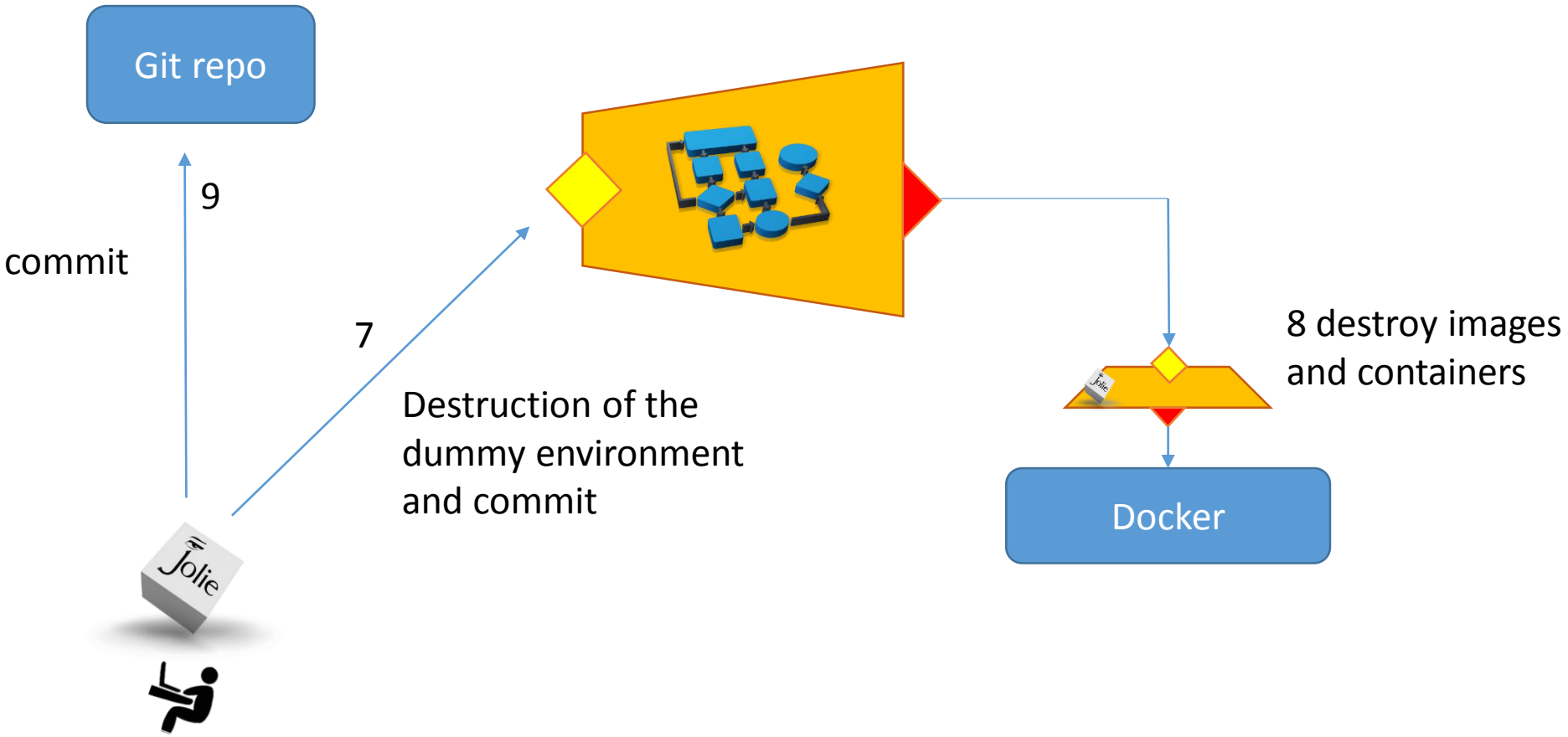


italianaSoftware

imola

Developing

(experimental)

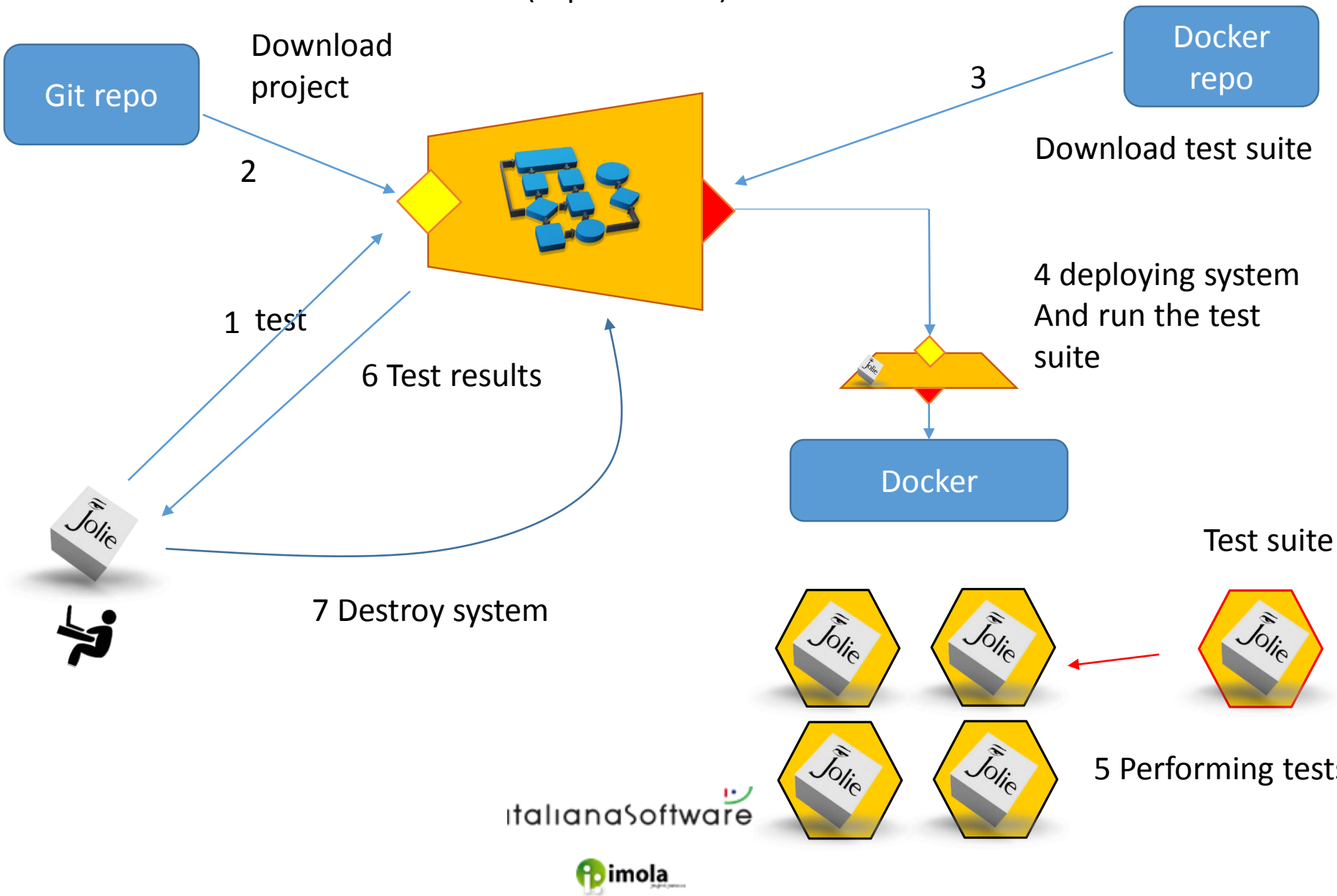


italianaSoftware

imola

Testing

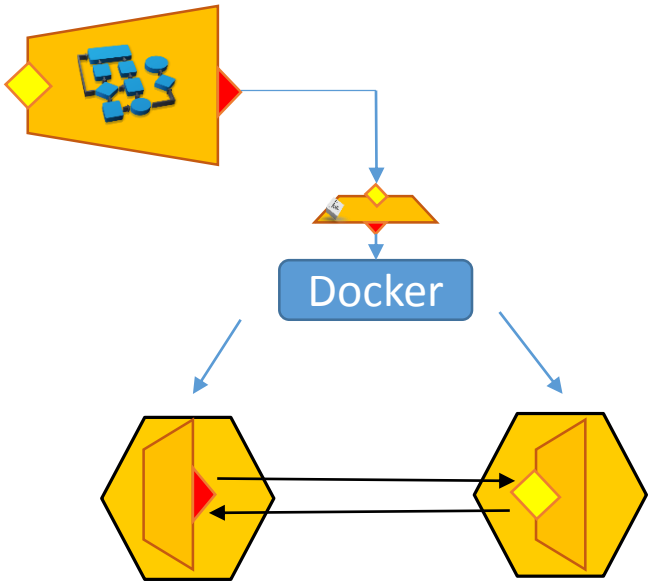
(experimental)



Dynamic Architectures

(Experimental)

From synchronous…

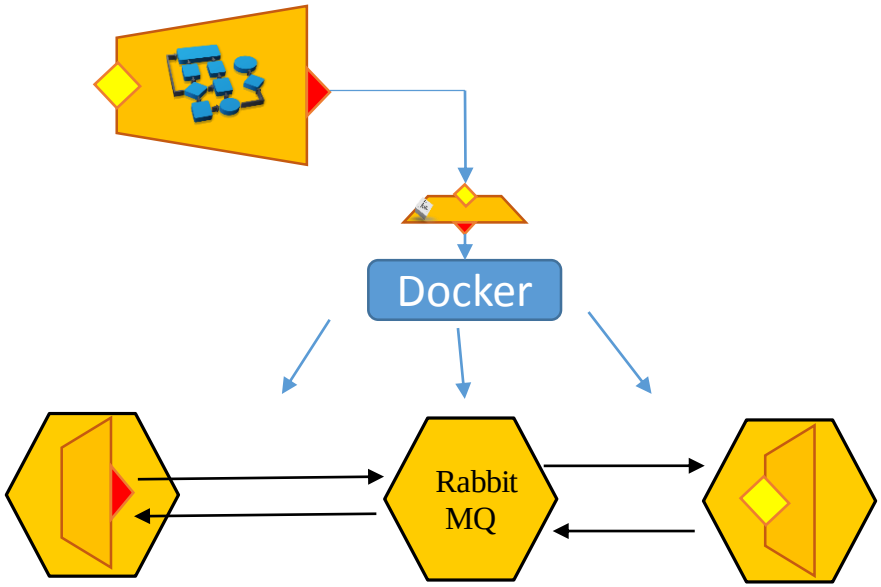



```
outputPort OPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}

main {
    request.field = ''hello world!''
    testRR@OPName( request )( response );
    …other activities…
}
```

```
inputPort IPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}

main {
        testRR( request )( response ) {
        …other activities…
        }
}
```

Dynamic Architectures

(Experimental)
…to asynchronous




```
outputPort OPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}

main {
    request.field = ''hello world!''
    testRR@OPName( request )( response );
    …other activities…
}
```

Keeping the code synchronous!!!!

```
inputPort IPName {
Location: ''socket://230.230.230.230:8000''
Protocol: sodep
Interfaces: MyInterface
}

main {
        testRR( request )( response ) {
        …other activities…
        }
}
```

Integration with other technologies

italianaSoftware

imola

Just a brief list of possibilities

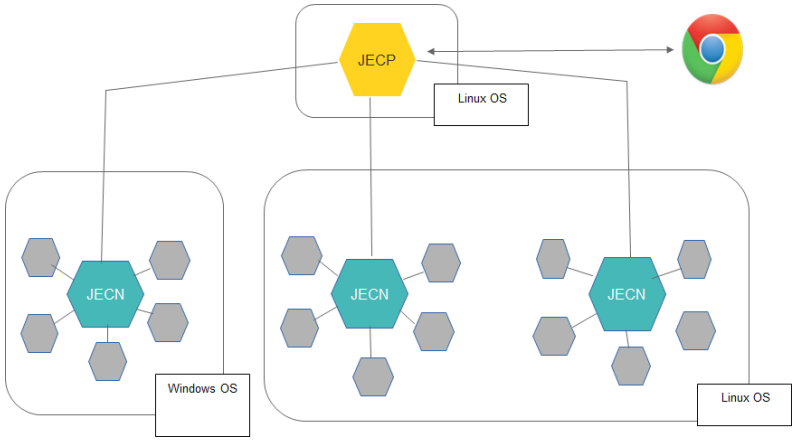
- ***Java***
  - There is a native possibility to integrate Java code into a Jolie service (JavaServices)
  - It is possible to run a Jolie service within an hosting application server (Ex: JBoss)
  - It is possible to send sodep messages to Jolie from a third party Java application
- ***Javascript***
  - As for Java, it is possible to embed javascript code into a Jolie service
- **Text Editors**
  - Plugins for **Atom** and **Sublime Text**
- ***HTTP and HTTPs***
  - *HTTP and HTTPs are supported protocols*
- ***SOAP Web Services***
  - SOAP is a supported protocol
  - jolie2wsdl and wsdl2jolie are tools which permit to convert Jolie interface into WSDL documents and viceversa
- **JSON**
  - JSON format is supported as format for the http Protocol
- ***Web***
  - **Leonardo** is a web server written in Jolie.
- ***REST***
  - Thanks to the usage JSON messages can be exploited just parameterizing a port
  - If necessary, it is possible to implement a standard REST services by exploiting **Jester**, a REST router for jolie services
- ***Databases***
  - SQL databases can be easily connected to a Jolie service using JDBC libraries.
  - MongoDB connector (by Balint Maschio)

Jolie in industry

italianaSoftware

imola

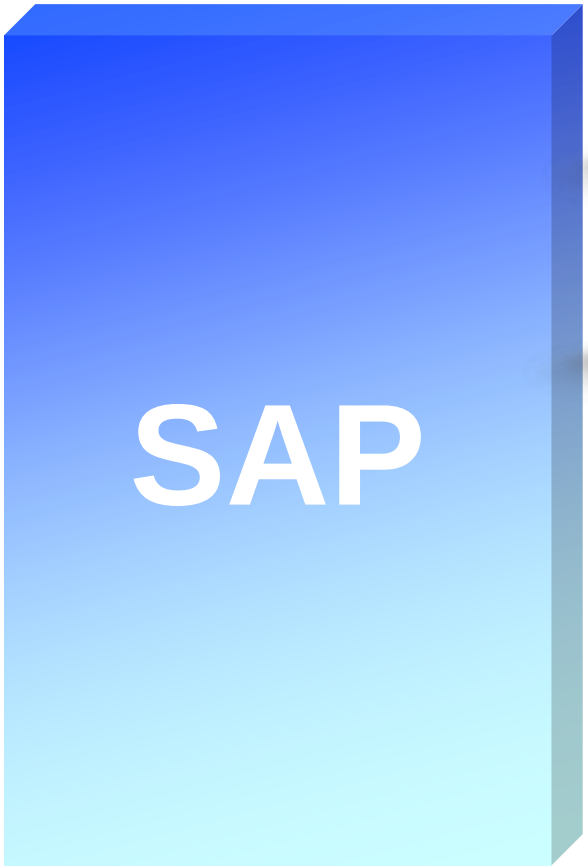
Jolie Enterprise

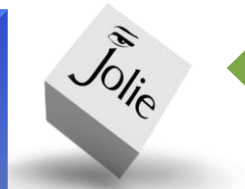


- Central control panel
  - Deployment
  - Start/stop
- Logging and monitor
  - All the Jolie service can be monitored and logged

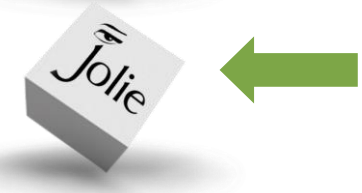
- Used as a digital platform for system integration and business process development

JSAP

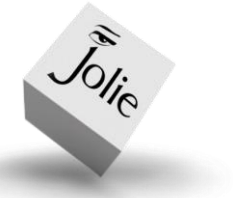




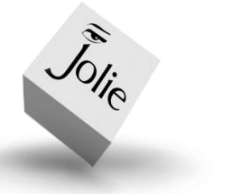
RFC client: it permits to call SAP RFC



IDOC client: it permits to send IDOC to SAP

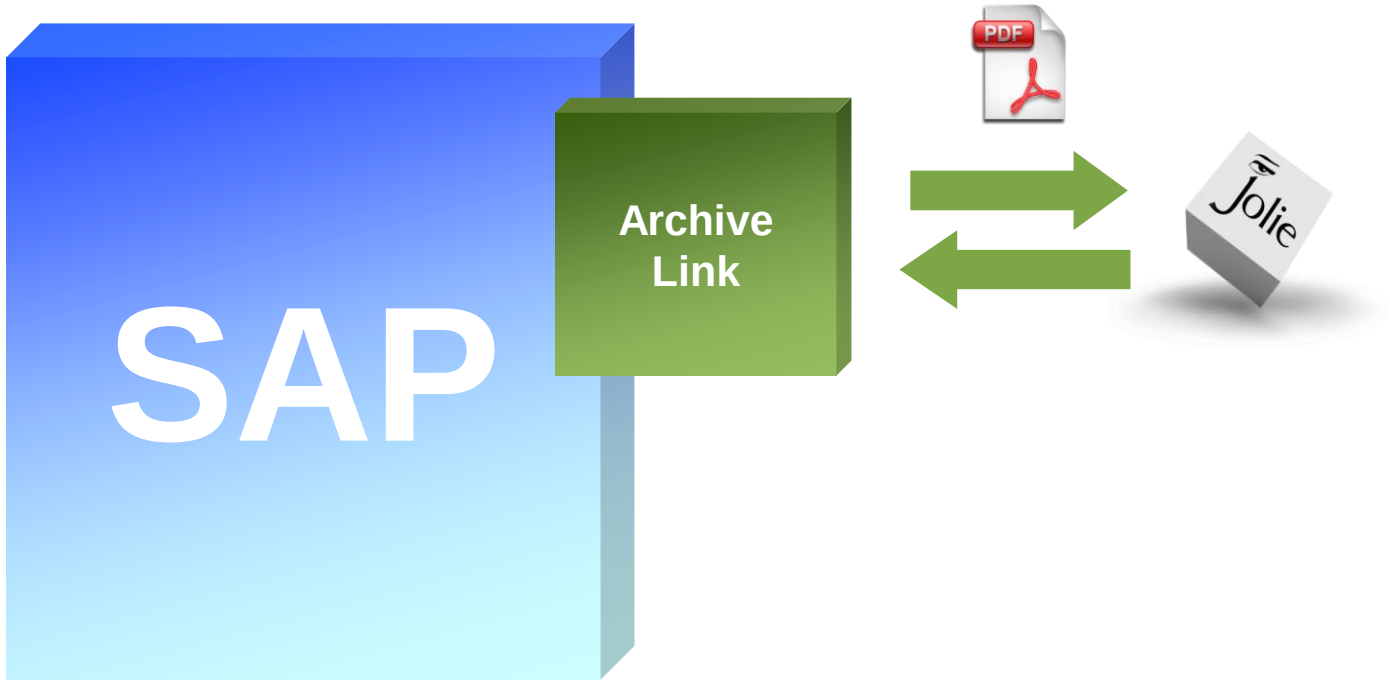


IDOC server:
It permits to receive IDOC documents from SAP



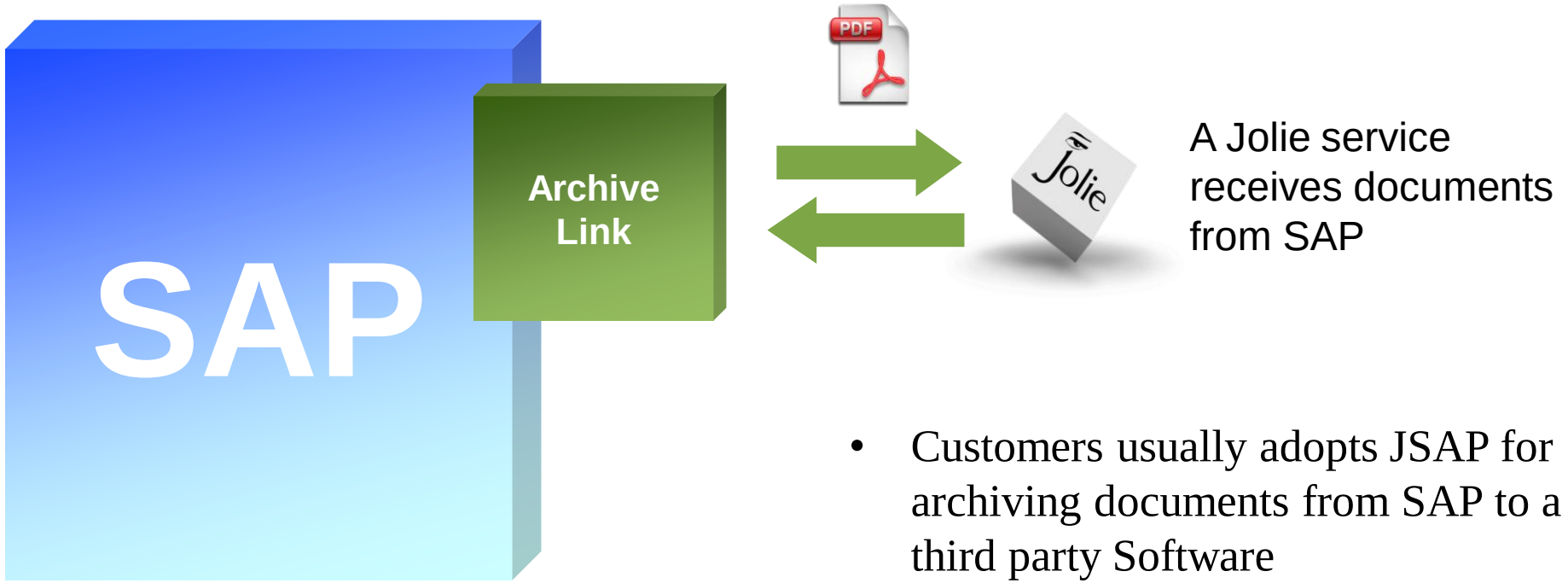
RFC server:
It permits to receive RFC calls from SAP

JSAP



A Jolie service receives documents from SAP

JSAP



A Jolie service receives documents from SAP

- Customers usually adopts JSAP for archiving documents from SAP to a third party Software

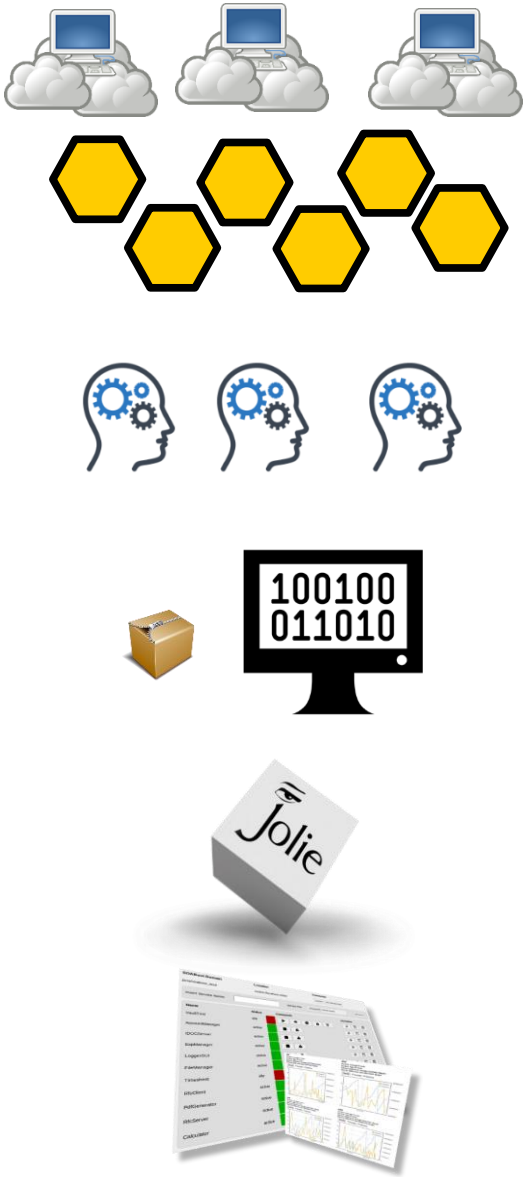
Conclusion

italianaSoftware

imola

Summarizing...

- A new computational resources paradigm has been introduced by cloud computing
- Software distribution is the natural paradigm for approaching such a kind of systems
- The high level of complexity of distributed systems require to reduce the required knowledge for managing them
- A new generation of languages which crystalize the basic microservice programming principles could help in reducing the required knowledge
- Jolie is a good candidate for representing this new generation of languages
- Jolie is already used as a technology in the industry with successfull results



Thank you

italianaSoftware

imola