Autonomous microservices for a Financial System

INPAY

THE GLOBAL PAYMENTS NETWORK

Jeppe Cramon - @jeppec

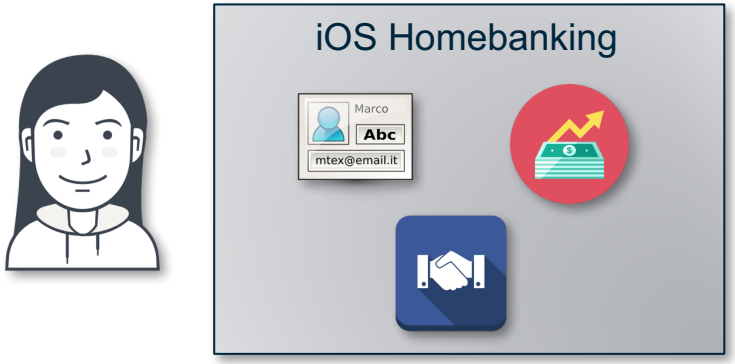
Chief Architect - **INPAY**

Let’s start with the end in mind

INPAY

Applications

INPAY

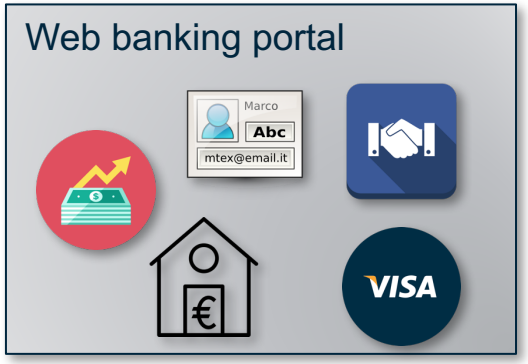


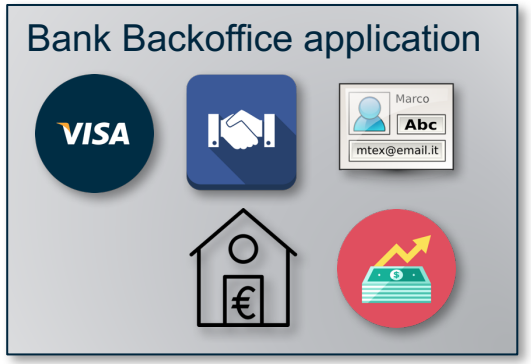
Customer information

Legal and contract information

Accounts

INPAY





Customer information

Legal and contract information

Accounts

Credit card

Mortgage loans

INPAY

Applications form compositions

INPAY

Table-like Data Composition

This example shows how client-side events are used to aggregate data which is then composed in a table-like manner.

Filter

Order	Placed on	Customer	Order Total
9876	01-05-2015 14:01:11	Joe	$8,400.00
23123	18-06-2015 22:13:11	Michelle	$25,350.00
102343	22-06-2015 12:43:01	John	$950.00

INPAY

Invoice

Customer Details

Name	Joe Mchannan
Address	565-4979 Blandit Rd. Schepdaal
Country	Lesotho

Order Details (9876)

Monitor	37" awesome 8K monitor	2	$400.00
SSD	Fantastic solid state disk	1	$7,000.00
Commodore 64	Old stuff rulez :-)	1	$1,000.00

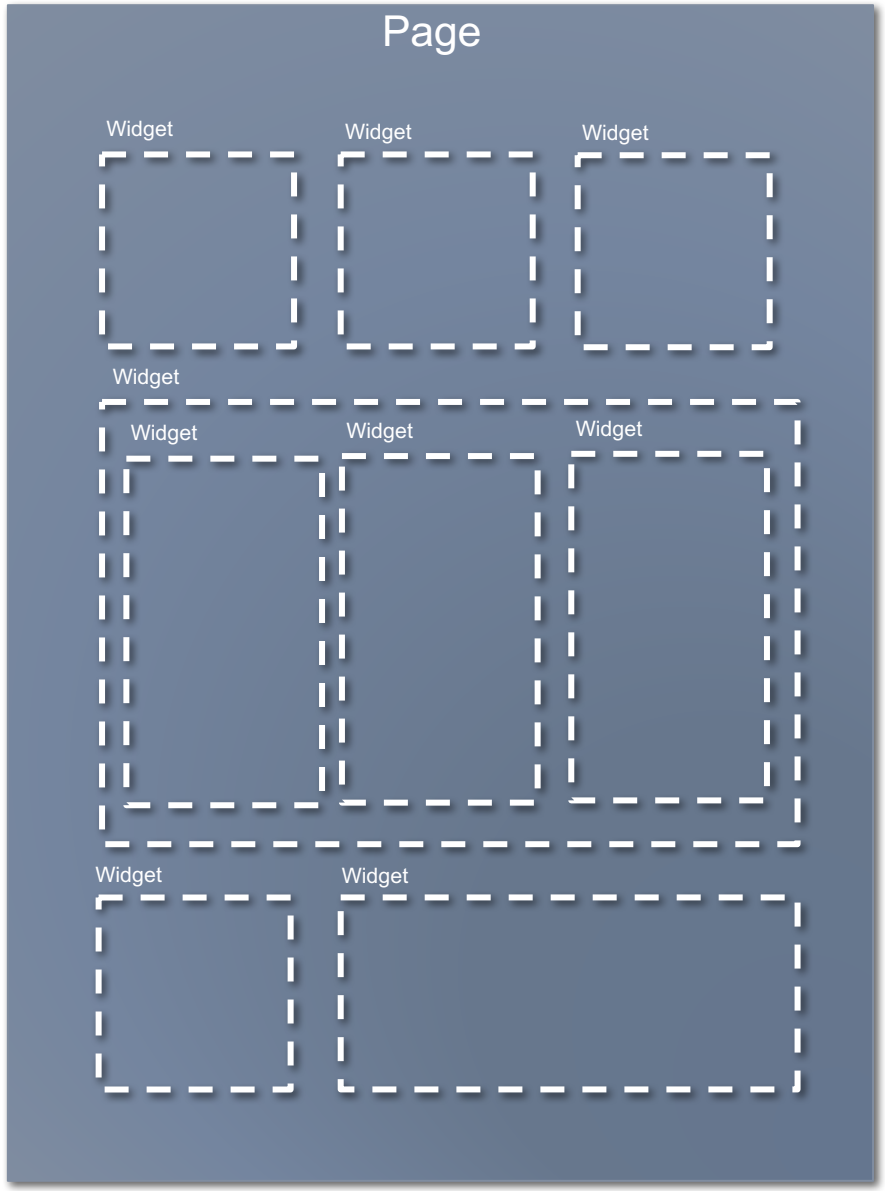
Total:

$8,400.00

INPAY

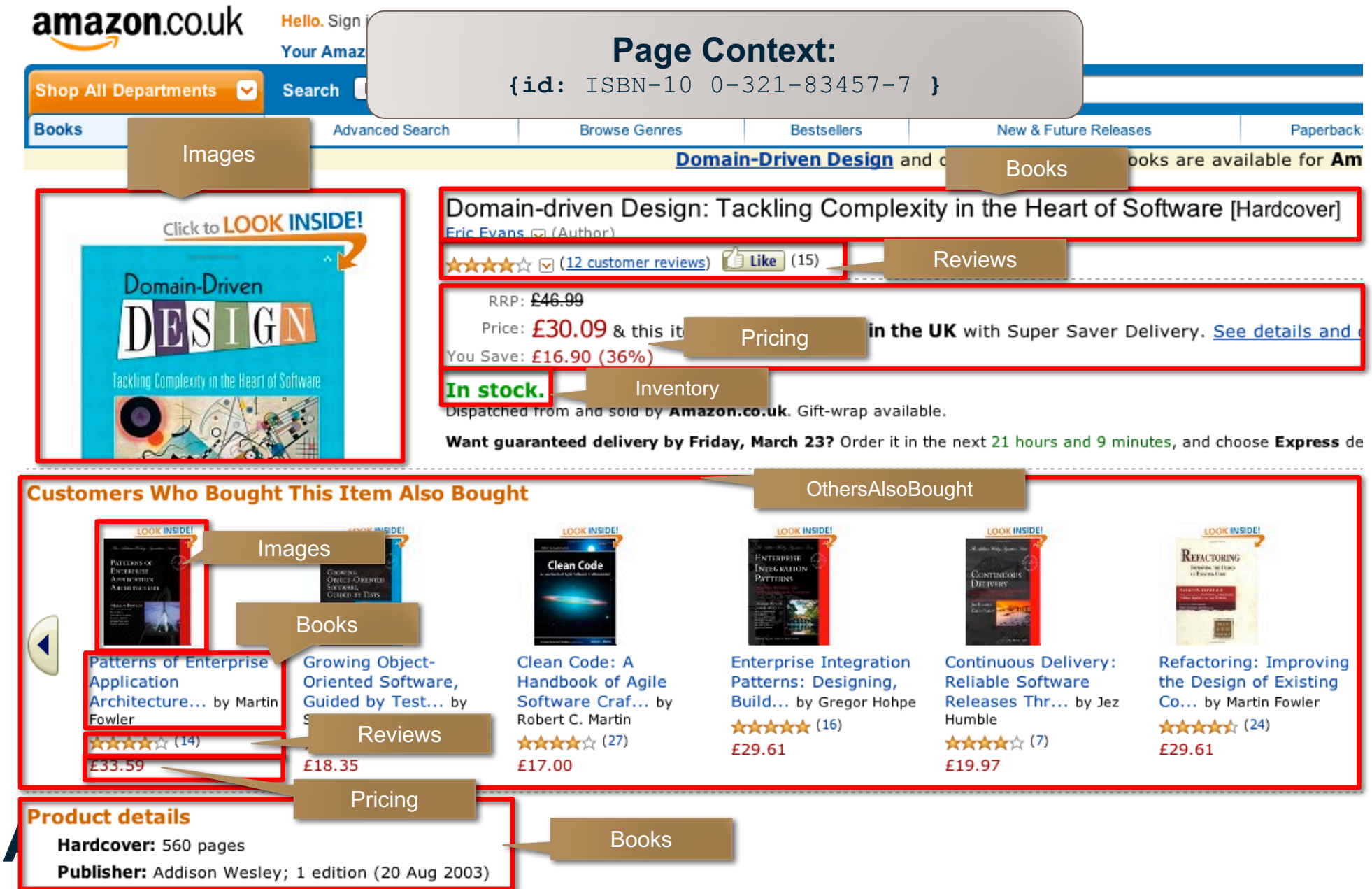
Composite page layout

- Overall structure of the page is “owned” by the application.
- Each widget is independent



INPAY

Composite page example

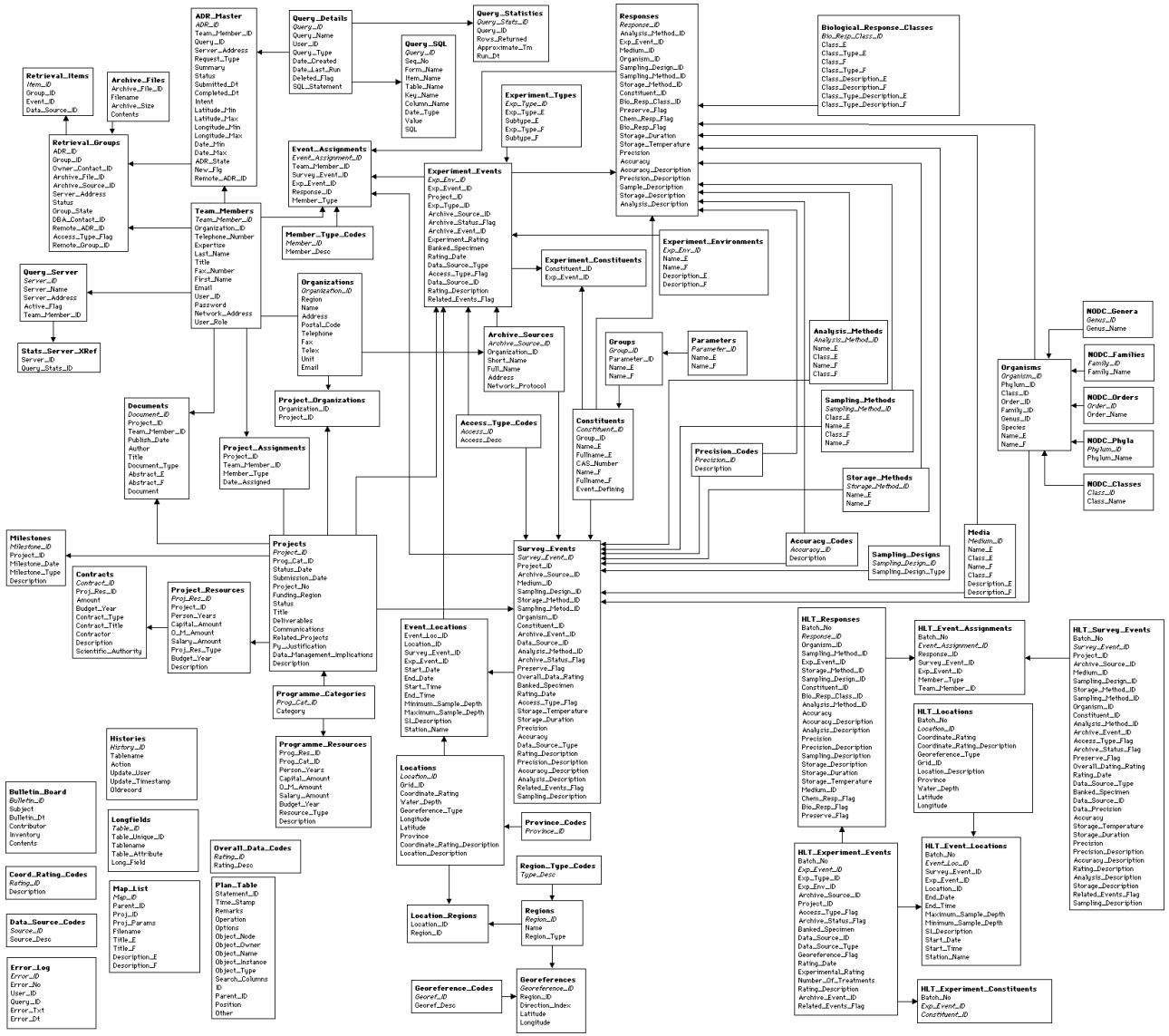


INPA

How could we support this?

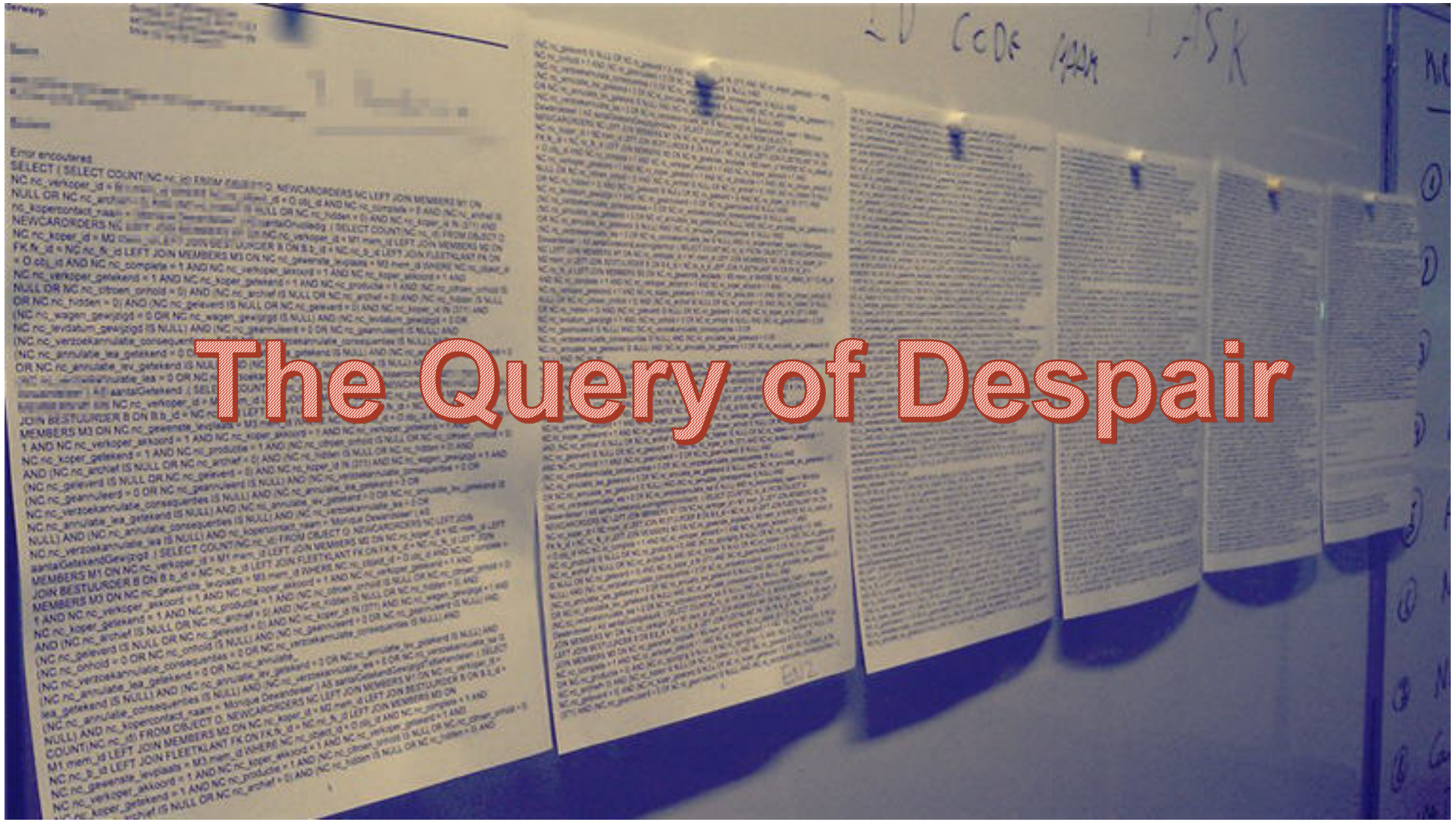
INPAY

One underlying domain model to solve it all?



INPAY

Reality rears its ugly head



INPAY

The result is often rigidity

Changing one thing requires you to modify a lot of other code to make the codebase consistent again

INPAY

…fragility

When you change a thing you end up breaking something else!

INPAY

If we do it wrong, the end result is

"Nobody is allowed to modify that module!"

or

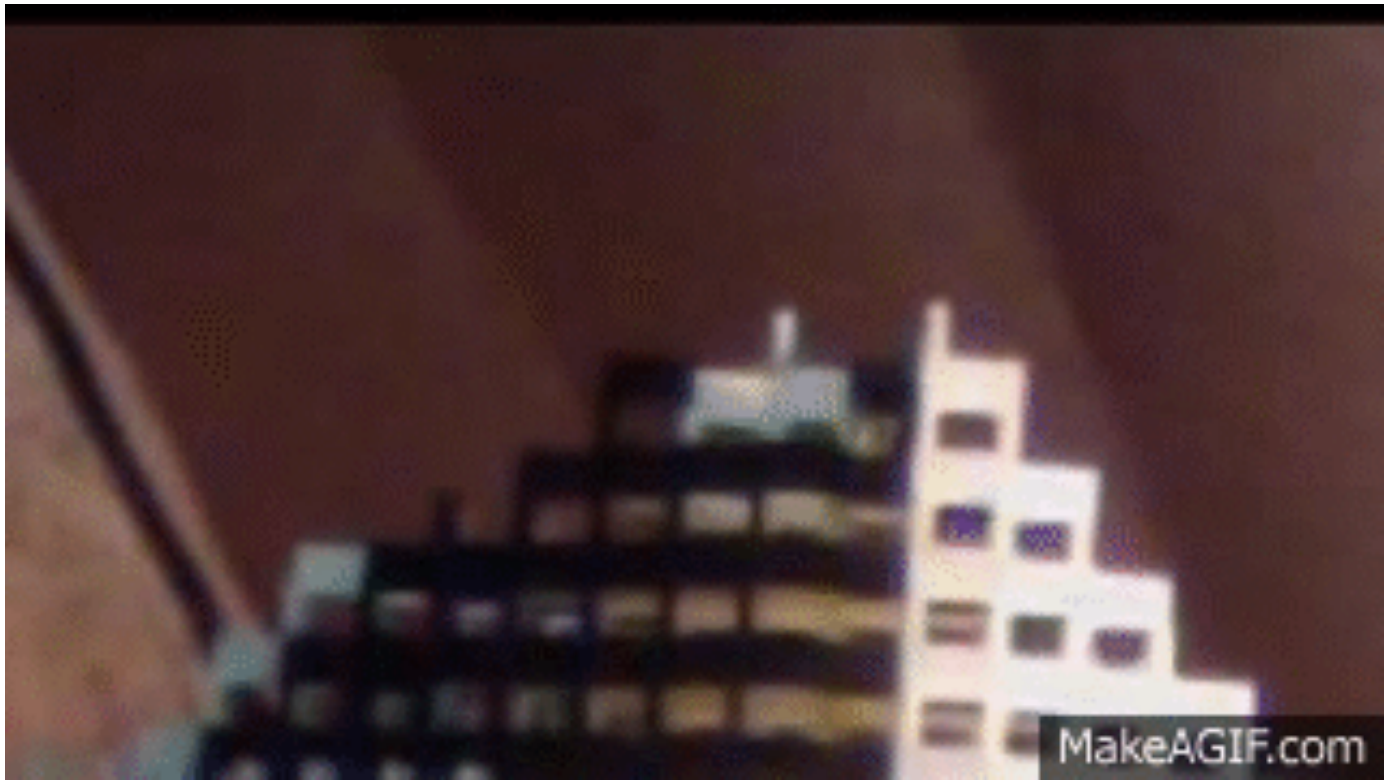
"Let's rewrite!"

INPAY

CORE PROBLEM?

INPAY

High coupling



INPAY

What is the right level of coupling?

This highly depends on

- How likely are things to change
- And what parts that change together

INPAY

Problem domain analysis

INPAY

Classical domain analysis tends to

INPAY

Focus on **Nouns** (Entities)
and retrofit *Verbs* later

Result is often fragmented domain logic

INPAY

If we primarily model around nouns/entities we can easily violate the SRP

Where a change to requirements is likely to require changes to multiple entity classes

"We don't sell domains, WE SELL USE CASES!"

Jim Coplien

INPAY

Focus on **Verbs (Use-cases)**

instead of focusing on Nouns!

INPAY

Also, when discussing use cases with Business Experts

Focus on **data fields** and NOT on entities

INPAY

Identify what data fields **clump** together

and what fields that are **separate**

INPAY

Place the **fields** in **piles**

INPAY

But don’t name the piles before you know what they are

This avoids cognitive bias!

INPAY

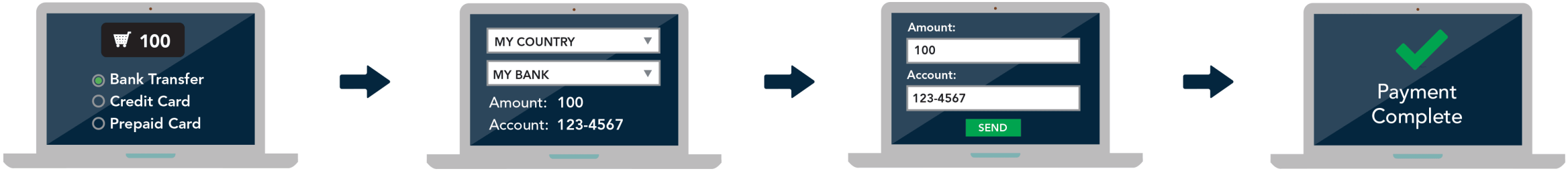
Give the piles **made up** names

Let's illustrate by example…

INPAY

What does INPAY do?

PayIn



An online shopper chooses to pay by bank transfer.

The online shopper selects a country and bank and receives transaction details.

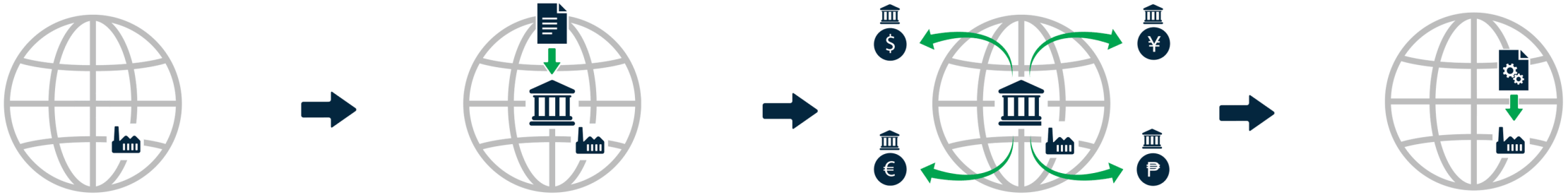
The online shopper logs in to a personal bank account and enters transaction details. Payment is sent to Inpay.

Confirmation of payment is displayed to the online shopper. Inpay informs the merchant when the funds are received.

INPAY

What does INPAY do?

PayOut



A company needs to make payments to various countries.

The company sends payment instructions and transfers funds to its local Inpay account in local currency.

Inpay fulfills the payment instructions, releasing the appropriate sum in local currency from the local Inpay account.

The company receives API call confirmation from Inpay.

INPAY

INPAY data fields

Blue

Bank Identifiers (BIC, ...)
Bank Contact Details
Bank Account Identifiers (IBAN, ...)

Green

Bank Account Transactions (Debit/Credit)
Virtual Bank Account Transactions (Debit / Credit)
Virtual Bank Account Settlement cycle
Service Agreement Monthly fee / Wire transfer fees / ...

Red

Bank Account Currency
Bank Country of operation
Bank Fees
Bank Cutoff Times
Bank Allowed Industries / Verticals / IIC's
Disbursement/Collection/... - Quote
Disbursement/Collection/... – Request
Service Agreement – Fees / Cutoff Times / Currency Agreements / etc.

INPAY

Now with names…

Banking

Bank Identifiers (BIC, …)
Bank Contact Details
Bank Account Identifiers (IBAN, …)

Virtual Banking

Bank Account Transactions (Debit/Credit)
Virtual Bank Account Transactions (Debit / Credit)
Virtual Bank Account Settlement cycle
Service Agreement Monthly fee / Wire transfer fees / …

PSP

Bank Account Currency
Bank Country of operation
Bank Fees
Bank Cutoff Times
Bank Allowed Industries / Verticals / IIC's
Disbursement/Collection/… - Quote
Disbursement/Collection/… – Request
Service Agreement – Fees / Cut off Times / Currency Agreements / etc.

INPAY

What are the these problem domain piles?

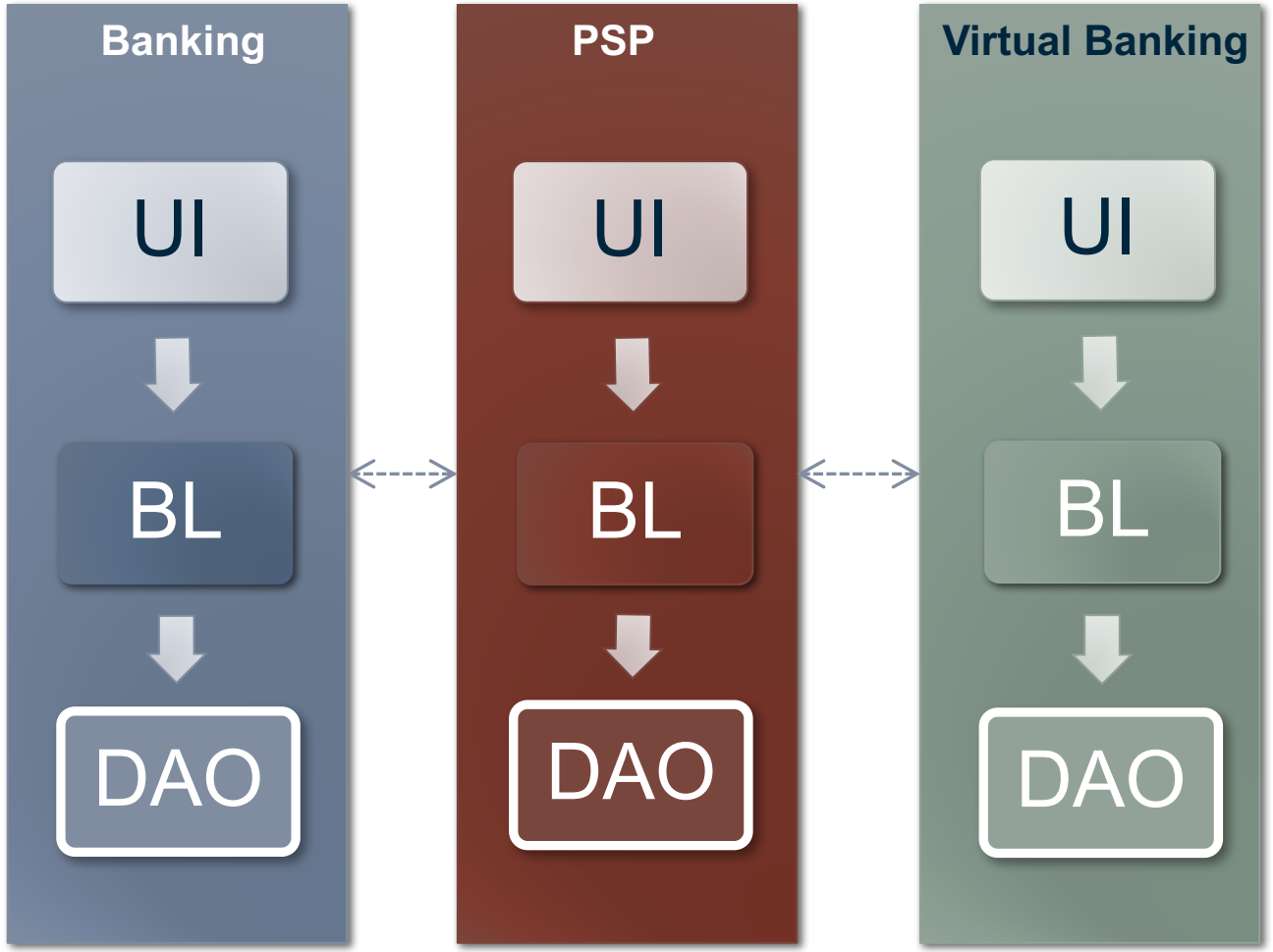
Bounded Contexts

INPAY

Solution domain design

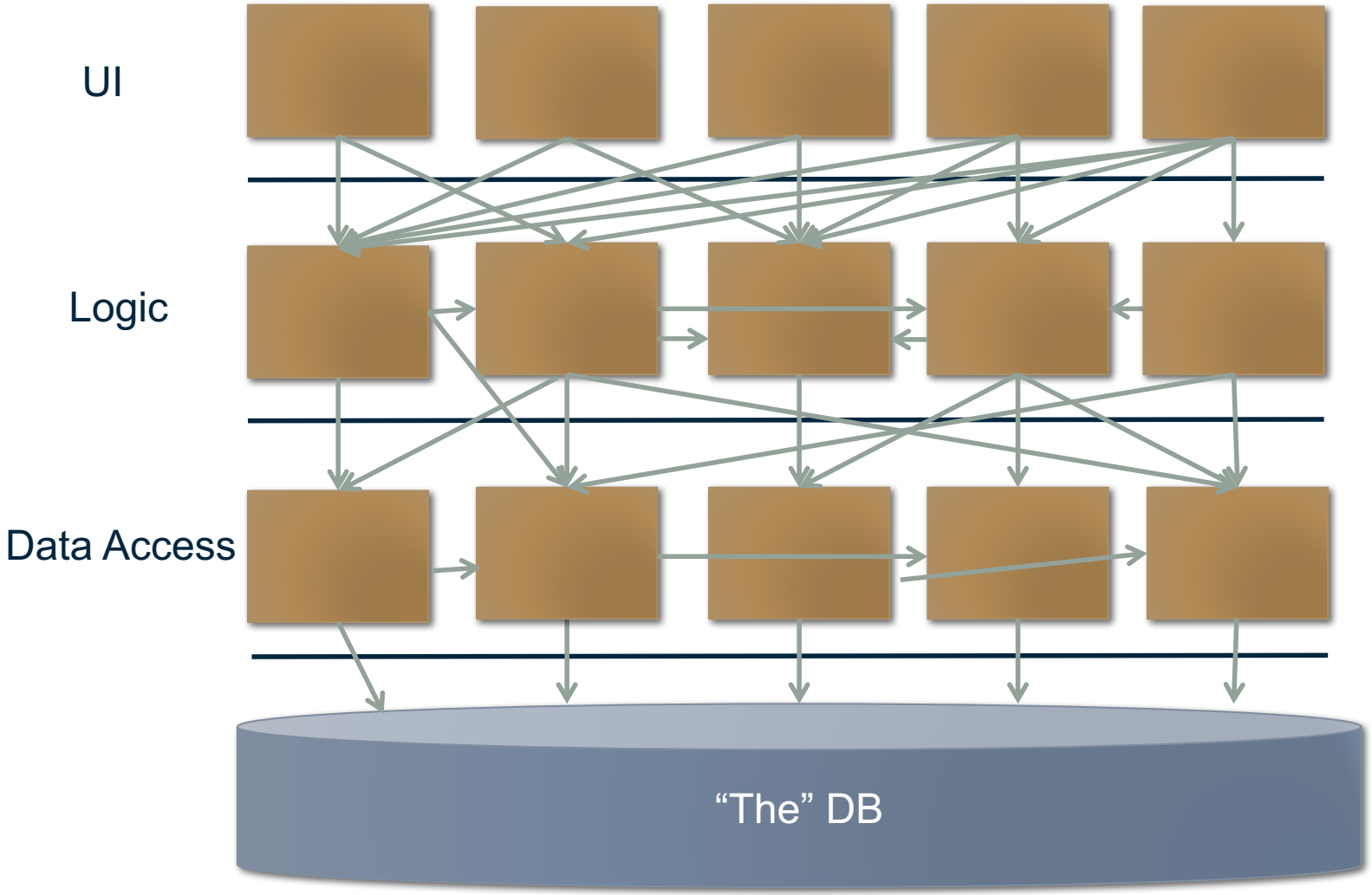
INPAY

If we align the problem domain with the solution domain



INPAY

Which is very different from



INPAY

What are the solution domain piles?

Services

INPAY

A Service is

- The technical **authority** for a given **bounded context**
- It is the **owner** of all the **data** and **business rules** that support this bounded context – **everywhere**
- It forms a single source of truth for that bounded context

INPAY

http://udidahan.com/2010/11/15/the-known-unknowns-of-soa/

But where do microservices fit into this?

INPAY

Microservices promise **a** solution to our problem

Monolith

Microservice

Microservice

Microservice

Microservice

Microservice

Microservice

Microservice

INPAY

There’s never time to do it right

But there’s always time to do it over

INPAY

Service design

INPAY

If we want a scalable and loosely coupled design

We could seek inspiration from…

INPAY

Life Beyond Distributed Transactions by Pat Helland

1. How do we split our data
2. How do we identify our data
3. How do we communicate between our services

INPAY

1. How do we split our data

Data must be collected in pieces called **aggregates**. These aggregates should be **limited in size (but not smaller)**, so that, **after** a **transaction** they are **consistent**.

Rule of thumb:

One transaction involves only **one aggregate.**

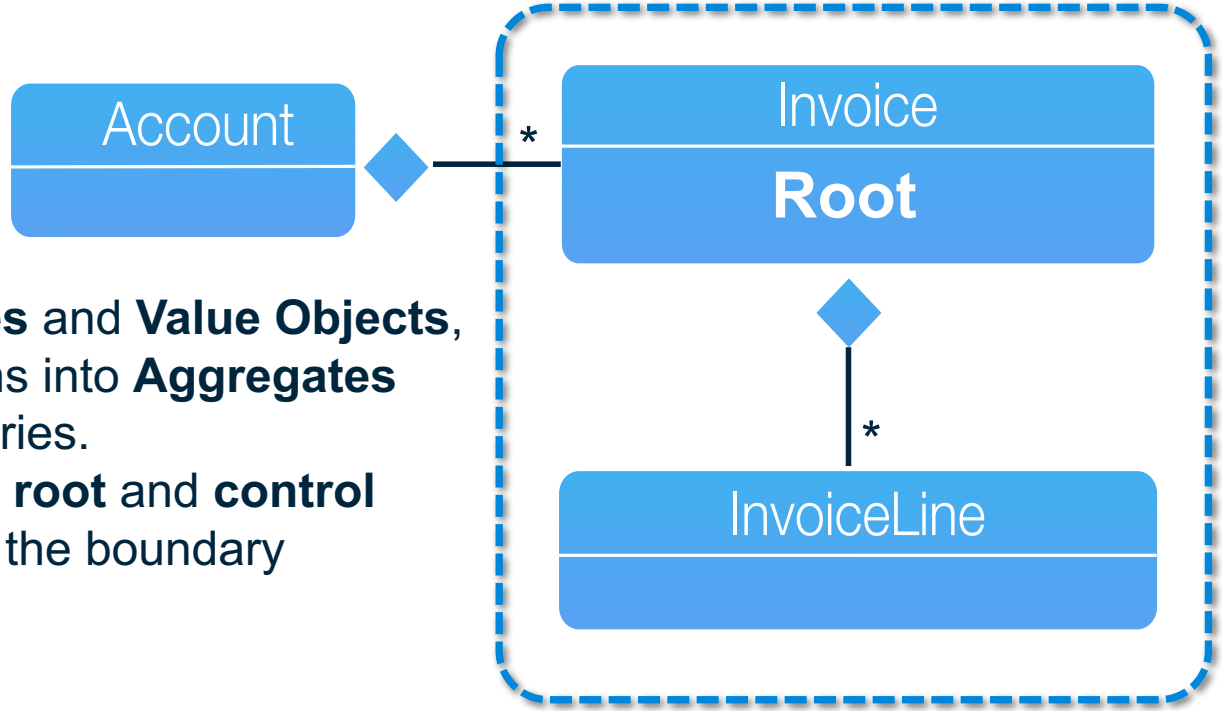
INPAY

DOMAIN DRIVEN DESIGN

The term Aggregate comes from DDD

INPAY

Aggregates



What:

- Cluster **coherent Entities** and **Value Objects**, with complex associations into **Aggregates** with well defined boundaries.
- Choose **one entity** to be **root** and **control access** to objects inside the boundary **through the root**.

Motivation:
Control **invariants** and **consistency** through the **aggregate root**.

Ensuring consistency & transactional boundaries for Distributed scenarios!

INPAY

2. How do we identify our data

According to Pat Helland we need to be able to uniquely identify each Aggregate using an ID.

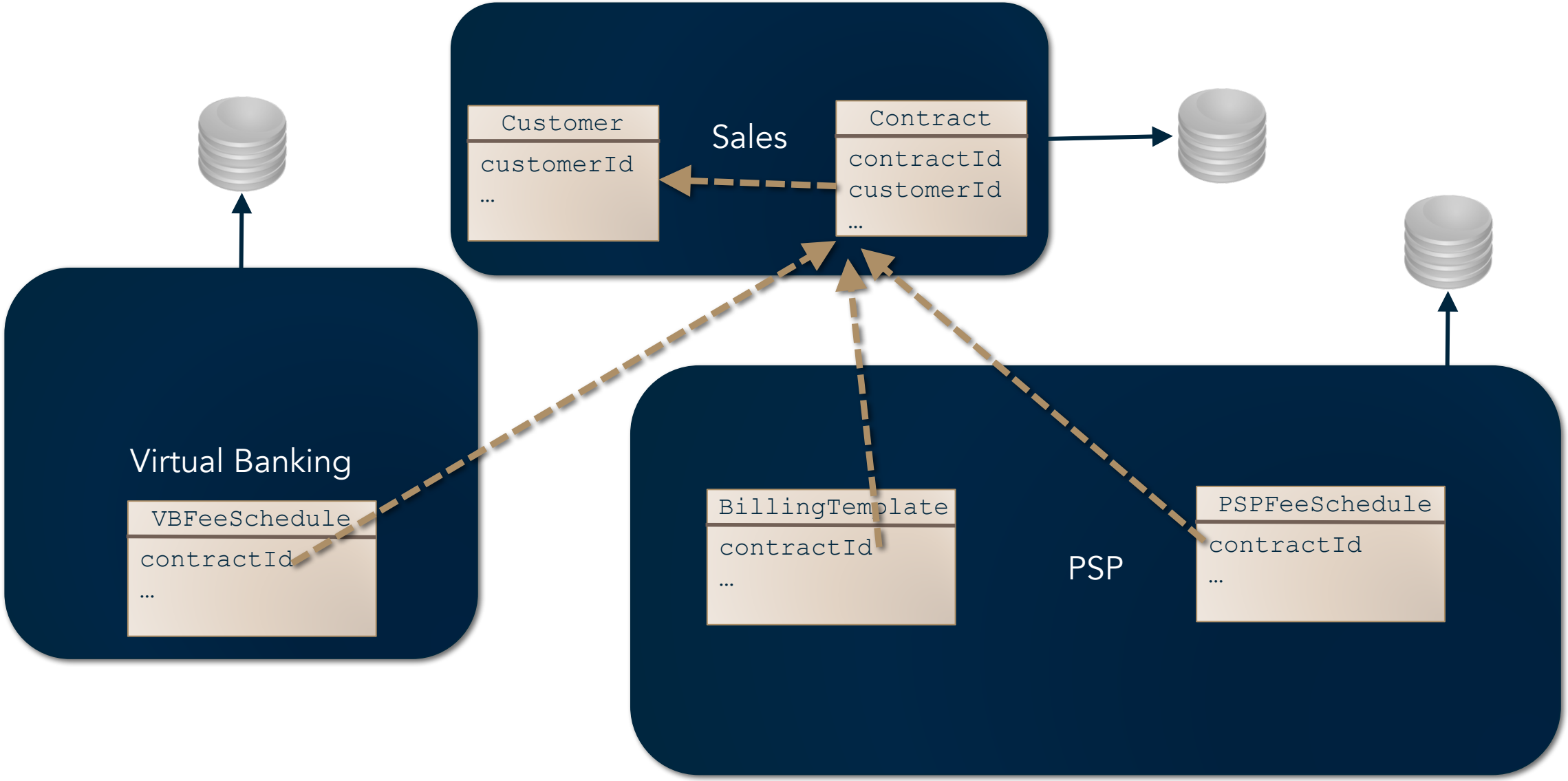
- This ID will usually a UUID/GUID
- **Aggregates refer to each other by their ID**
  - they **NEVER** use memory pointers, join tables or remote calls

`{21EC2020-3AEA-4069-A2DD-08002B30309D}`

2^{122} (approximately 5.3×10^{36}) combinations

INPAY

Services/Bounded Contexts and Aggregates



INPAY

3. How do we communicate between our services

- What do we do when our use case involves more than one aggregate and therefore likely more than one service?

INPAY

Synchronous calls are the **crystal meth** of **programming**

*At first you make good progress but then the sheer **horror** becomes evident when you realise the **scalability limitations** and how the **brittleness holds back** both **performance** and **development flexibility**. By then it is too late to save.*

*We **need** the **reactive** properties and then apply **protocols** for the **message** interactions. **Without** considering the **protocols** of **interaction** this world of **micro-services** will become a **coordination nightmare**.*

Martin Thompson

INPAY

http://www.infoq.com/news/2014/10/thompson-reactive-manifesto-2

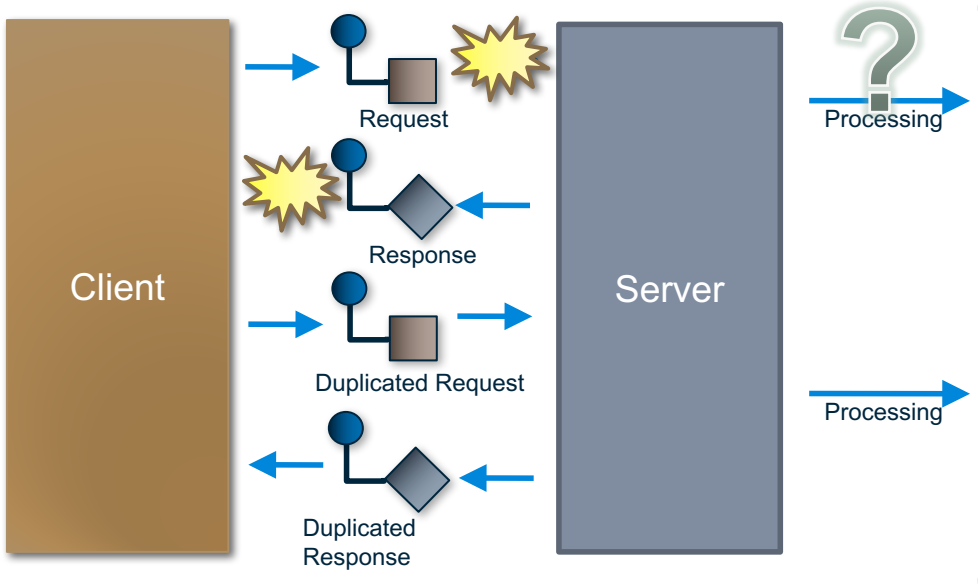
WHAT'S THE CHALLENGE WITH USING **RPC/REST/...** BETWEEN SERVICES?

INPAY

Synchronous calls lower our tolerance for faults

- When you get an IO error
- When servers crash or restarts
- When databases are down
- When deadlocks occurs in our databases
- **Do you retry?**

With synchronous style Service interaction we can loose business data if there's no automatic retry
Or we risk creating data more than once if the operation isn't idempotent*



The same message can be processed more than once

***Idempotence** describes the quality of an operation in which result and state does not change if the operation is performed more than 1 time*

INPAY

Also remember: REST isn't magic!

INPAY

WITH CROSS SERVICE INTEGRATION WE'RE BOUND BY THE LAWS OF DISTRIBUTED COMPUTING

INPAY

The 8 Fallacies of Distributed Computing

These fallacies are assumptions architects, designers and developers of distributed systems are likely to make. The fallacies will be proven wrong in the long run - resulting in all sorts of troubles and pains for the solution and architects who made the assumptions.

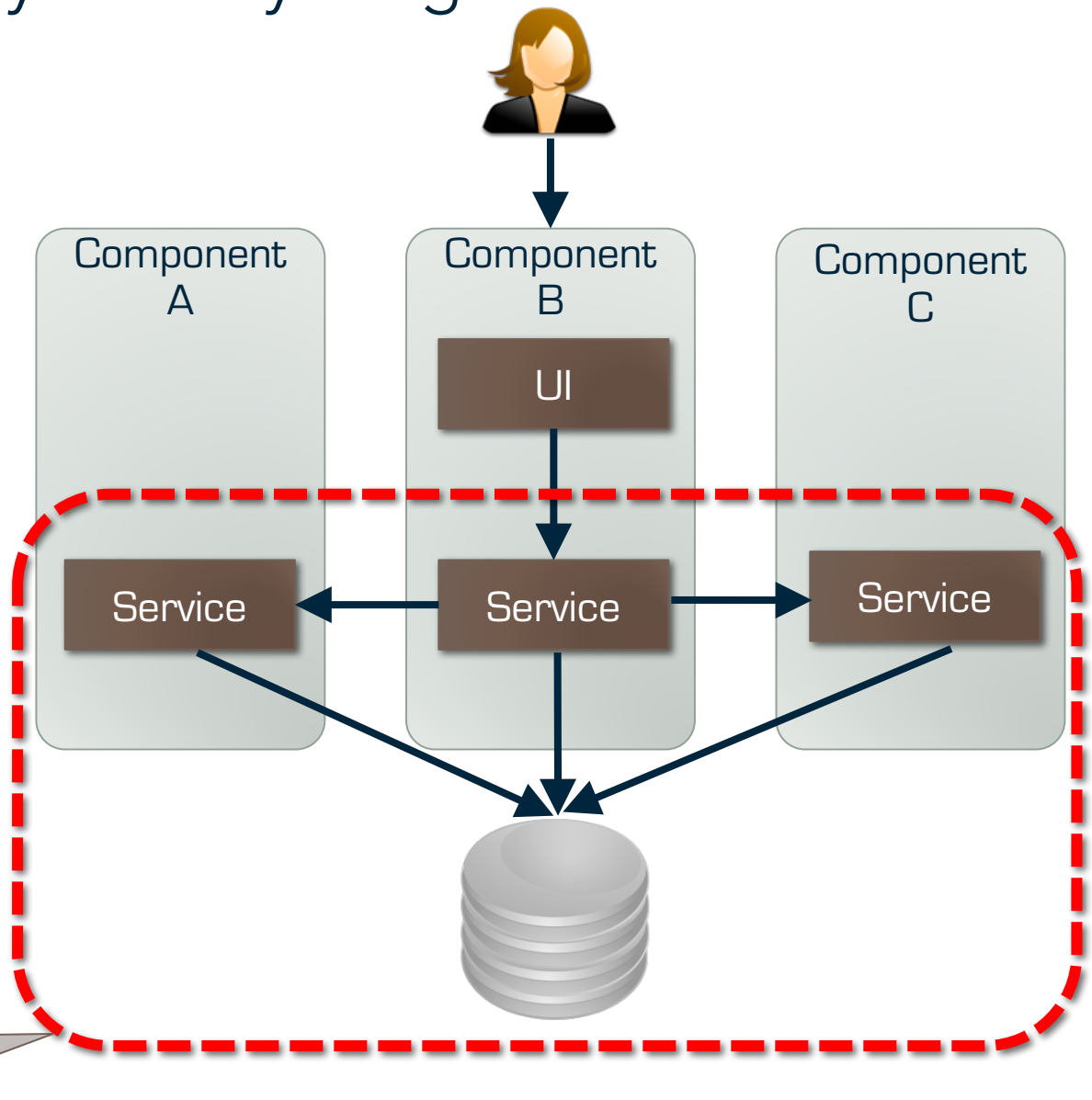
1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

See http://www.rgoarchitects.com/Files/fallacies.pdf for a walkthrough of the fallacies and why they're fallacies

INPAY

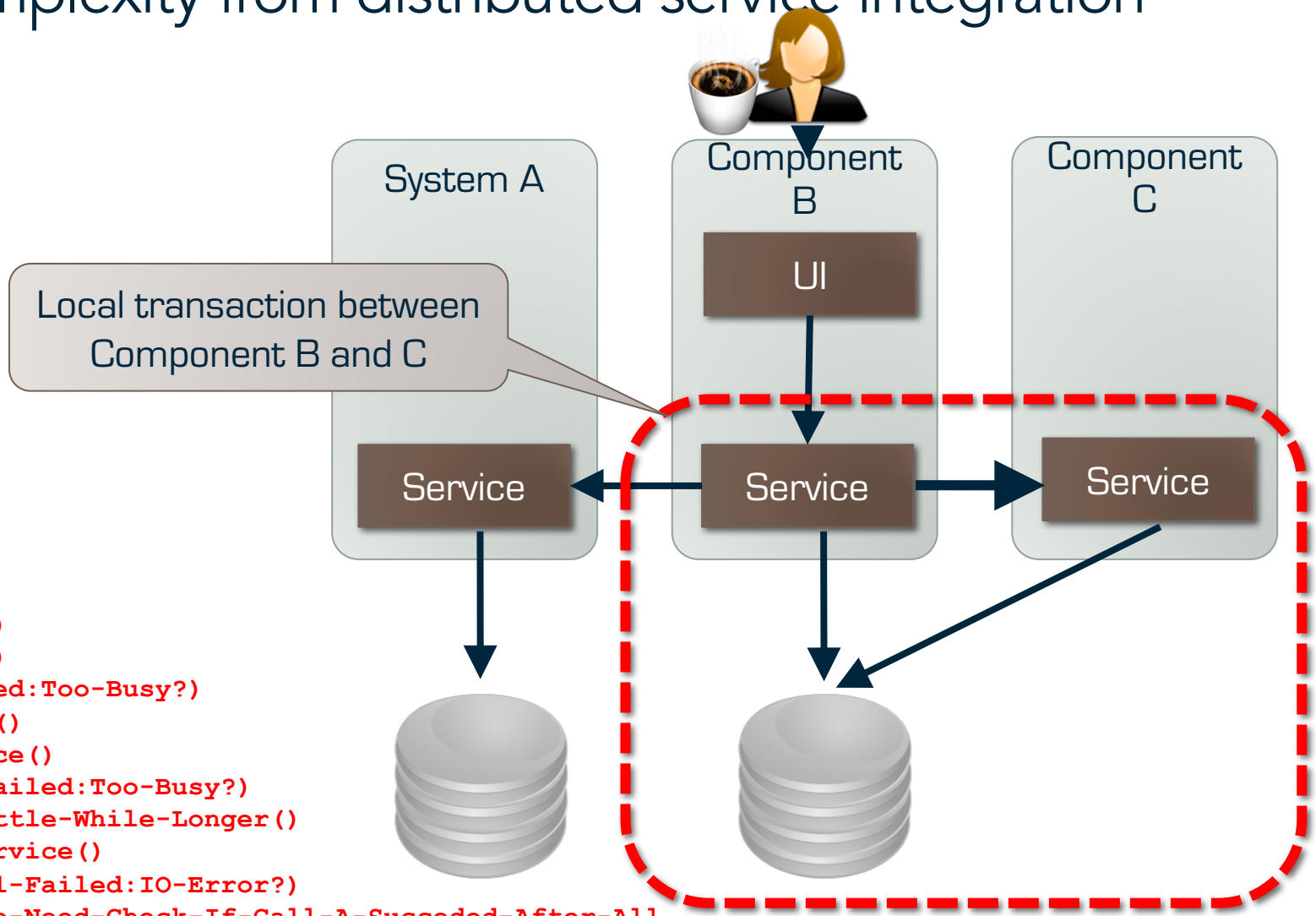
Essential complexity of 2 way integration

```
B:Service()➔
   call C:Service()
   call A:Service()
   commit()
```



INPAY

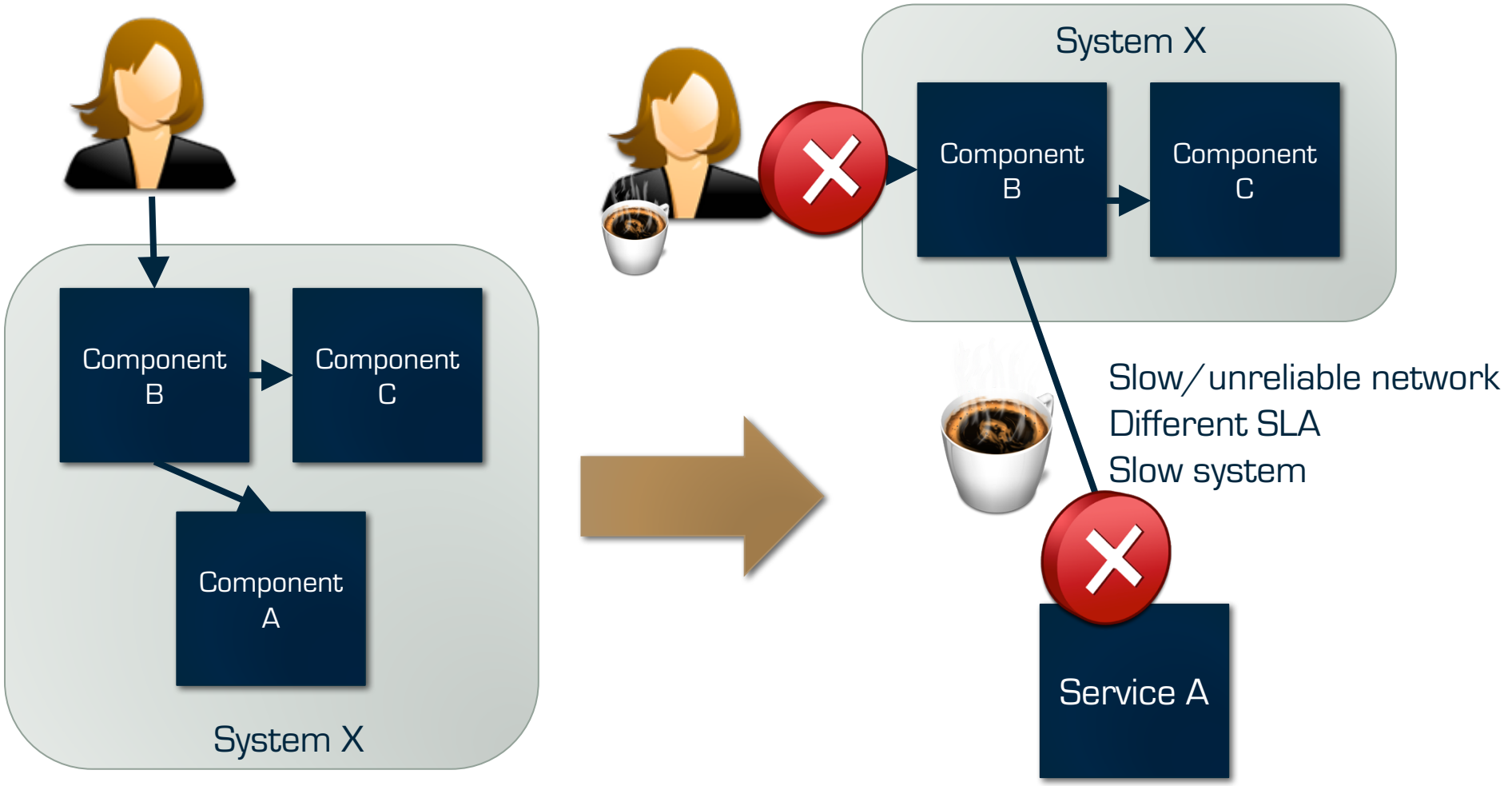
Accidental complexity from distributed service integration




```
B:Service()➔
  call C:Service()
  call A:Service()
  if (A:Call-Failed:Too-Busy?)
     Wait-A-While()
     call A:Service()
     if (A:Call-Failed:Too-Busy?)
        Wait-A-Little-While-Longer()
        call A:Service()
        if (A:Call-Failed:IO-Error?)
           Save-We-Need-Check-If-Call-A-Succeded-After-All
           AND We-Need-To-Retry call C:Service and call B:Service
           AND Tell-Customer-That-This-Operation-Perhaps-Went-Well
  if (A:Call-Went-Well?)
     commit()
```

INPAY

Service autonomy



INPAY

Clarification of Autonomy vs. Authority

Definition of Autonomy

A service is **autonomous** if it doesn't directly depend on another service to complete its work. It can determine on its own what to do.

Definition of Authority

A service is the **authority** if other services needs to ask it for data or instruct it to perform a task on their behalf for them to complete their job

INPAY

SERVICES ARE AUTONOMOUS

For a service to be autonomous is must **NOT share state**

INPAY

SERVICES ARE AUTONOMOUS

Autonomy is essential for
Scalability (scale out clustering)
Reliability (fail over clustering)

INPAY

SERVICES ARE AUTONOMOUS

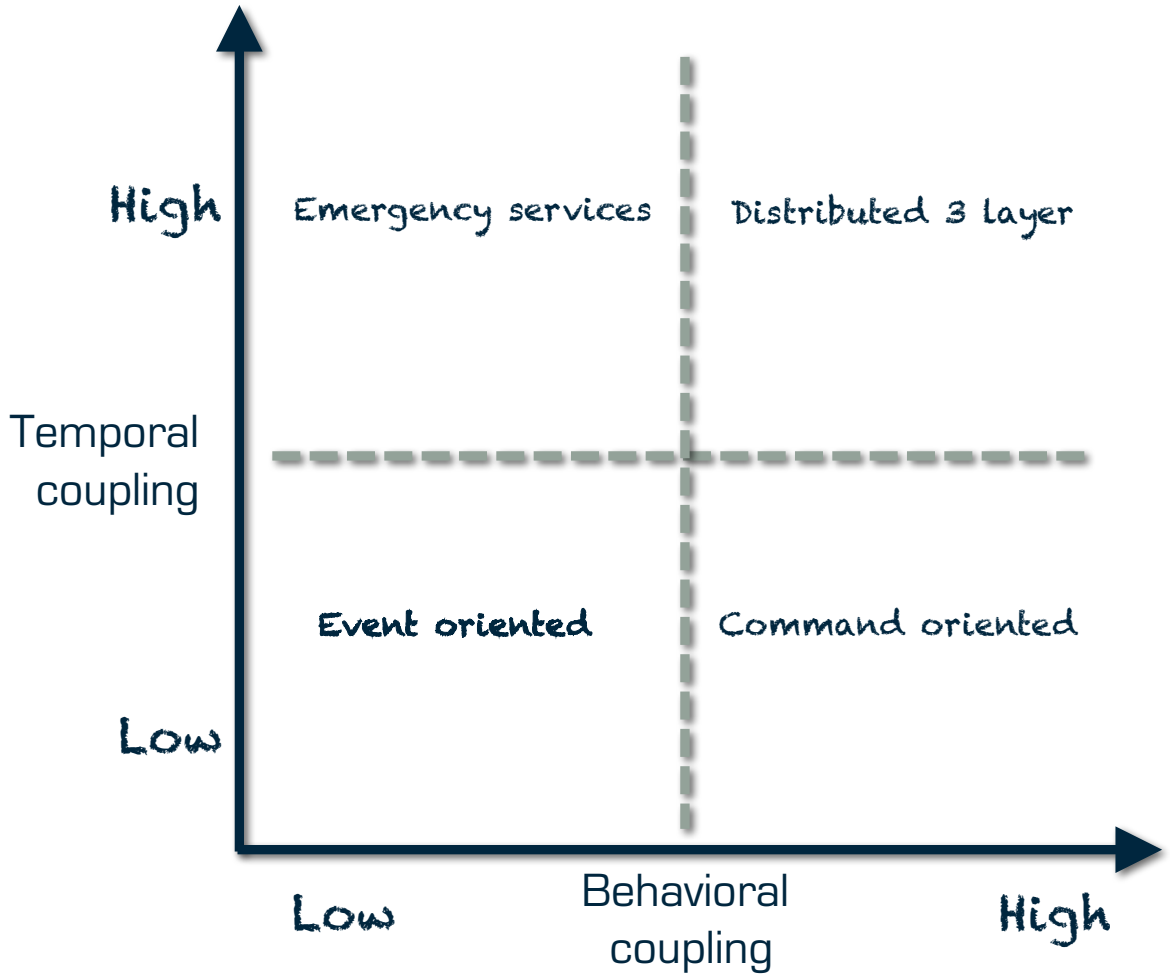
Autonomy is essential for

Reusability

Adaptability

INPAY

Coupling matrix*



* Modified version of Ian Robinson's matrix: http://iansrobinson.com/2009/04/27/temporal-and-behavioural-coupling/

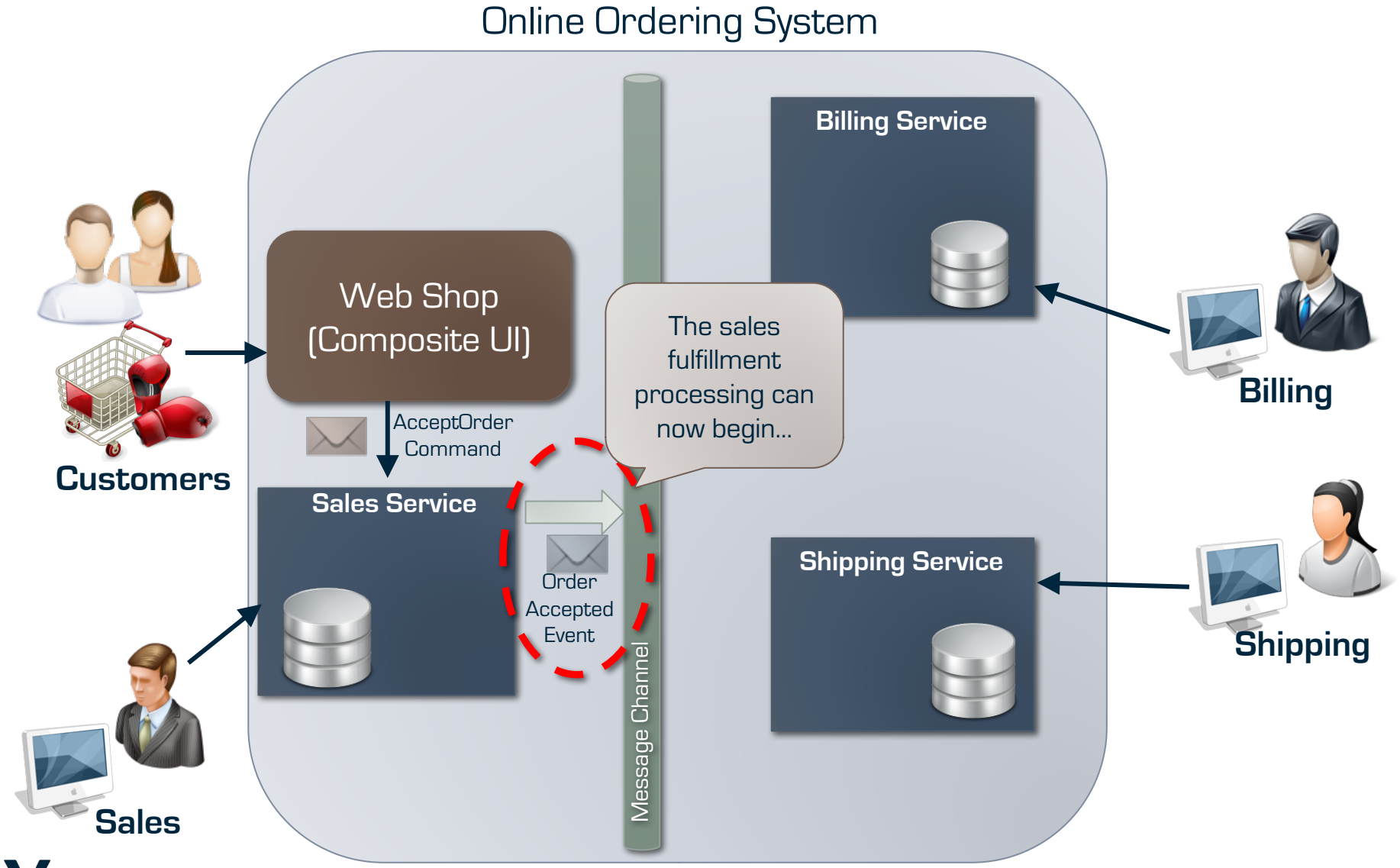
INPAY

WE NEED TO CHANGE FOCUS FROM SHORT TECHNICAL TRANSACTIONS

To long running business transactions supporting business processes

INPAY

Using Business Events to drive Business Processes



INPAY

Events

> *"An Event describes something that HAS happened"*

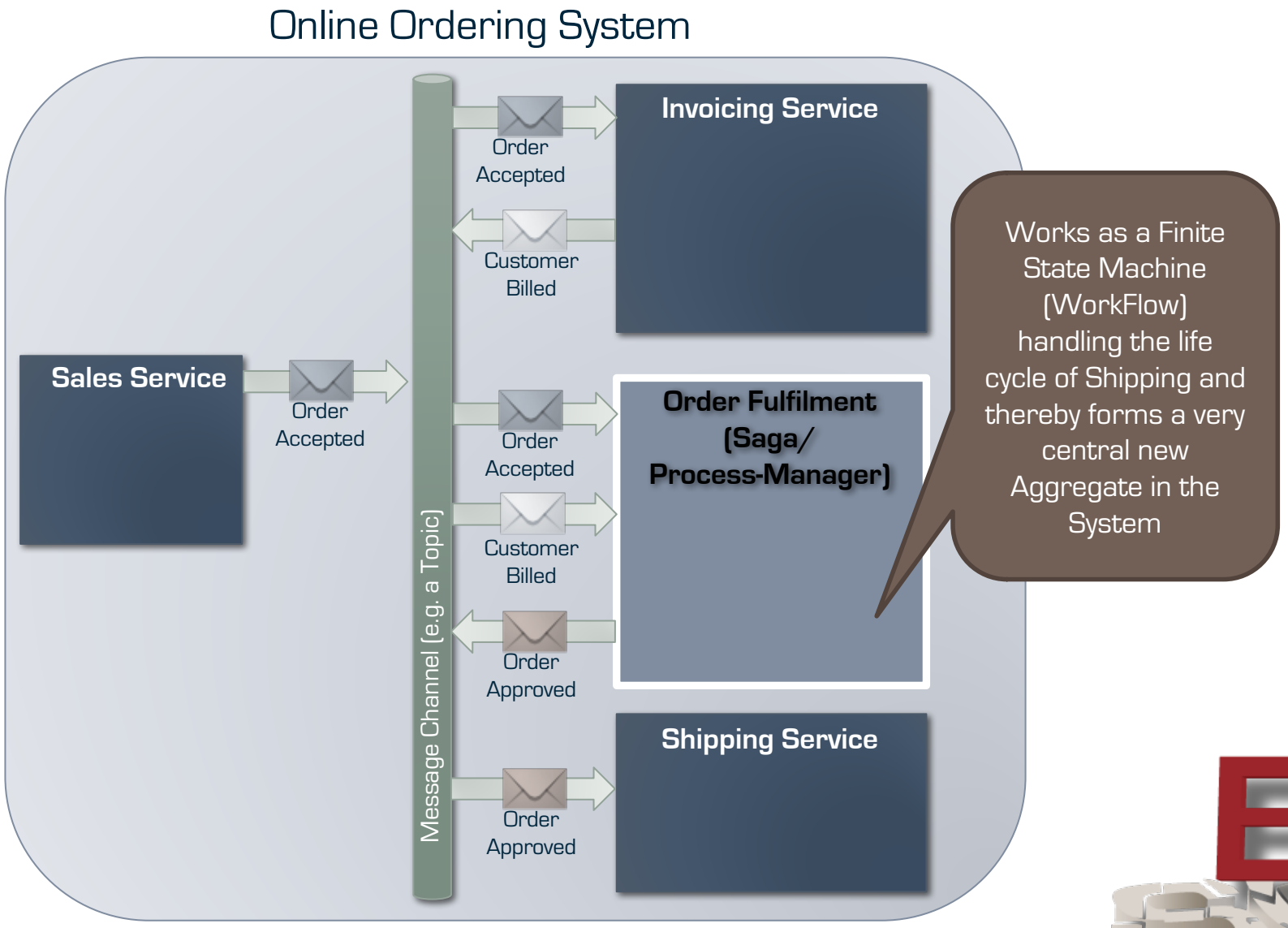
An **Event** is non-prescriptive of what should happen in other parts of the system. It leaves this open to the recipients, so that they themselves determine what to do based on occurrence of the event.

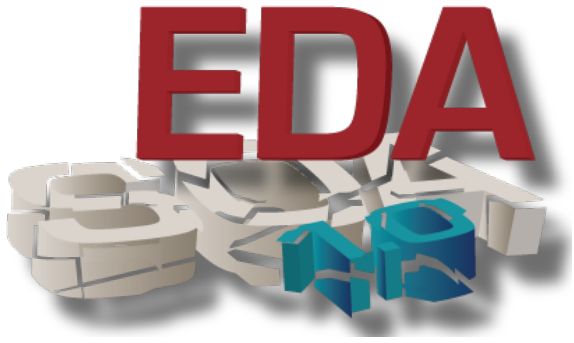
Events always carry a **name** in its **past-tense form**: ***OrderWasAccepted, OrderHasShipped, CustomerWasReimbursed***

Other qualities

- **Immutable**, i.e. content cannot be changed
- Always carries the ID of the Aggregate it relates to
- An event can and will typically will be published to multiple consumers.
  - The publisher of the event does not know who the recipients are
  - And it doesn't know what the recipients intend to do with the event

Choreographed Event Driven Processes



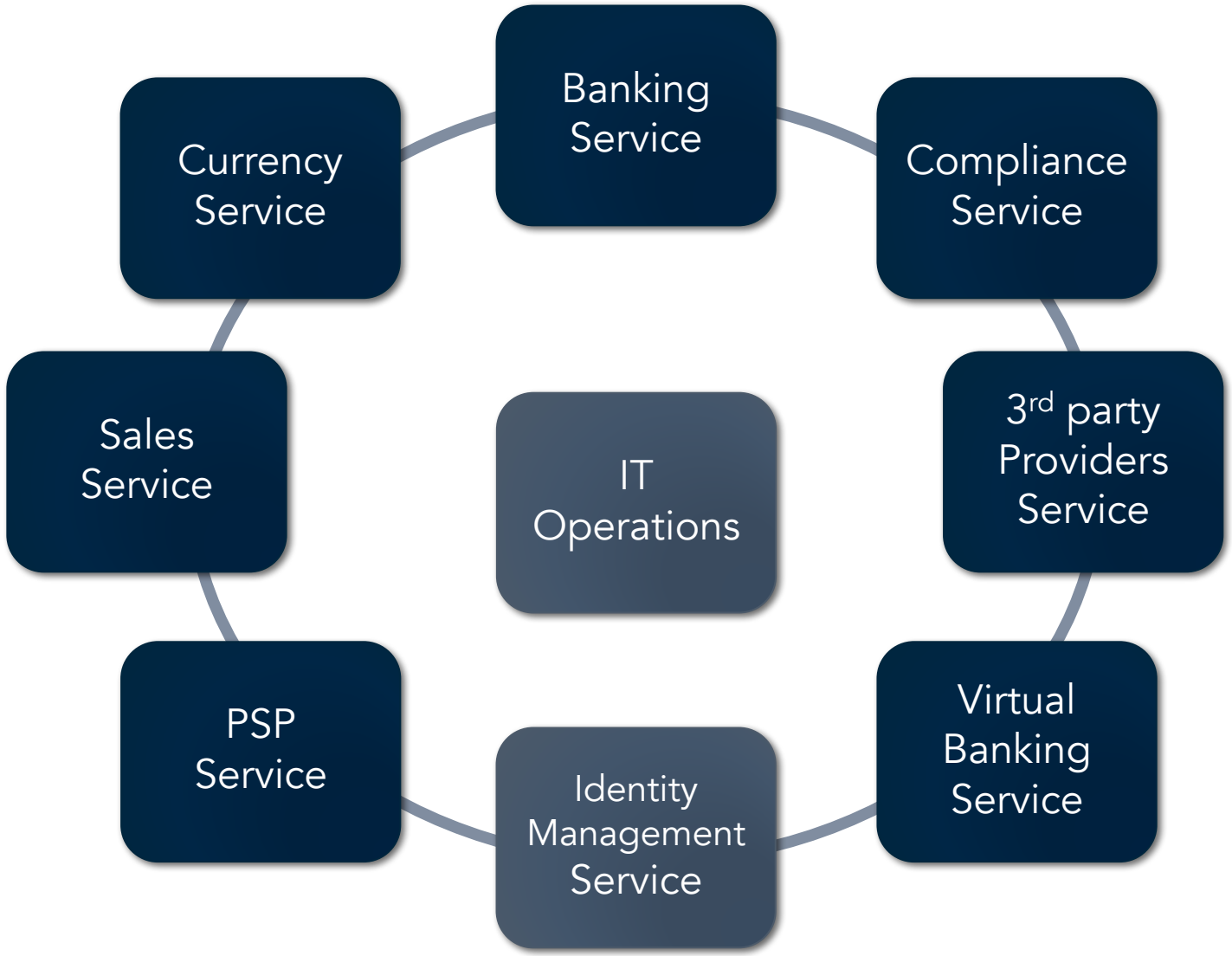


INPAY

SERVICES AT **INPAY**

INPAY

Services/Business-capabilities in **INPAY**



INPAY

Service and deployment

A **service** needs to be **deployed** everywhere its **data** is **needed**

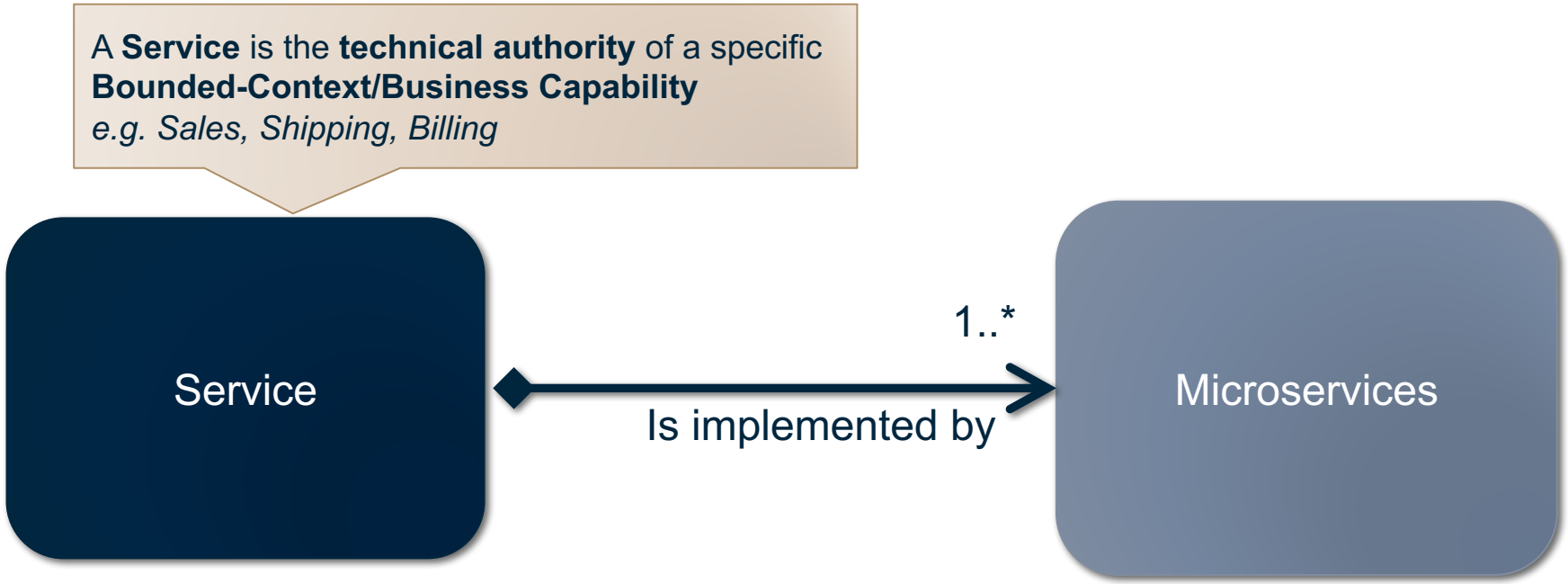
- A **Service** represents a **logical responsibility boundary**
- Logical responsibility and physical deployment of a Service DOES NOT have to be 1-to-1
  - It's too constraining
  - We need more degrees of freedom
  - Philippe Krutchen 4+1 views of architecture: Logical and Physical designs should be independent of each other

INPAY

We need more fine grained building blocks

INPAY

Service vs Microservices



INPAY

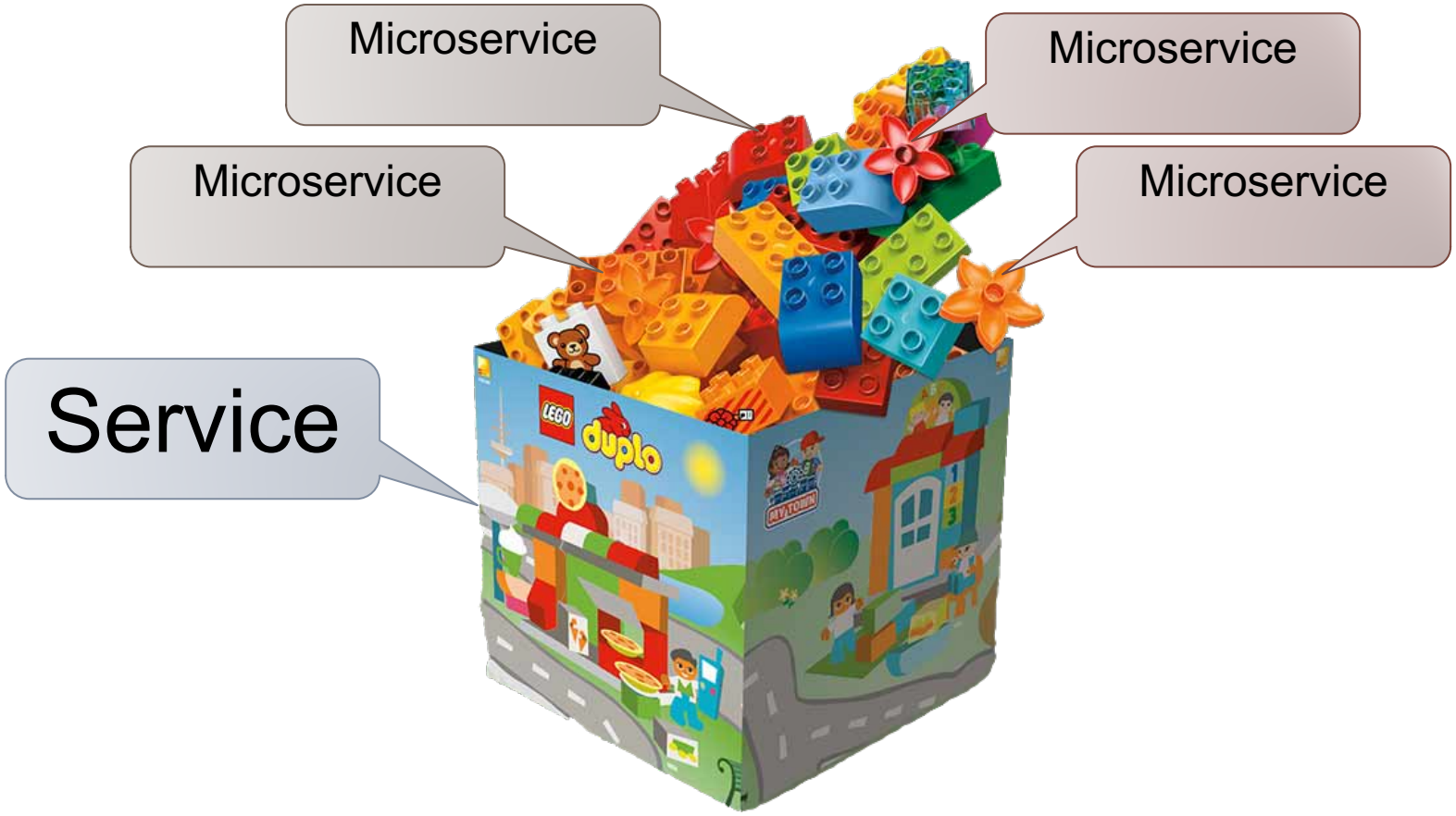
Service vs Microservices

Microservices are a **division** of **Services** along **Transactional boundaries** (a transaction stays within the boundary of a Microservice)

Microservices are the **individually logical deployable units** of a *Service* with their own *Endpoints*. Could e.g. be the split between **Read** and **Write** models (CQRS) - each would be their own Microservices

INPAY

A Service represents a logical boundary



INPAY

Services are the corner stone

- We talk in terms of Services capabilities and the processes/use-cases they support
- Microservices are an implementation detail
  - They are much less stable (which is a good thing – it **means they're easier to replace**)

INPAY

Microservices is an architectural style

INPAY

Service deployment

- Many services can be deployed to the same physical server
- **Many services can be deployed in the same application**
  - Application boundary is a Process boundary which is a **physical** boundary
  - A Service is a **logical** boundary
- Service deployment is not restricted to tiers either
  - Part of service A and B can be deployed to the Web tier
  - Another part of Service A and B can be deployed to the backend/app-service tier of the same application
- The **same service** can be **deployed** to multiple **tiers** / multiple **applications**
  - ie. **applications and services are not the same and does not share the same boundaries**
- **Multiple services** can be "deployed" to the same **UI page** (service mashup)

Be pragmatic

There’s cost in deploying 1000’s of microservices

INPAY

Autonomous Components

Not everything needs to be individually deployable

INPAY

Autonomous-components are logical deployable units

This means they CAN, but they don't HAVE to be deployed individually.

Design for Distribution
But take advantage of locality

INPAY

Be even more pragmatic

Some services are very stable

In which case we allow other services
to call them using **local calls**

INPAY

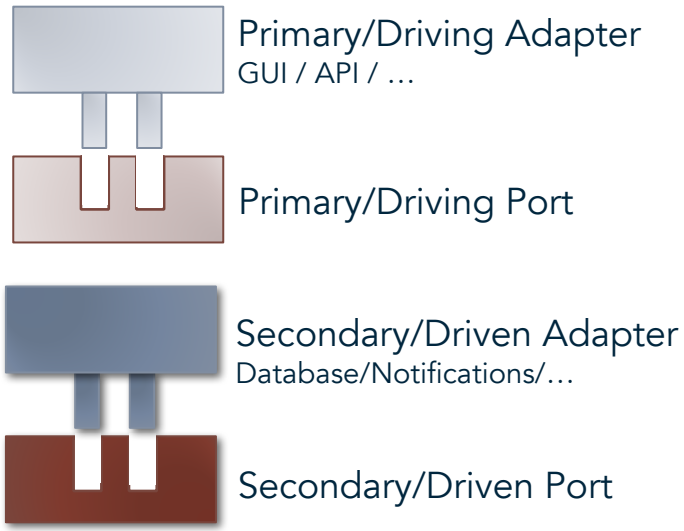
AC in code

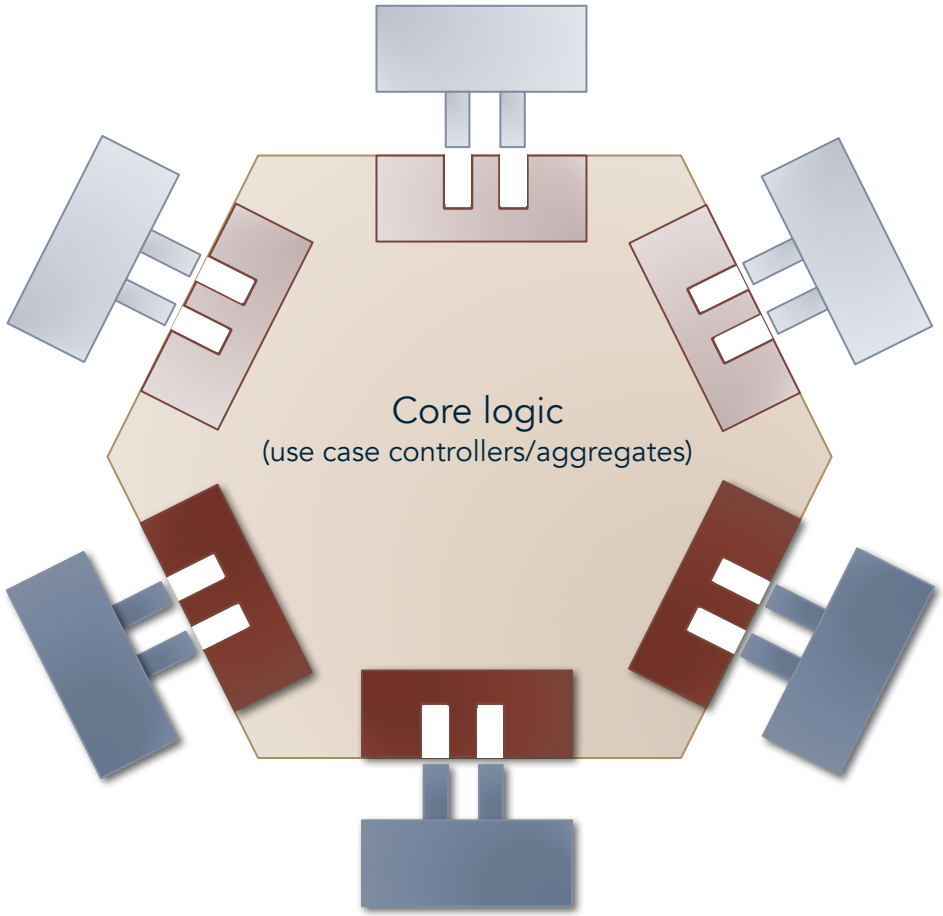
```
public class PSPAgreementAc extends HzBackedAutonomousComponent {
    public static AutonomousComponentId SERVICE_AC_ID = PSP_SERVICE_ID.ac("psp_agreement_ac");
    …

    public PSPAgreementAc(CurrencyConverter currencyConverter) {
        this.currencyConverter = currencyConverter;
    }

    @Override
    public void onInitialize(IConfigureACEnvironment acSetup) {
        acSetup.withAutonomousComponentId(SERVICE_AC_ID)
                .usingServiceDataSource()
                .withBusConfiguration(cfg -> {
                    bus.registerAxonReplayableTopicPublisher(InternalTemplateEvents.TOPIC_NAME,
                                                             replayFromAggregate(PSPTemplate.class)
                                                                 .dispatchAggregateEventsOfType(InternalTemplateEvents.class));

                    bus.subscribeTopic(SERVICE_AC_ID.topicSubscriber("ContractEvents"),
                                       ExternalContractEvents.TOPIC_NAME,
                                       new SalesTopicSubscription(bus));
                })
                .runOnBusStartup((bus, axonContext) -> {
                });
    }
}
```

INPAY

Autonomous Component

- Can be deployed alone or co-located, together with one or more **adapters** from the same service
- Works transparently in a clustered environment

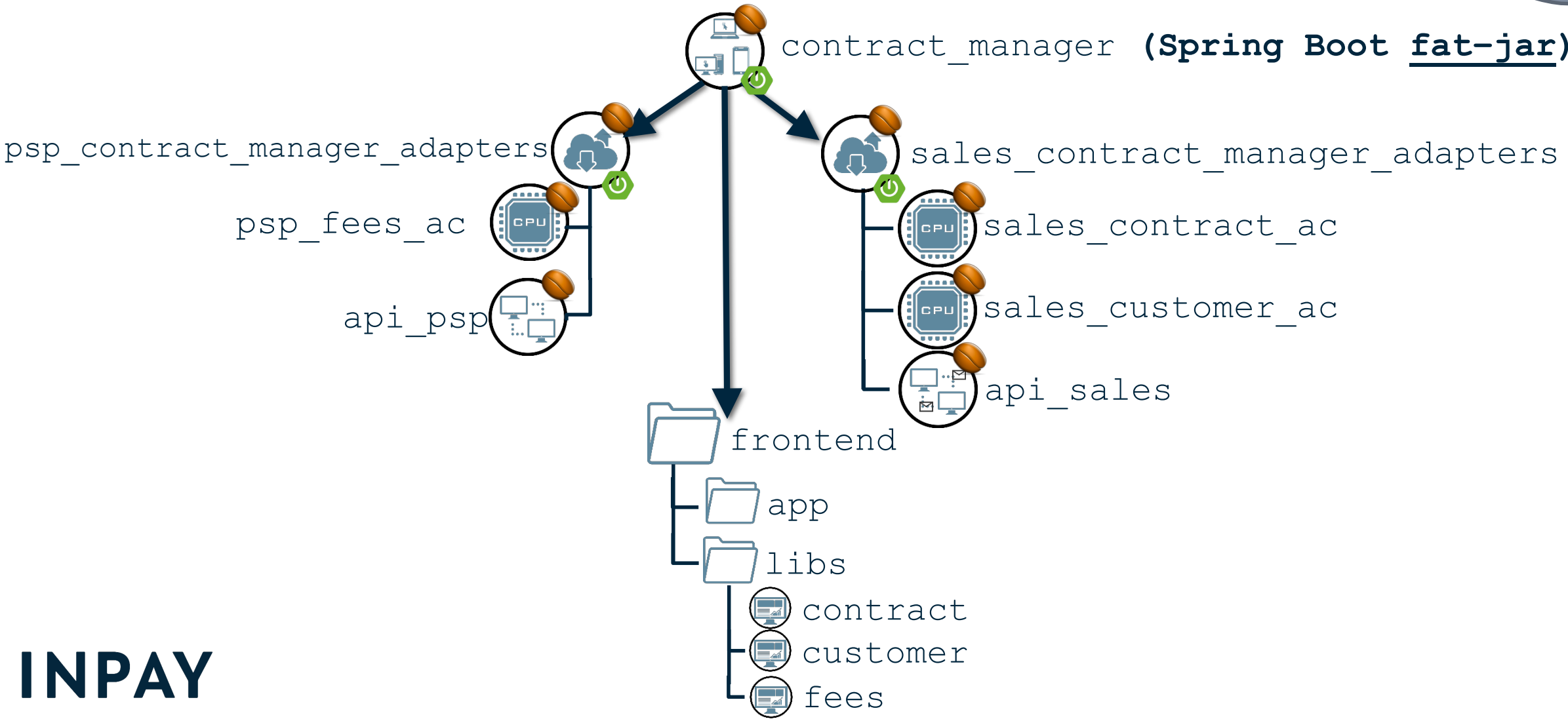




INPAY

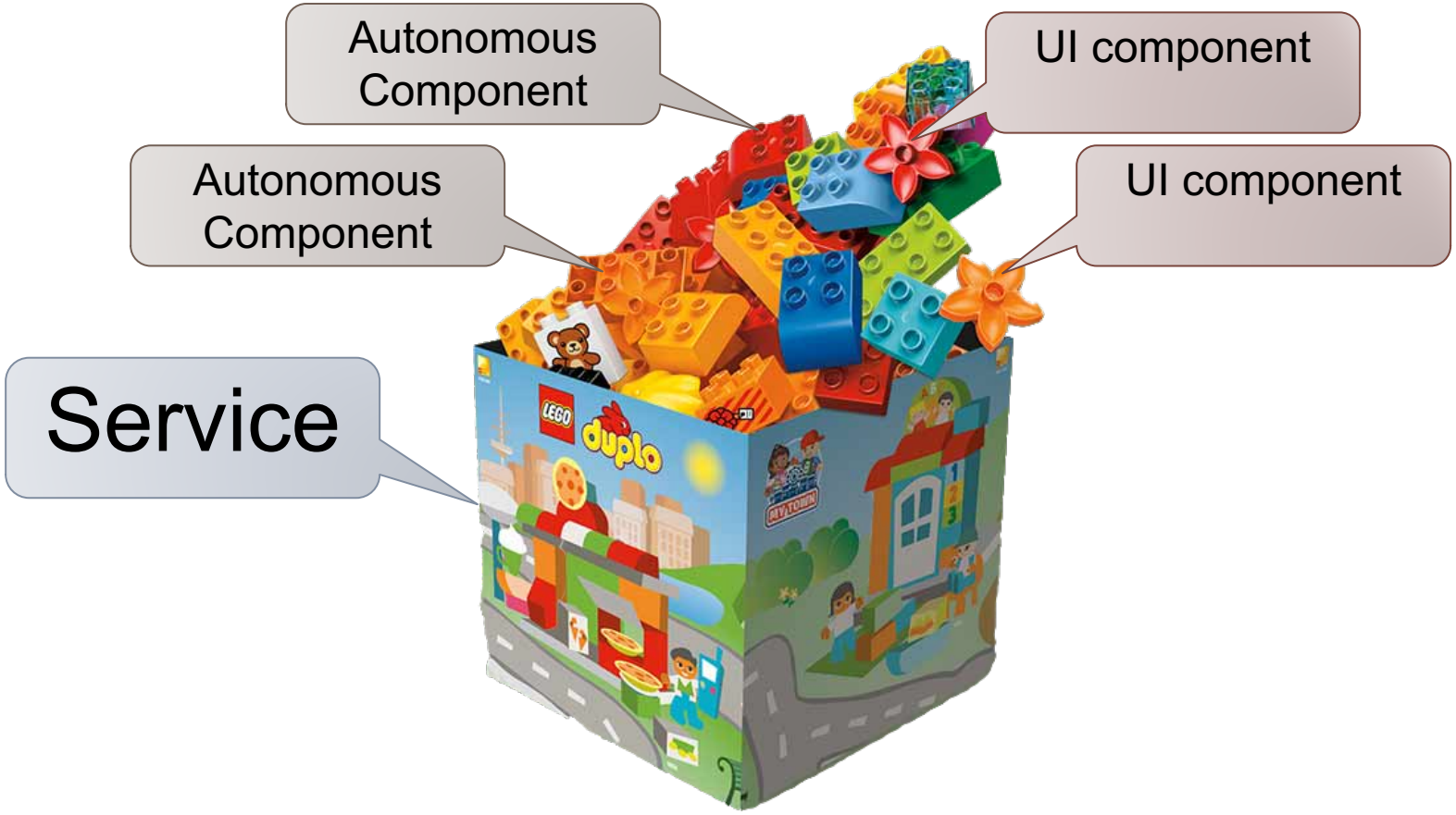
Autonomous Components **can** be co-deployed together with Appli backends

Docker is NOT a requirement - KISS



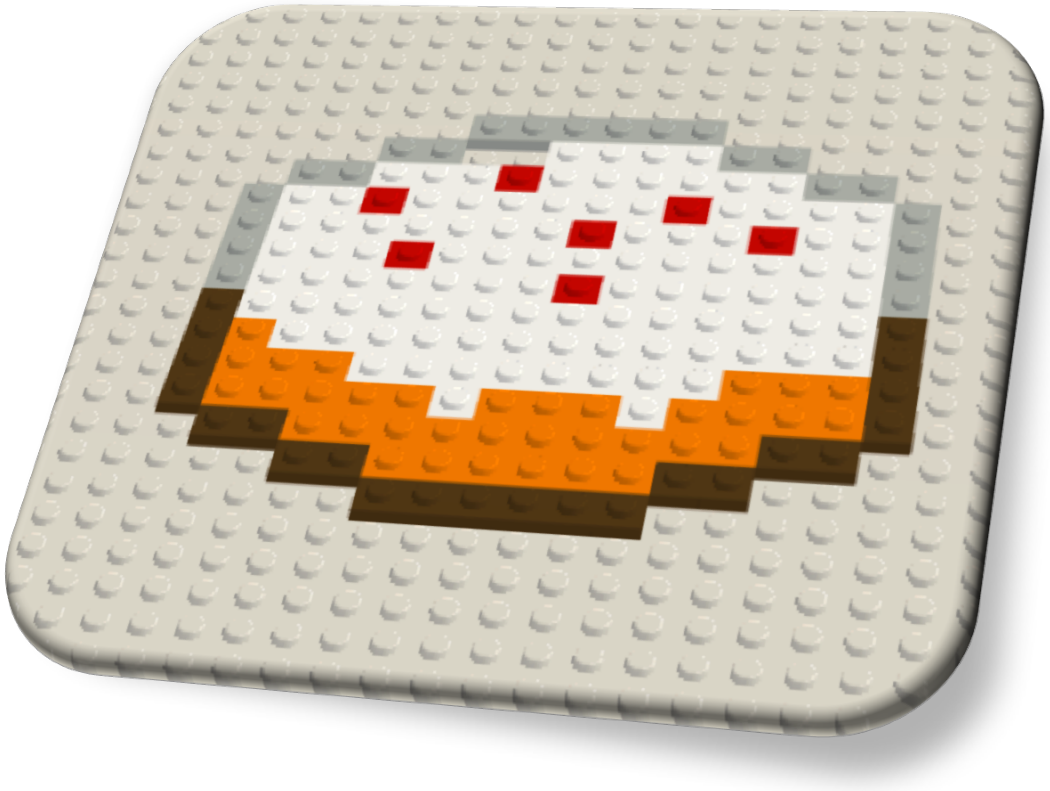
INPAY

A Service represents a logical boundary



INPAY

An Application is the plate where Components are co-deployed



Sales service components

PSP service components

…

INPAY

Applications in **INPAY**

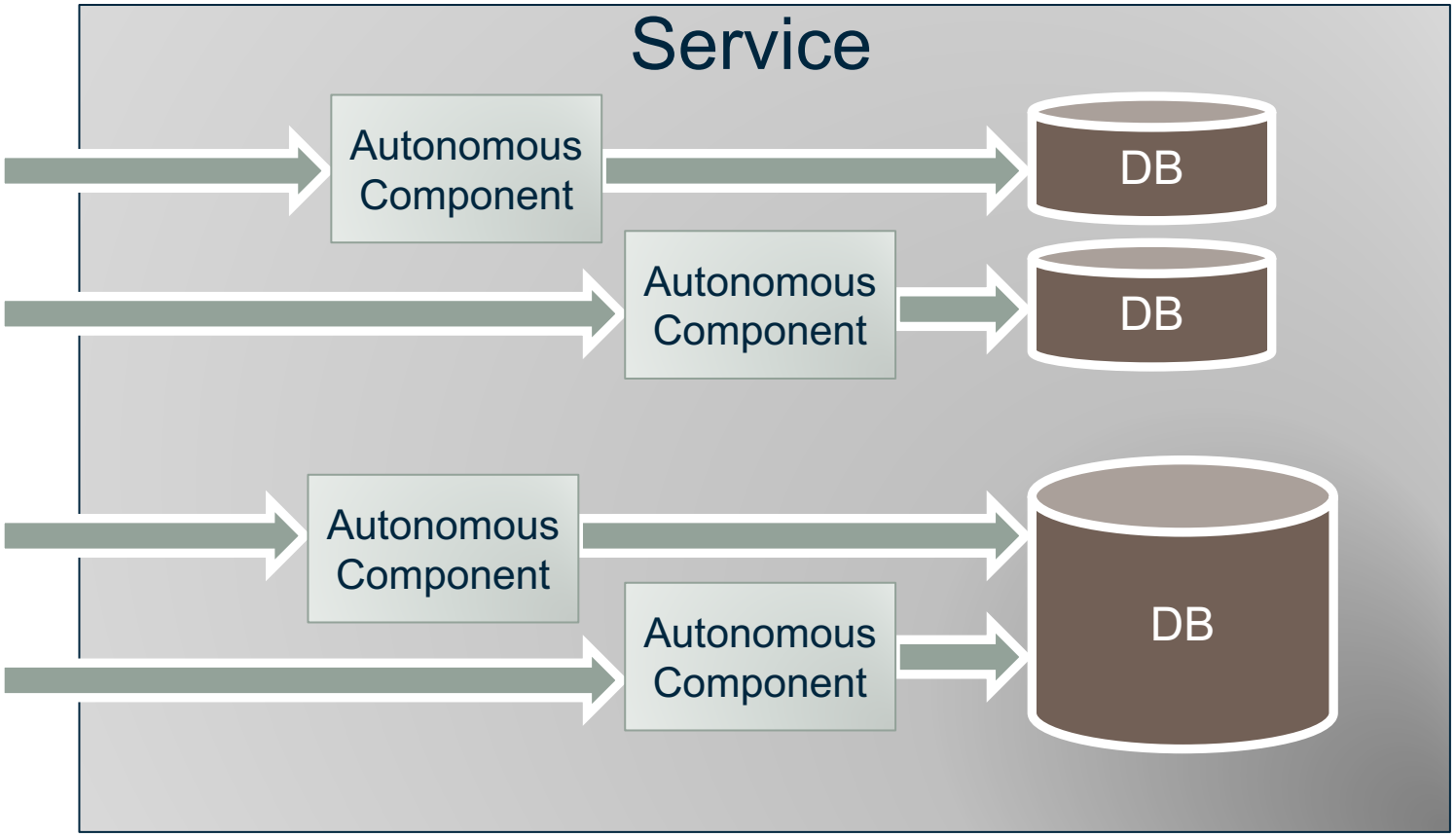
INPAY

Application in code

```
@Configuration
@ComponentScan(basePackages = { "com.inpay.contractmanager",
                                "com.inpay.adapters",
                                "com.inpay.itops.spring" })
public class Application extends InpaySpringBootApplication {
    @Override
    protected ApplicationIdentifier getApplicationIdentifier() {
        return ApplicationIdentifier.from("ContractManager");
    }

    @Override
    protected Collection<AutonomousComponent> getAutonomousComponentsHostedInThisApplication() {
        CurrencyExchangeRateAc currencyExchangeRateAc = new CurrencyExchangeRateAc();
        return list(
            new PSPFeeScheduleAc(currencyExchangeRateAc.getCurrencyConverter()),
            new VBFeeScheduleAc(currencyExchangeRateAc.getCurrencyConverter()),
            new ContractAc(),
            new CustomersAc(),
            currencyExchangeRateAc
        );
    }


    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

INPAY

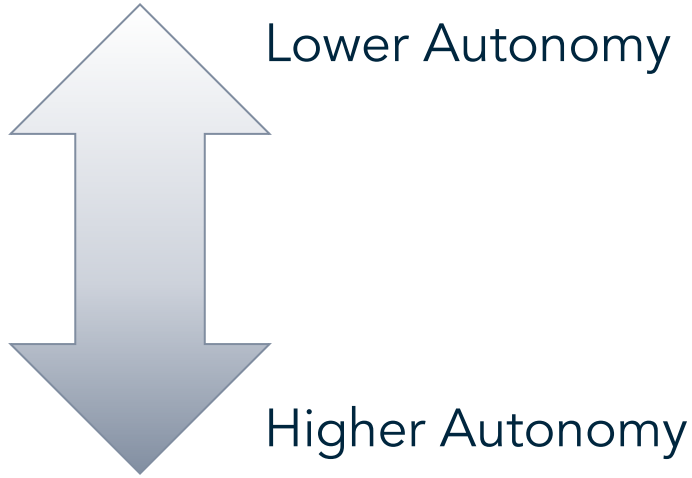
AC, autonomy and “shared” service data



INPAY

50 shades of inter service AC Autonomy*

Endpoint	Process	Database	Storage
Shared	Shared	Shared	Shared
Own	Shared	Shared	Shared
Own	Own	Shared	Shared
Own	Shared	Own	Shared
Own	Own	Own	Shared
Own	Own	Own	Own

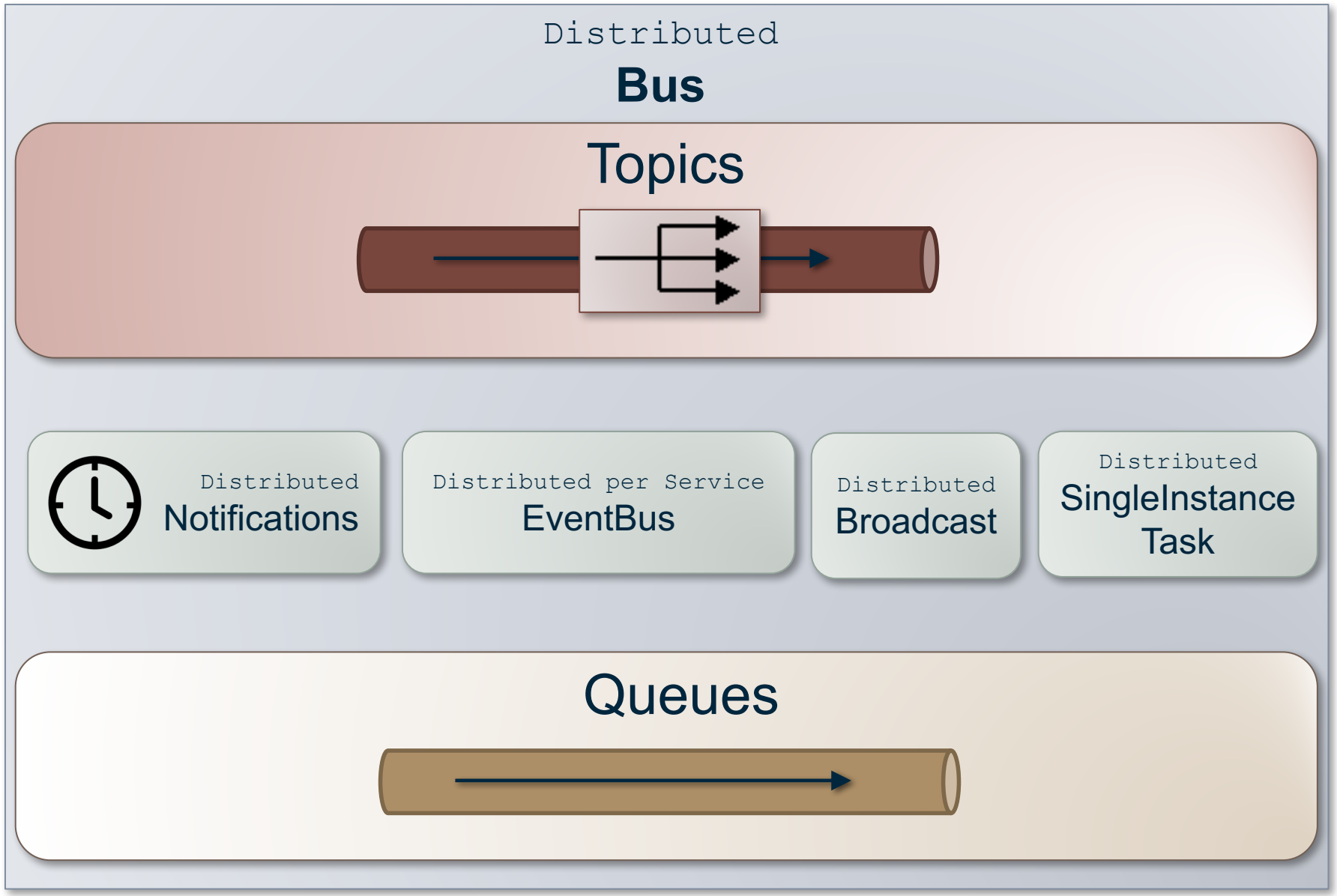


* No RPC in use!

INPAY

Infrastructure patterns

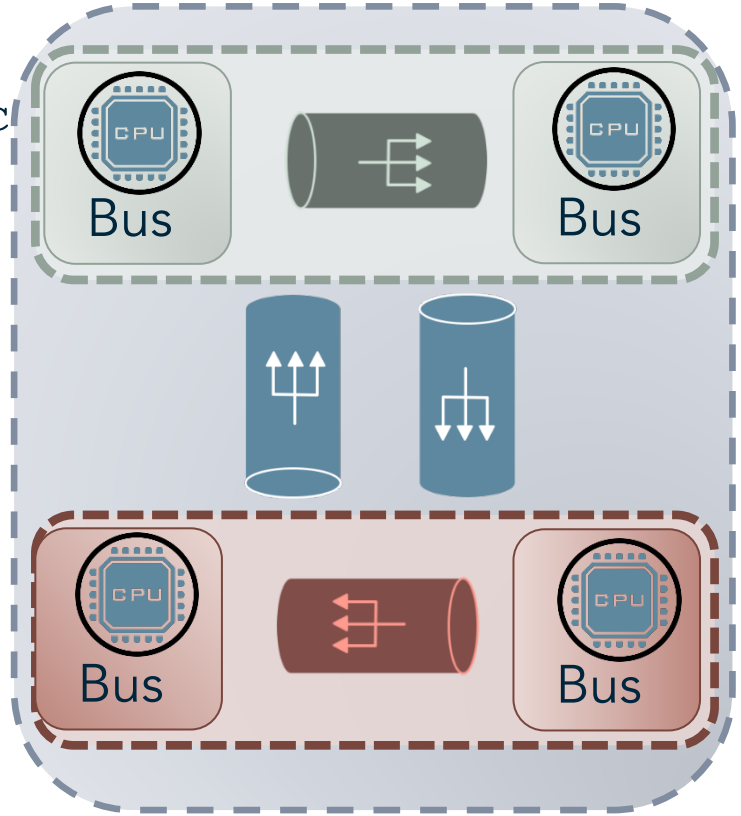
INPAY


Distributed
Bus
Topics
Distributed
Notifications
Distributed per Service
EventBus
Distributed
Broadcast
Distributed
SingleInstance
Task
Queues

INPAY

Federated Bus

psp_fees_ac
(deployed on 10.25.26.101)

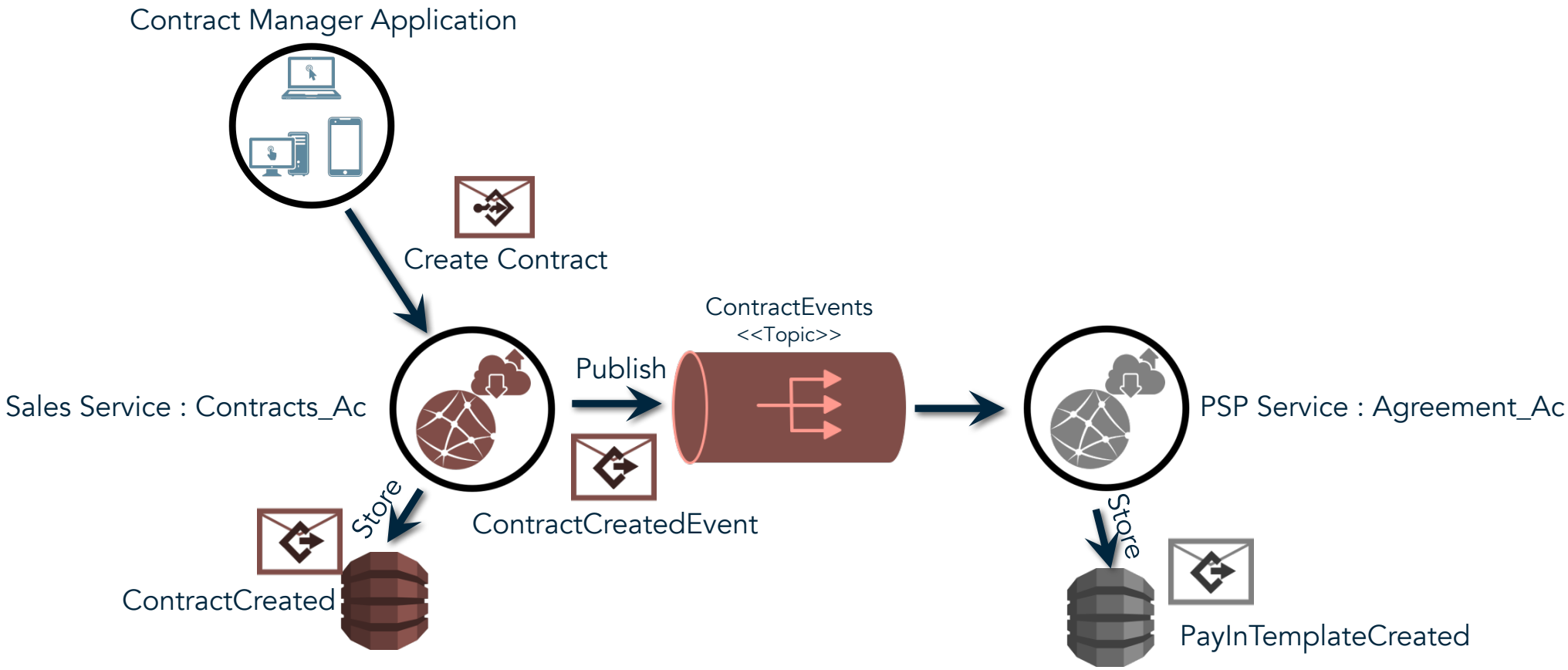


psp_fees_ac
(deployed on 10.25.26.102)

sales_contract_ac
(deployed on 10.25.26.103)

sales_contract_ac
(deployed on 10.25.26.104)

INPAY

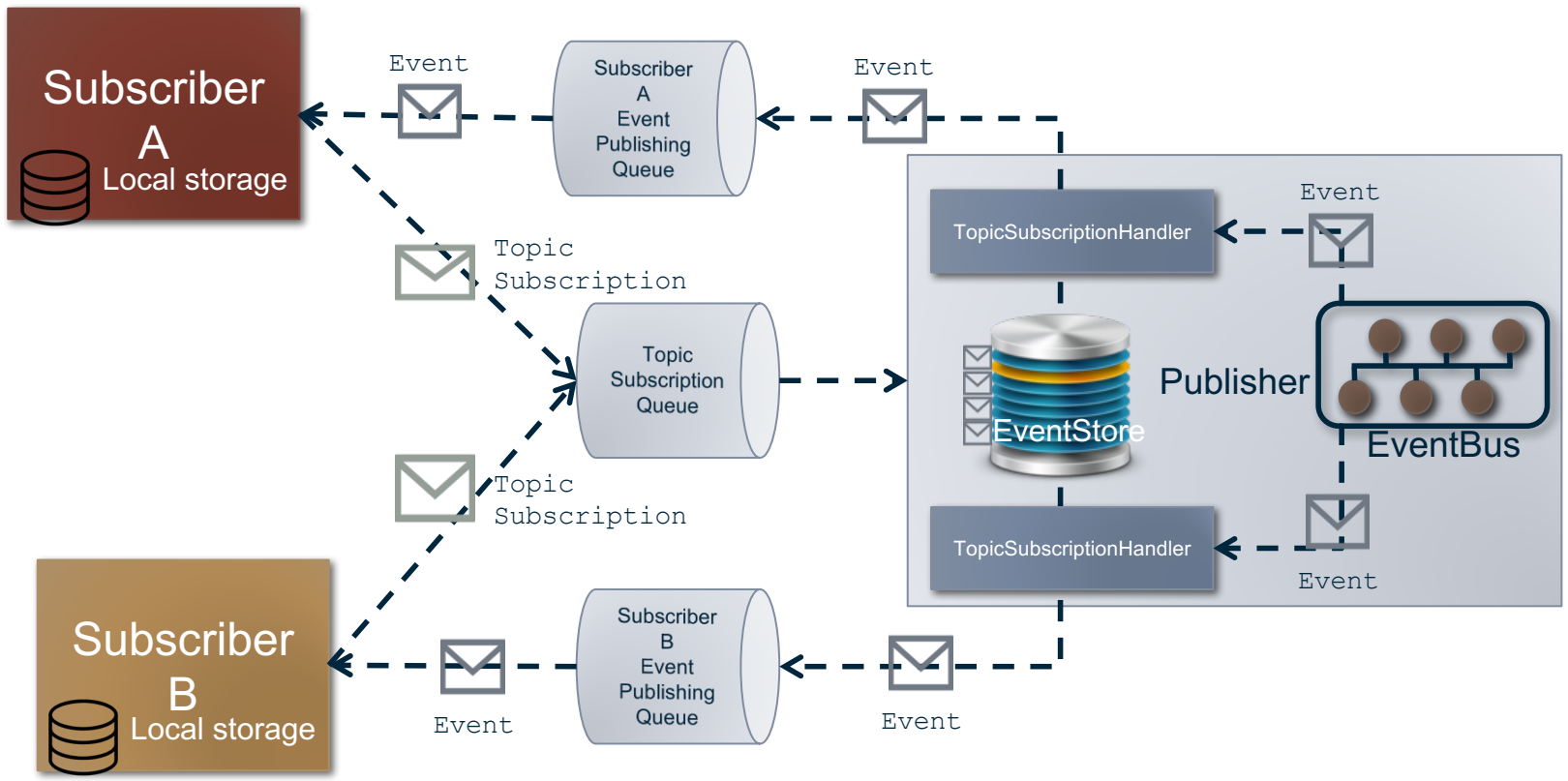

Contract Manager Application
Create Contract
ContractEvents
<<Topic>>
Publish
Sales Service : Contracts_Ac
PSP Service : Agreement_Ac
Store
ContractCreatedEvent
Store
ContractCreated
PayInTemplateCreated

INPAY

Client handled subscriptions

- Highly resilient pattern for an Event Driven Architecture that's backed by Event-Sourced AC's
- In this model the **publisher** of the Events is **responsible** for the **durability** of all its **Events**, typically to an EventStore/EventLog.
- Each **client** (**subscriber**) **maintains durable information** of the **last event** it has **received** from each **publisher**.
- When ever the **client** starts up it makes a **subscription** to the **publisher** where it **states** from which **point in time** it wants **events published/streamed** to it.
- This effectively means that publisher can remain simple and the client (subscriber) can remain simple and we don't need additional sophisticated broker infrastructure such as Kafka+ZooKeeper.

Client handled subscriptions



INPAY

5 Microservices **Do's**

- **Identify Business Capabilities (or Bounded Contexts) and split your services according to them**. A service is owned by one team that builds and runs the service. This gives you proper business and IT alignment and allows pin point accuracy with regards to spending money to solve problems.
- **Spend time to understand the business processes and the domain. At first you must go slow to go fast.** Building microservices properly takes time and is not trivial. Identify how likely things are to change and what things change together.
- **Focus on the business side effects - also know as the Events and make them a first class principle**. Avoid RPC/REST/Request-Response between Services - events are the API.
- **Consider building composite application and Backend's For Frontend's (BFF's) to decouple services further**. An application is owned by a dedicated team, but may borrow developers from service teams.
- **Learn from history. Don't repeat the mistakes that gave (misapplied) SOA a bad name**. Also, microservice might not be as small as you think - we need low coupling as well as high cohesion :)

INPAY

5 Microservices **Don’ts**

- **Do not introduce a network boundary as an excuse to write better code** - many have troubles with poorly modularized monoliths and believe that introducing network between modules magically solves the problem. If you don’t change your thinking and design, you will end up with a distributed monolith, which has all the disadvantages of a monolith, the disadvantages of a distributed system and none of the advantages of microservices
- **Don't split the atom! Distributed Transactions are never easy**. **Learn about the CAP theorem and avoid Request/Response API’s between Services**.
- Don't fall into the trap of **“Not my problem”**. When working on isolated code bases teams can loose sight of the big picture.
- **Identify the bottlenecks and possible solutions before deciding to split a problem into one or more microservices.** There’s nothing the guarantees that your microservice scales better than your monolith.
- **Don't do big bang rewrites**. Move towards microservices gradually while focusing on functional areas that can replace or support the old monolith. Don’t rewrite core business while being new to microservices.

INPAY

Thanks :)

INPAY

Jeppe Cramon - @jeppec