Practical Messaging

- A 101 guide to messaging
- Ian Cooper
- X and Hachyderm: ICooper

Who are you?

www.linkedin.com/in/ian-cooper-2b059b

I am a polyglot coding architect with over 20 years of experience delivering solutions in government, healthcare, and finance and ecommerce. During that time I have worked for the DTI, Reuters, Sungard, Misys, Beazley, Huddle and Just Eat Takeaway delivering everything from bespoke enterprise solutions, 'shrink-wrapped' products for thousands of customers, to SaaS applications for hundreds of thousands of customers.

I am an experienced systems architect with a strong knowledge of OO, TDD/BDD, DDD, EDA, CQRS/ES, REST, Messaging, Design Patterns, Architectural Styles, ATAM, and Agile Engineering Practices

I am frequent contributor to OSS, and I am the owner of: https://github.com/BrighterCommand. I speak regularly at user groups and conferences around the world on architecture and software craftsmanship. I run public workshops teaching messaging, event-driven and reactive architectures.

I have a strong background in C#. I spent years in the C++ trenches. I dabble in Go, Java, JavaScript and Python.

Day One Messaging

Distribution

Integration Styles

Messaging Patterns

Queues and Streams

Managing Asynchronous Architectures

Day Two Conversations

Conversations

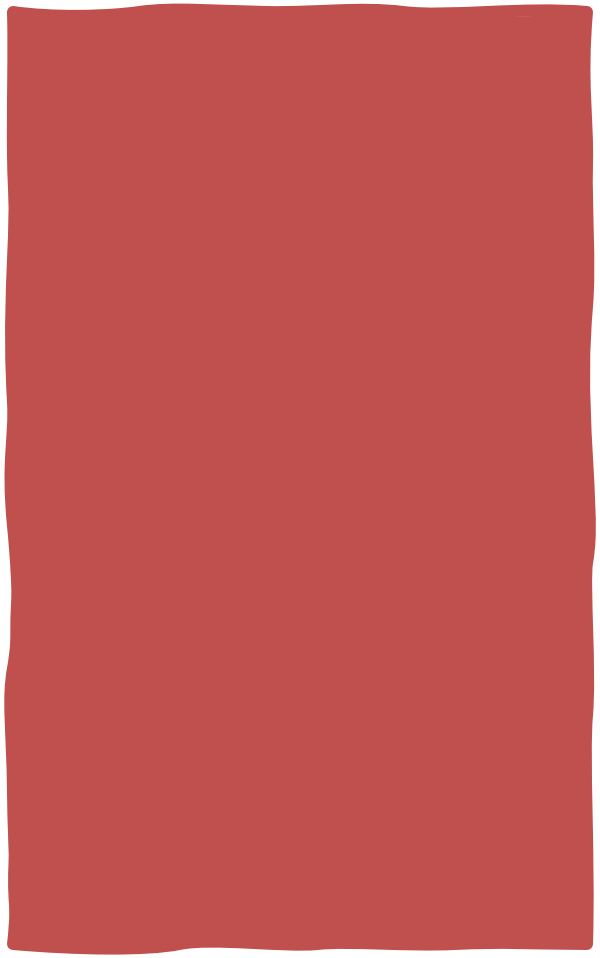
Flow

Process Automation

Orchestration and Choreography

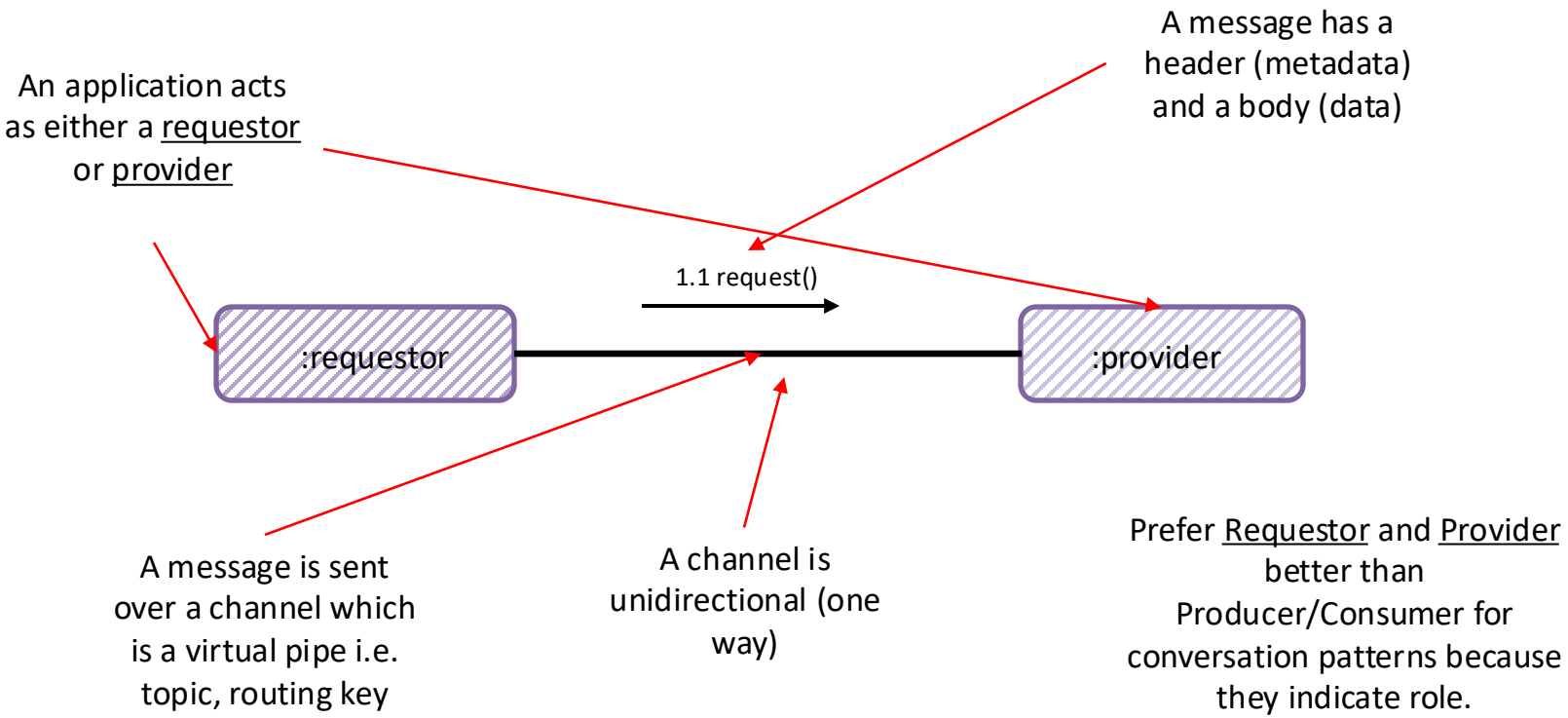
Day Two

Conversations

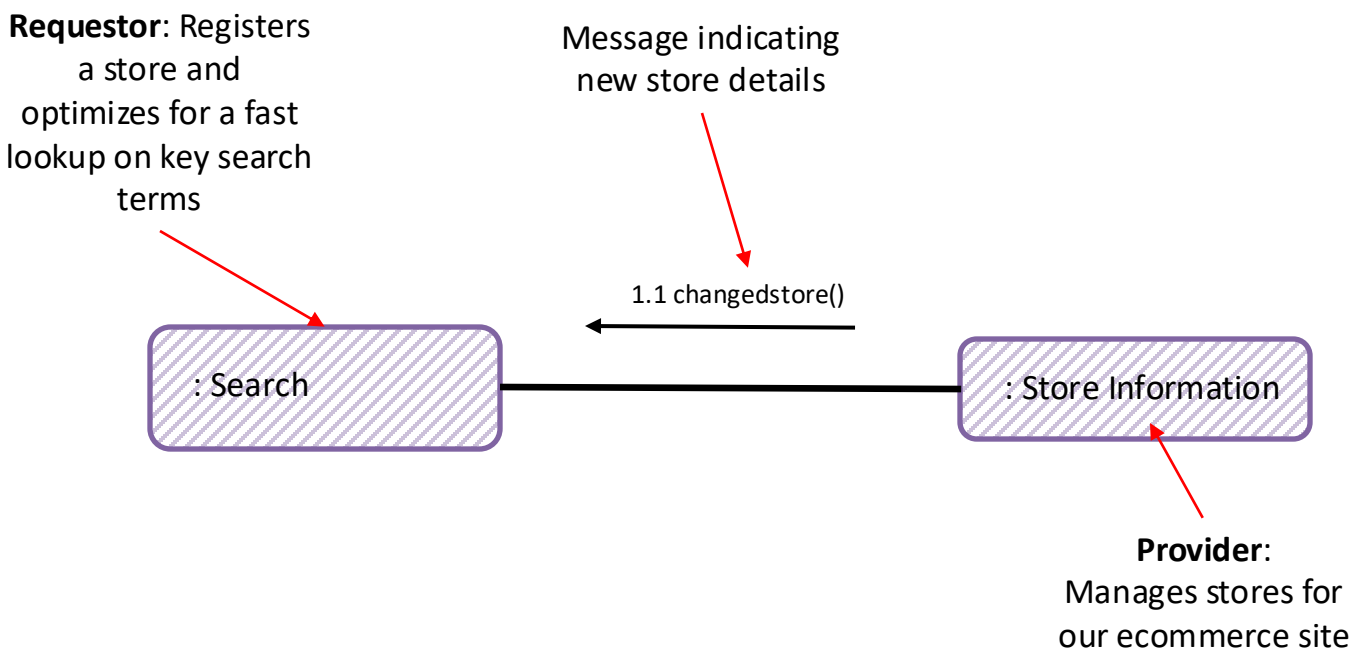


In & Out

Messaging Participants



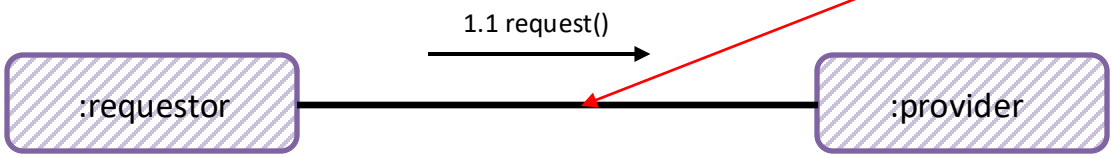
Messaging Participants



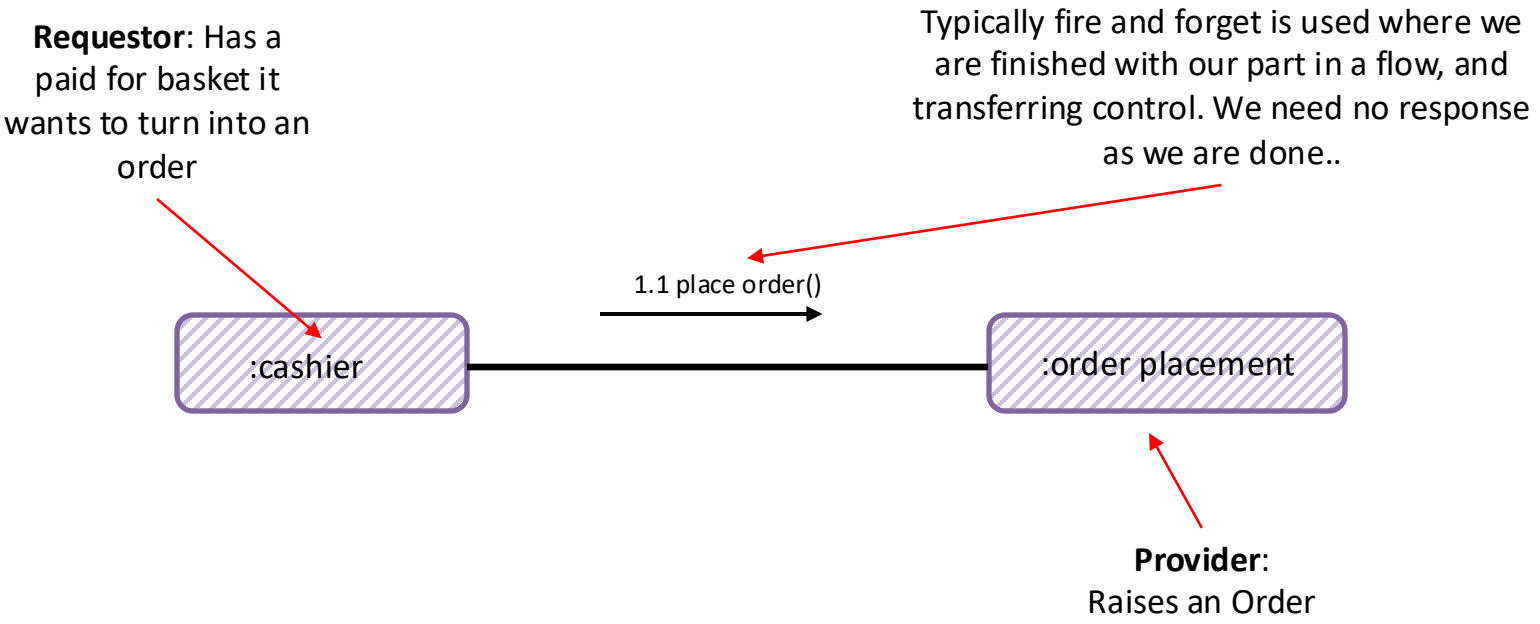
In-Only (Fire and Forget)

Typically we call this **fire-and-forget**.

Under a In-Only pattern the requestor sends a request to the provider, but does not seek an acknowledgment of completion of the requested operation



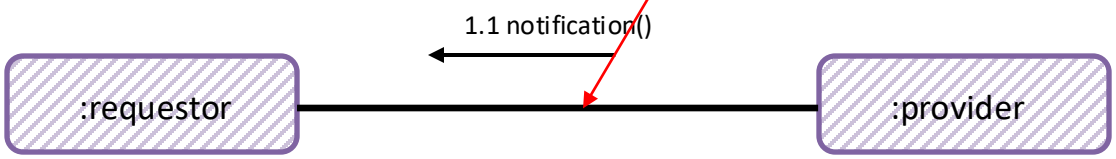
In-Only (Fire and Forget)



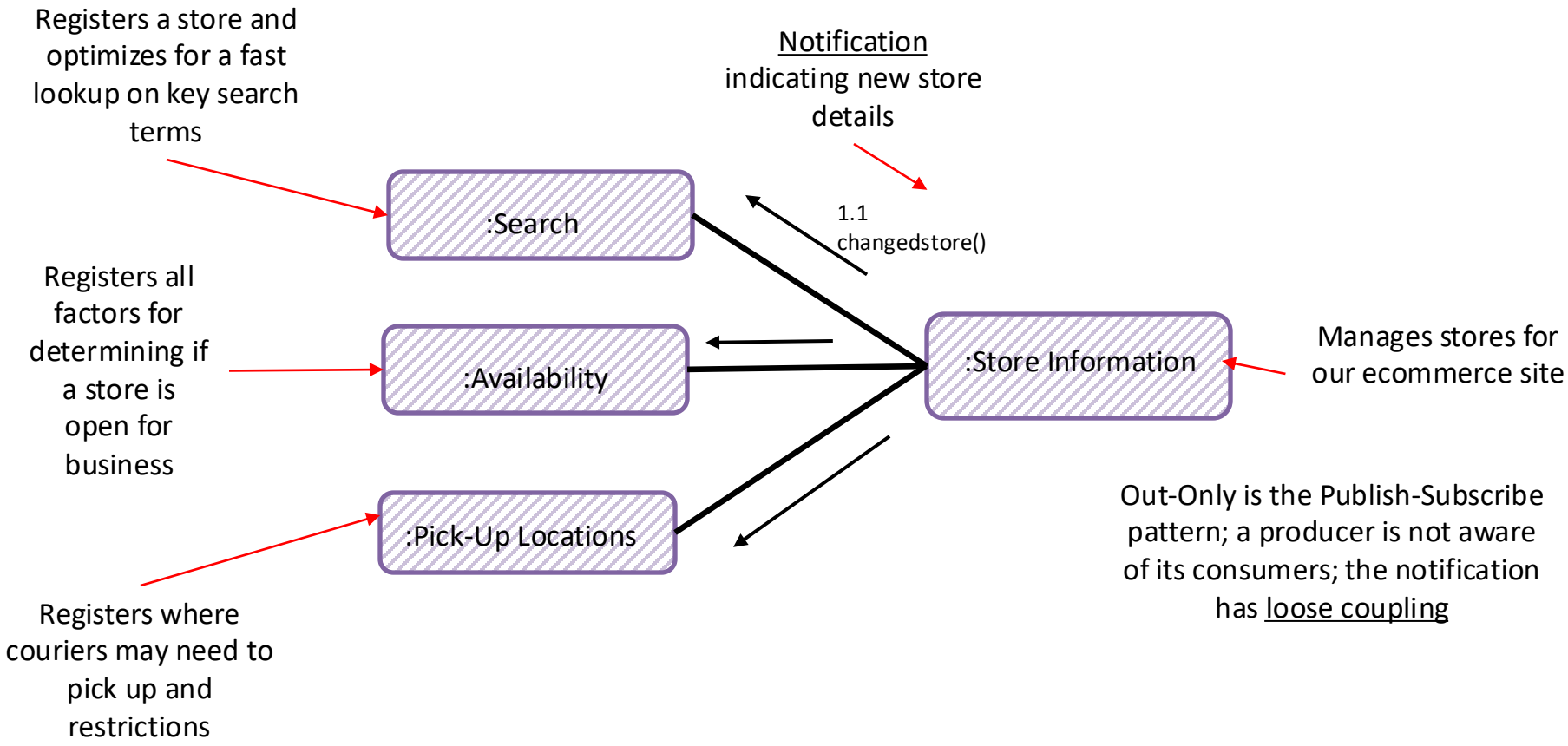
Out-Only (Notification)

Typically we call this a **notification**.

Under an Out-Only pattern the requestor subscribes to the provider, and an operation is triggered by receipt of a message but it does not acknowledge the message to the provider.

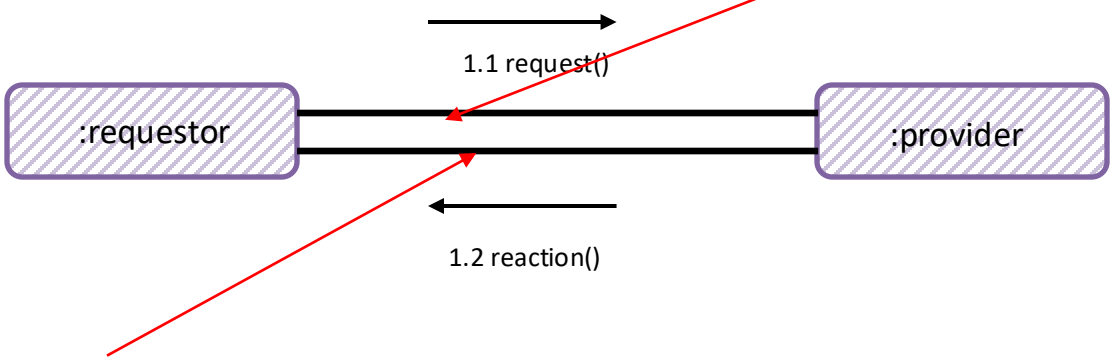


Out-Only (Publish-Subscribe)



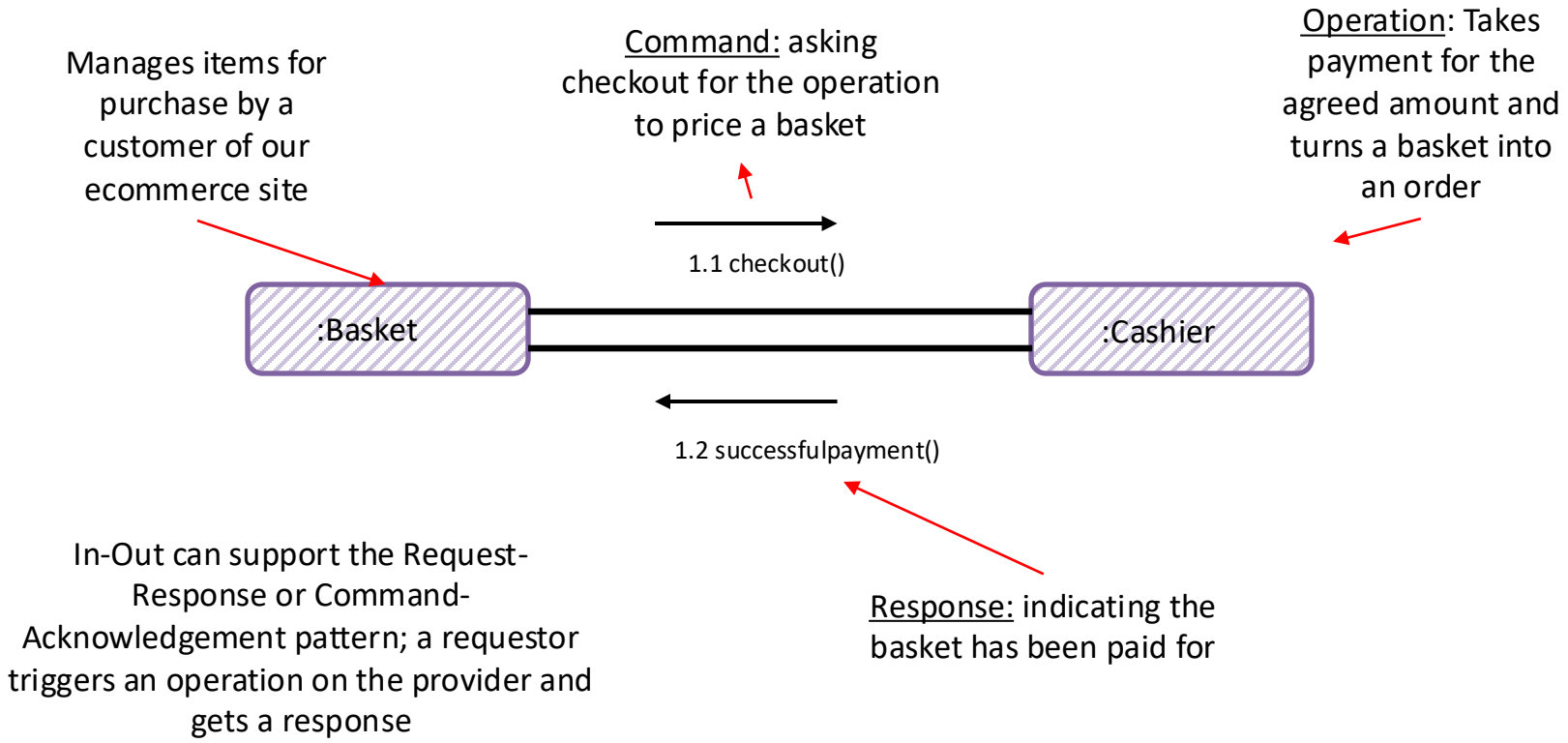
In-Out (Request-Reaction)

Under an In-Out pattern the provider receives a request from the requestor on one channel, and returns a response from the operation triggered by that message on a separate channel.

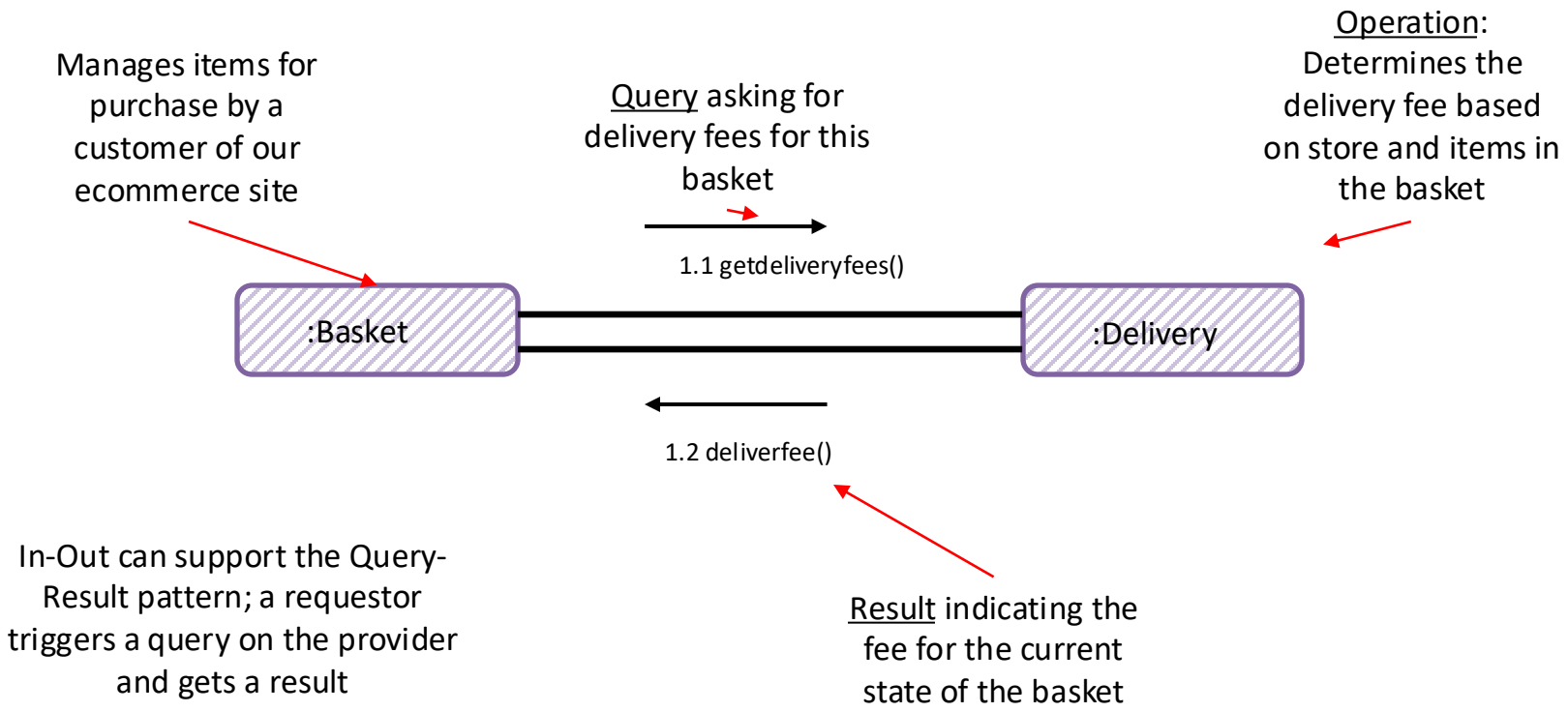


When using an In-Out pattern the provider sets a **correlation id** (conversation id) in the header of the request and the provider returns that id in the header of the response so that we can correlate replies sent over a separate channel.

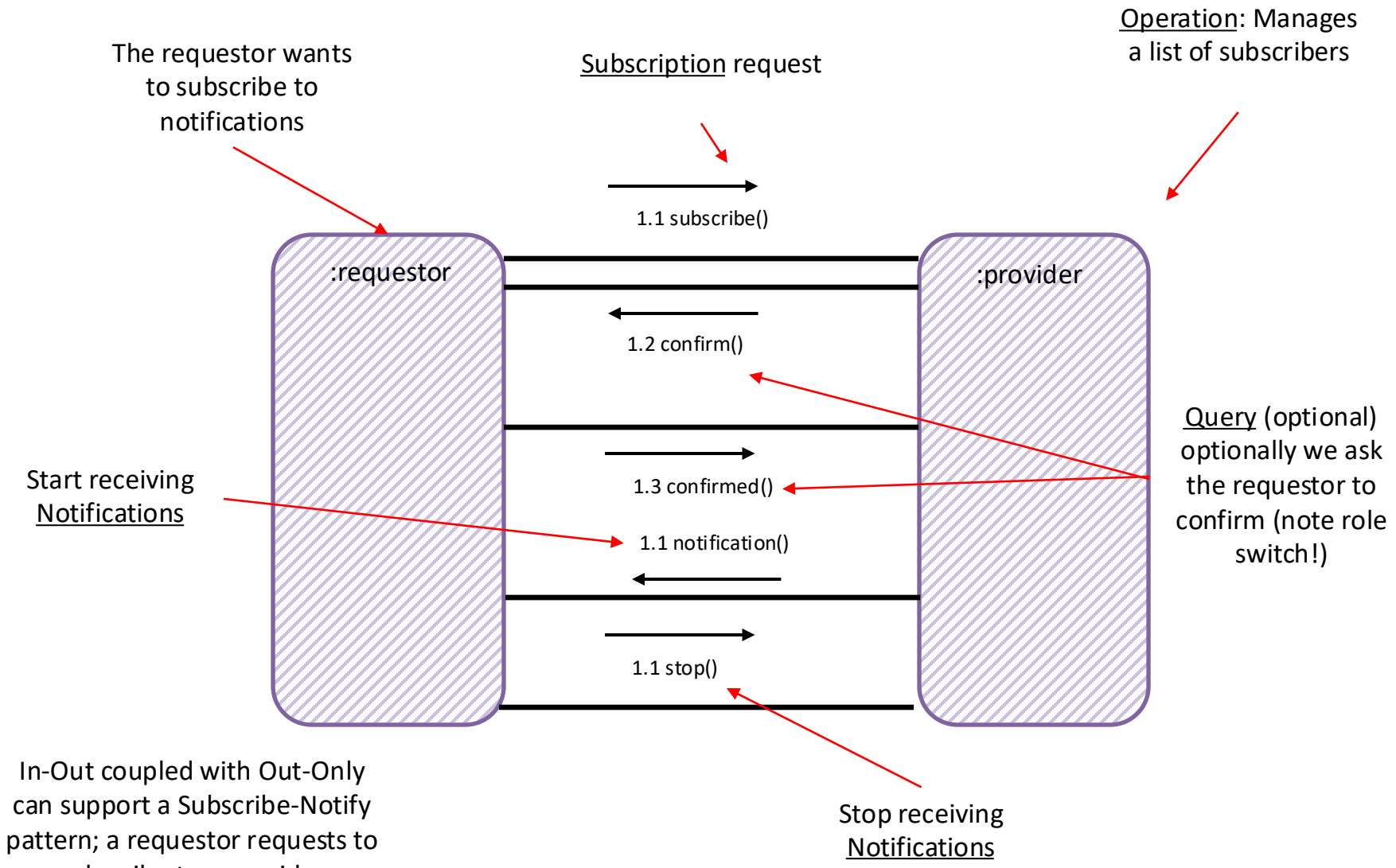
Request-Reaction (Command-Acknowledgement)



In-Out (Query-Result)



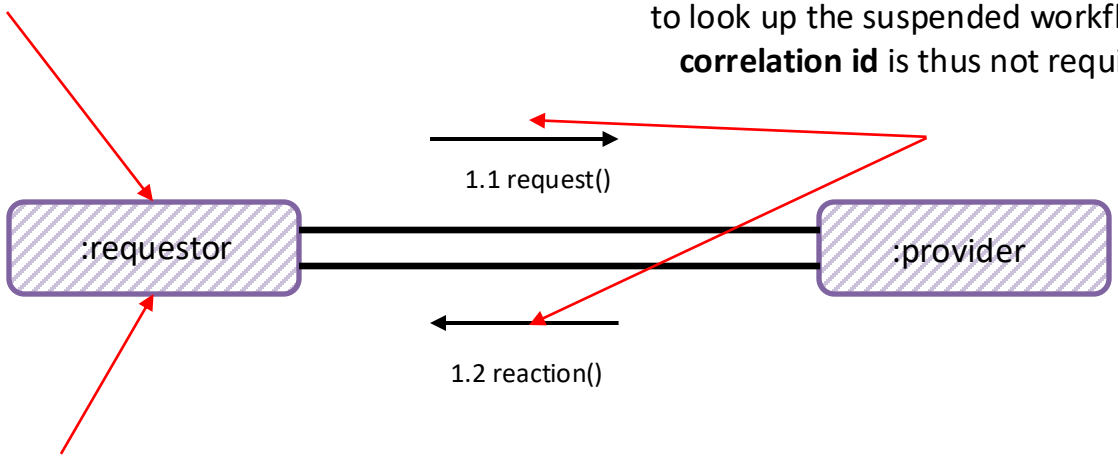
In-Out (Subscribe-Notify)



Blocking In-Out (Request-Reply)

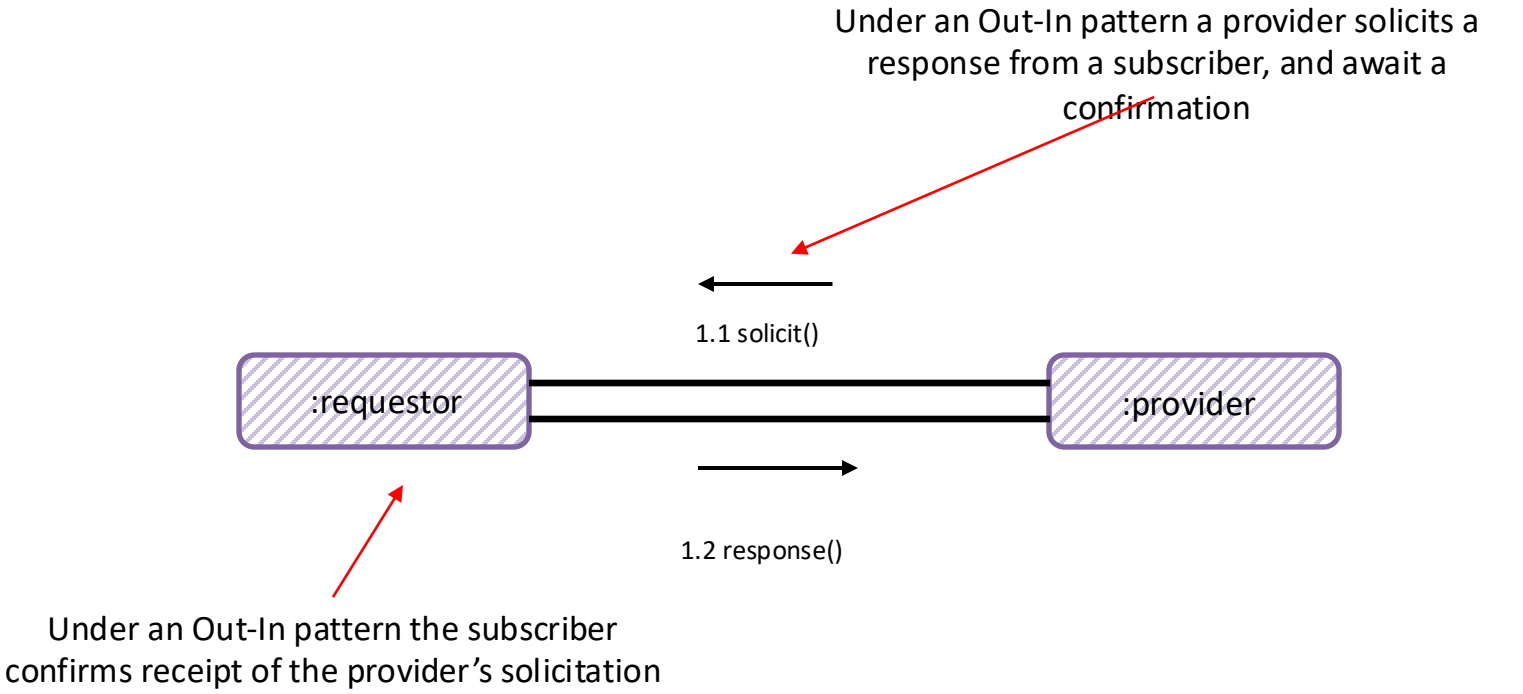
In order to resume, we need a way to identify the suspended workflow.

The requestor provides a unique **Reply To** channel (in the request message header/metadata) and the provider uses that channel for the response, allowing the sender to look up the suspended workflow. A **correlation id** is thus not required

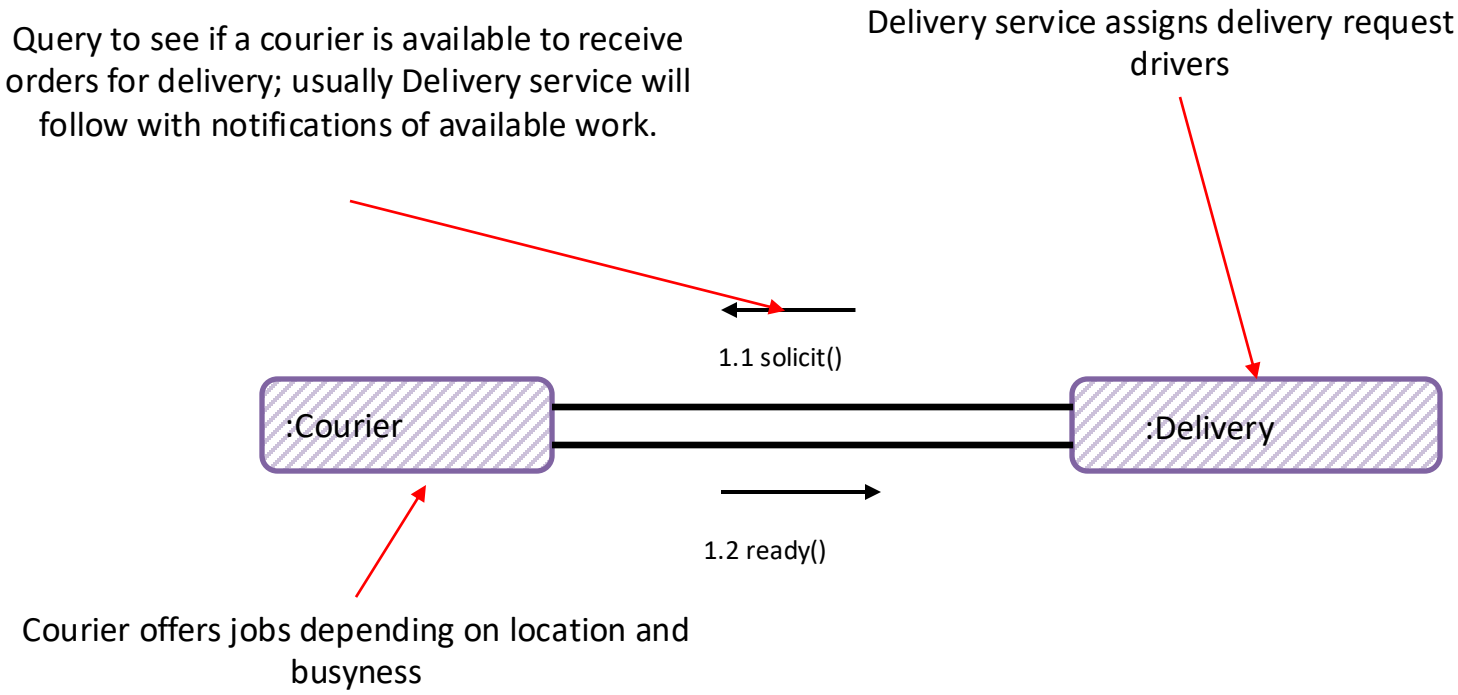


The requestor may block on the Reply To channel whilst awaiting the reply.

Out-In (Solicit-Response)



Out-In (Solicit-Response)



Messaging

Has Intent

Request An Answer (Query)

Transfer of Control

(Command)

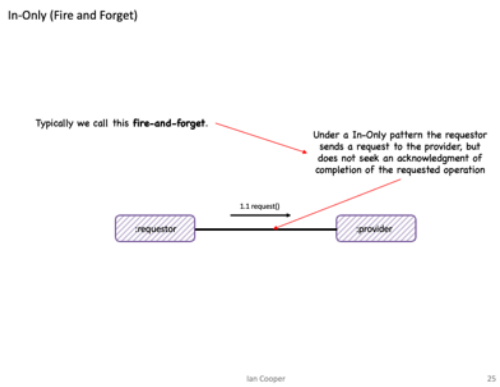
Transfer of Value

Part of a Workflow

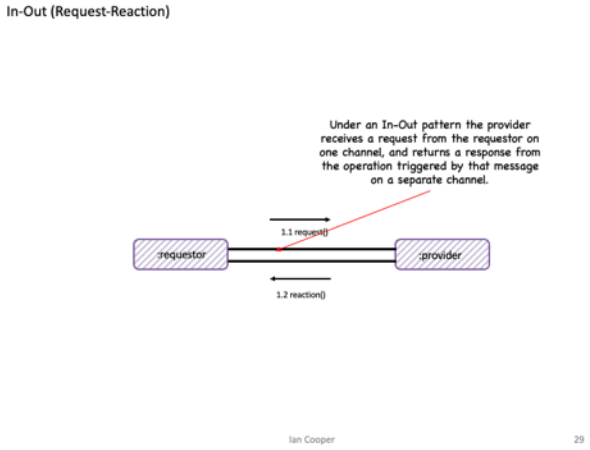
Part of a Conversation

Concerned with the Future

In-Only (fire and forget)



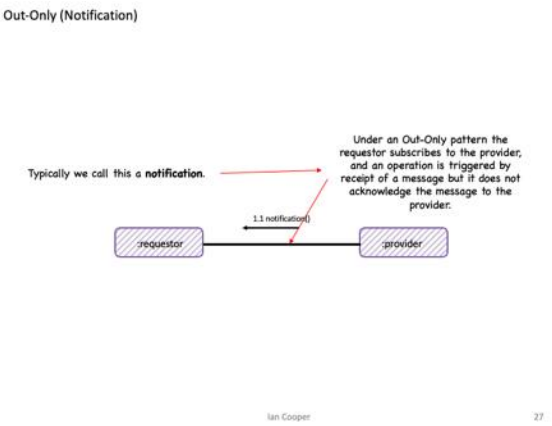
In-Out (request-reaction)



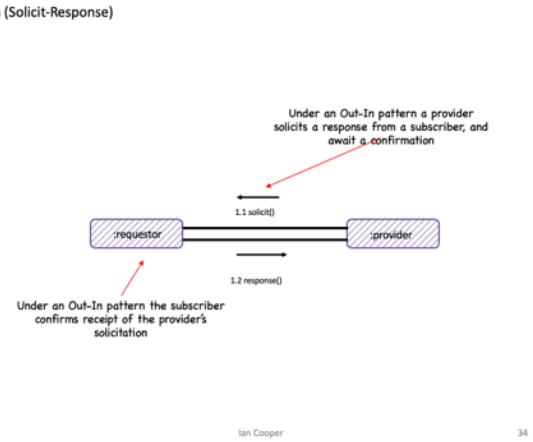
Eventing

- Provides Facts
- Things you Report On
- No Expectations
- History
- Context
- Concerned with the Past

Out-Only (notification)

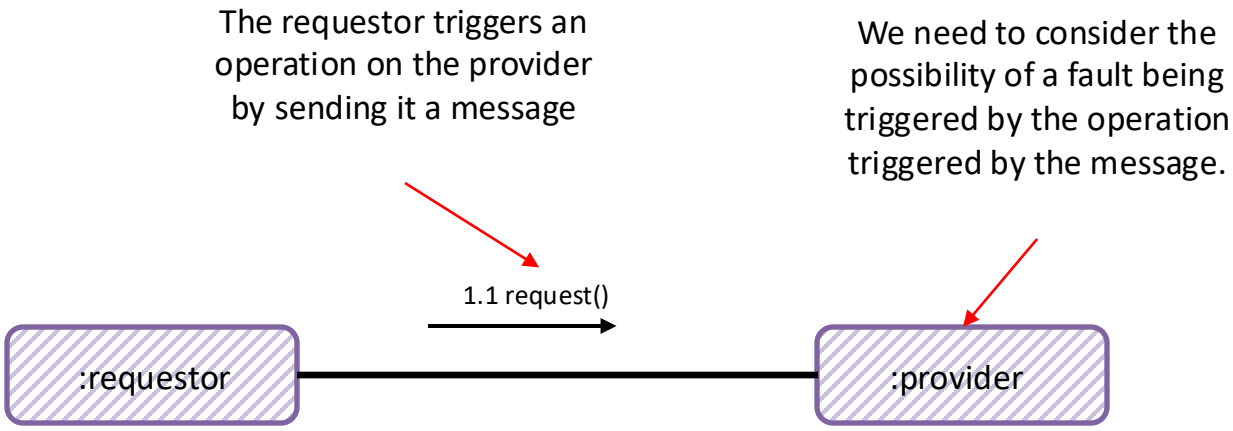


Out-In (solicit-response)



Repair and Clarification

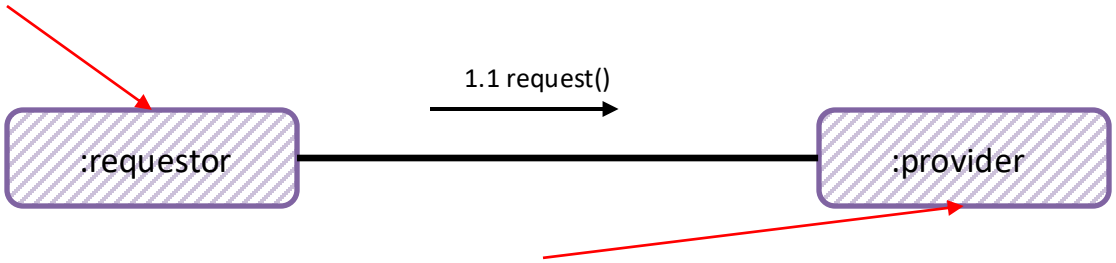
Repair and Clarification



What guarantees does the provider make to the requestor about communicating any faults that are triggered?

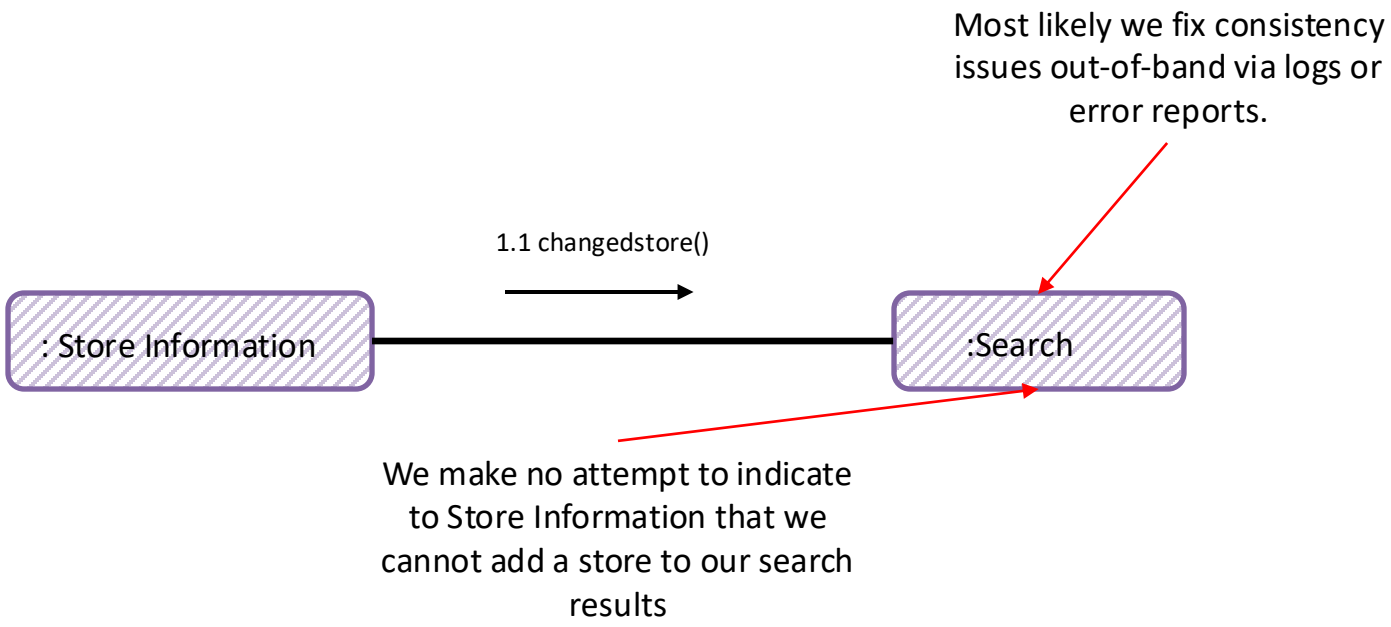
No Fault

From the requestor's perspective, faults are an application issue for the provider, and not its concern



Under a No Fault pattern the provider makes no attempt to communicate triggered faults from the operation to the requestor

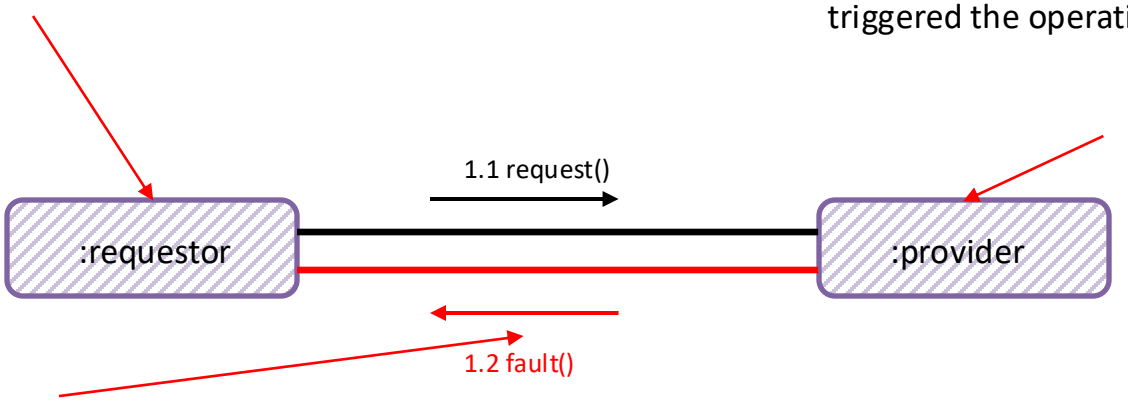
No Fault


Most likely we fix consistency issues out-of-band via logs or error reports.
1.1 changedstore()
: Store Information
:Search
We make no attempt to indicate to Store Information that we cannot add a store to our search results

Message Triggers Fault (Robust In-Only)

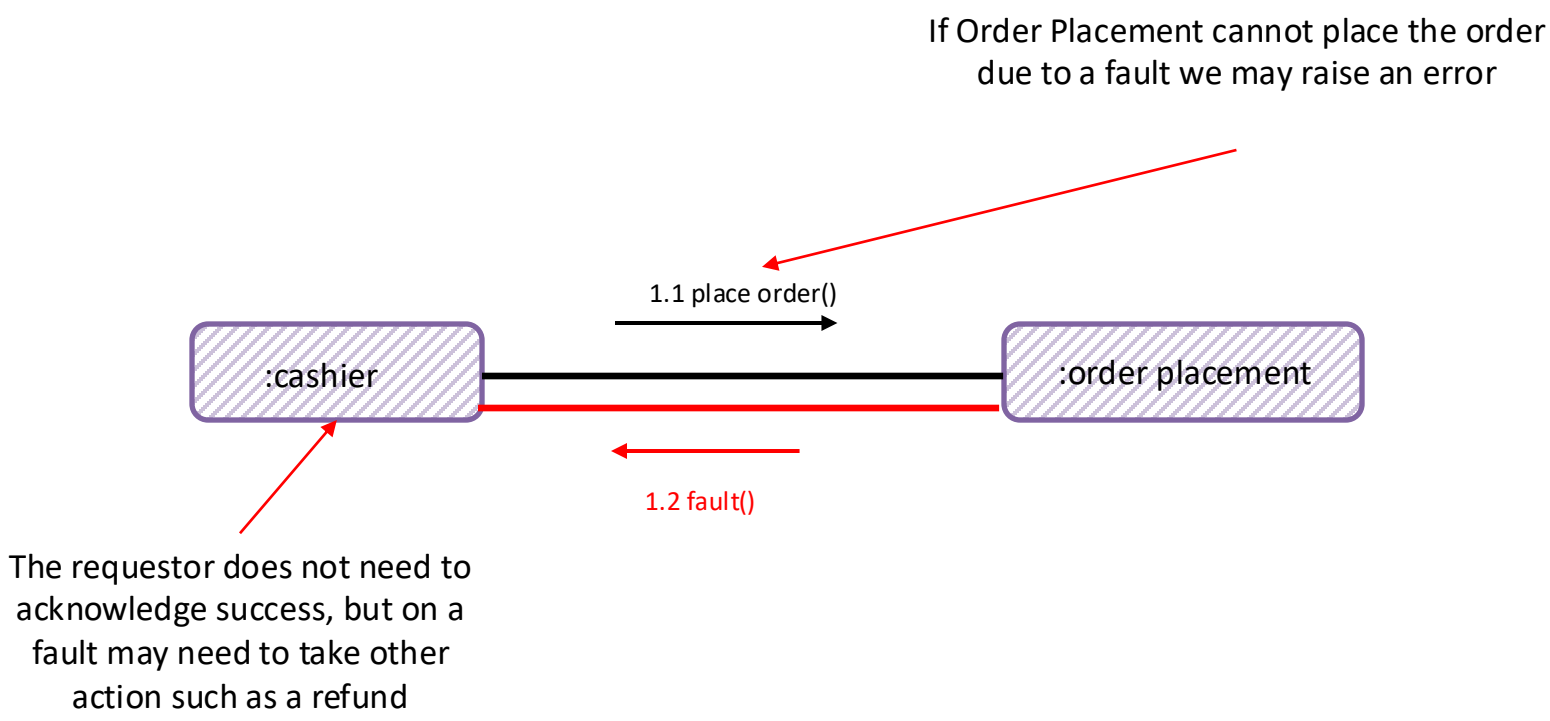
Assumption here is that the requestor can take action on receipt of the fault; but does not normally take a response.

Under Message Triggers Fault pattern a **provider** propagates triggered faults from the operation back to the **requestor** that triggered the operation via a message



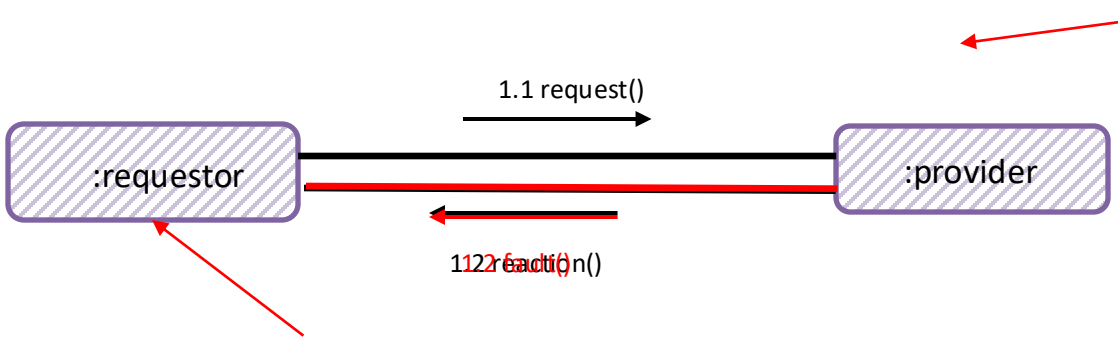
The message must have the opposite direction and go back to the requestor

Message Triggers Fault



Fault Replaces Message

Under the Fault Replaces Message pattern a provider propagates faults triggered by an operation by switching to a fault flow, replacing any message subsequent to the first with a fault.

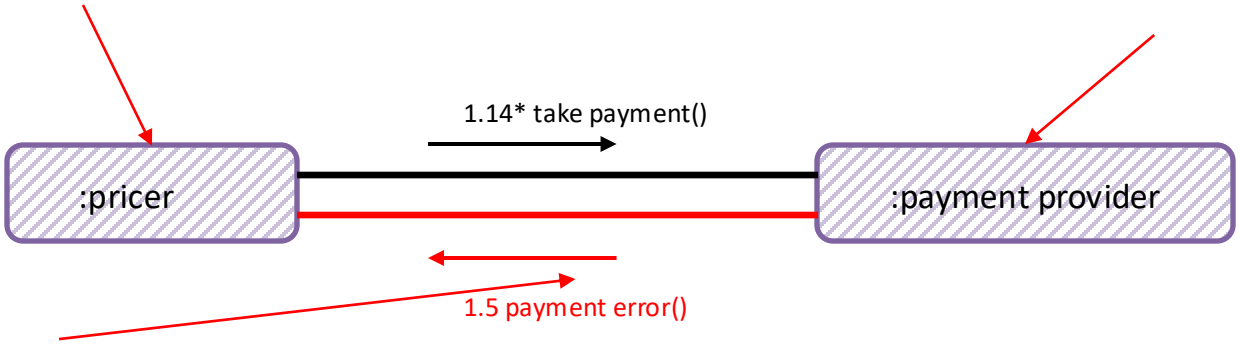


The requestor can handle the error; the error replaces the existing response

Fault Replaces Message (Robust In-Out)

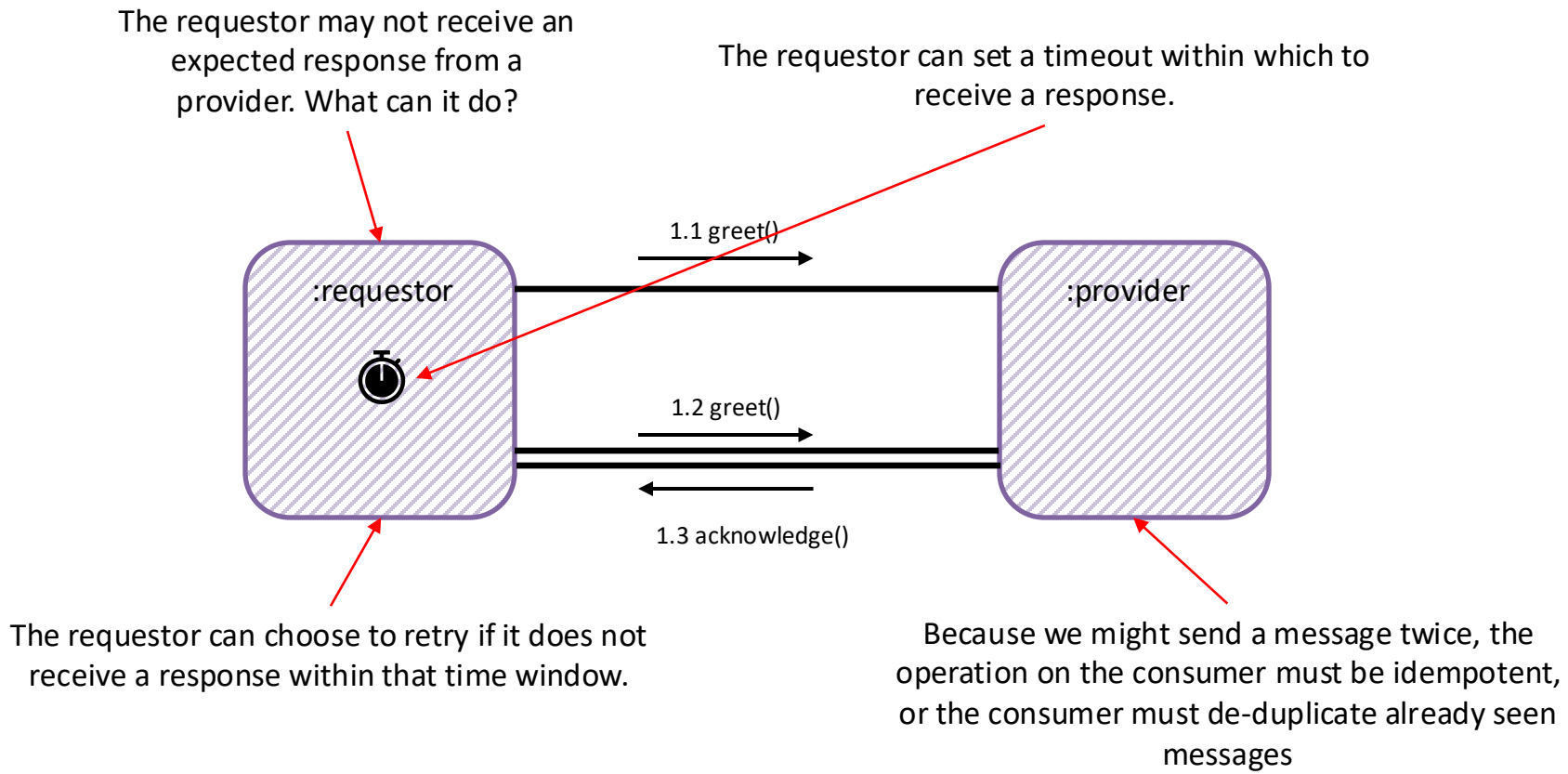
Assumption here is that the pricer can orchestrate a new flow, such as asking for an alternate payment method, or cancel the order.

Under Fault Replaces Message pattern the payment **provider** would signal errors taking a card payment back to the pricer



The message should indicate why the payment failed. This might be an issue with the payment provider but it also might be an issue like an invalid card or insufficient funds

In-Out-Retry

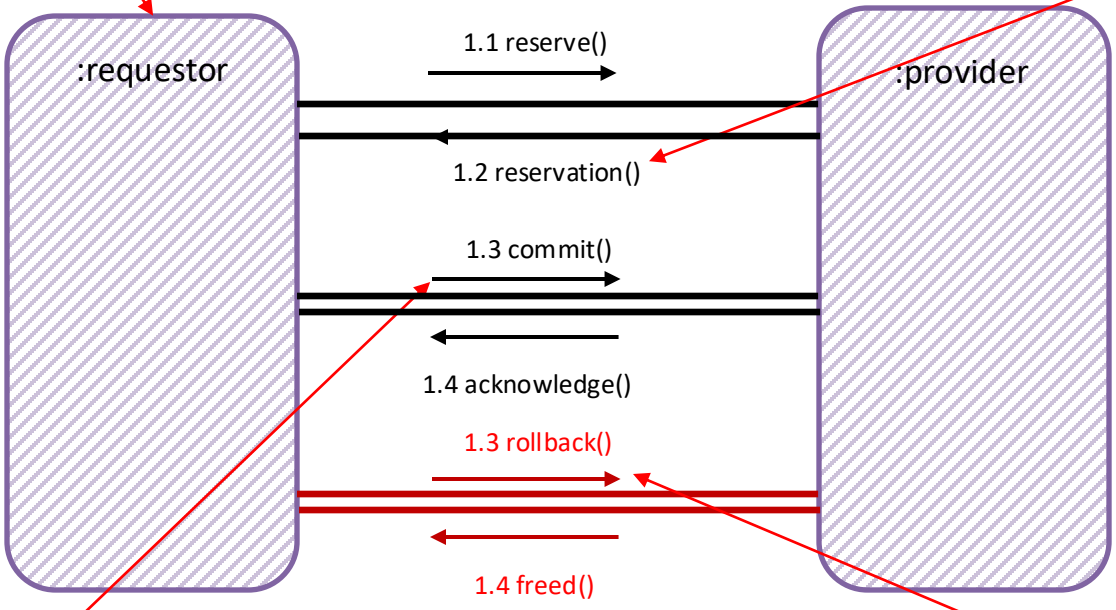

The requestor may not receive an expected response from a provider. What can it do?
The requestor can set a timeout within which to receive a response.
1.1 greet()
:requestor
:provider
1.2 greet()
1.3 acknowledge()
The requestor can choose to retry if it does not receive a response within that time window.
Because we might send a message twice, the operation on the consumer must be idempotent, or the consumer must de-duplicate already seen messages

Tentative Operations

The requestor may not know if the provider will be able to succeed, and might not want to proceed without knowing that

The requestor asks if the operation is possible, and reserves the resources to perform it.

The provider reserves the resource – usually with a timeout – during which time it awaits a message to commit or rollback



If the requestor completes its work, it then commits to ask the provider to allocate the reserved capacity.

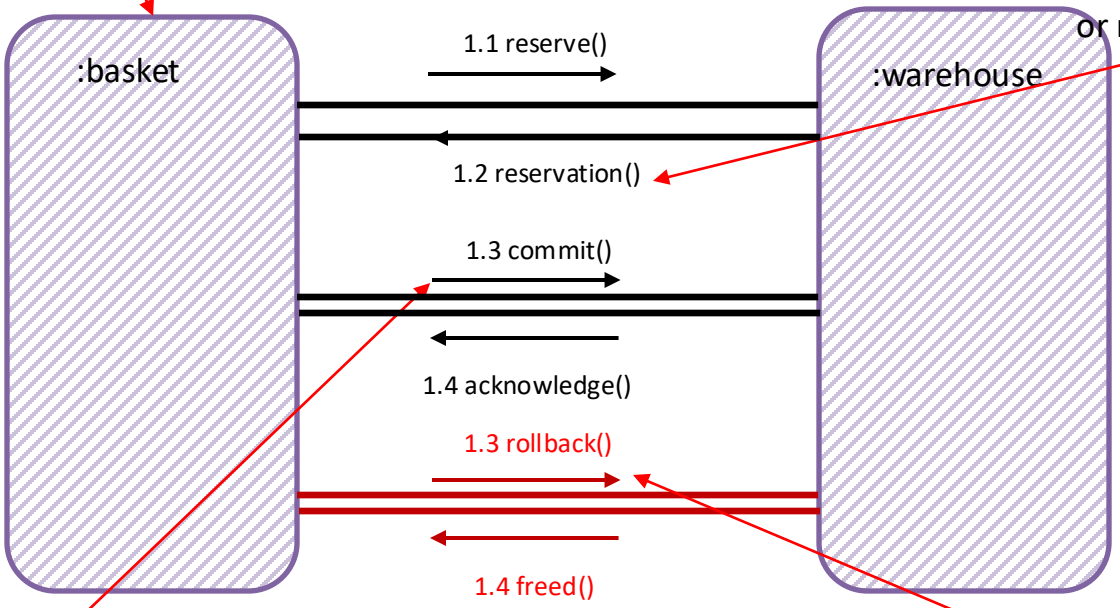
If the requestor fails, it then rolls back to ask the provider to free the reserved capacity.

Tentative Operations

We don't want to purchase the item if the stock is not available

The basket asks the warehouse if there is stock, and if so reserves it. Fault could replace message here, if the item is out-of-stock

The warehouse reserves the stock – with a timeout (so others can purchase if we don't) – during which time it awaits a message to commit or rollback

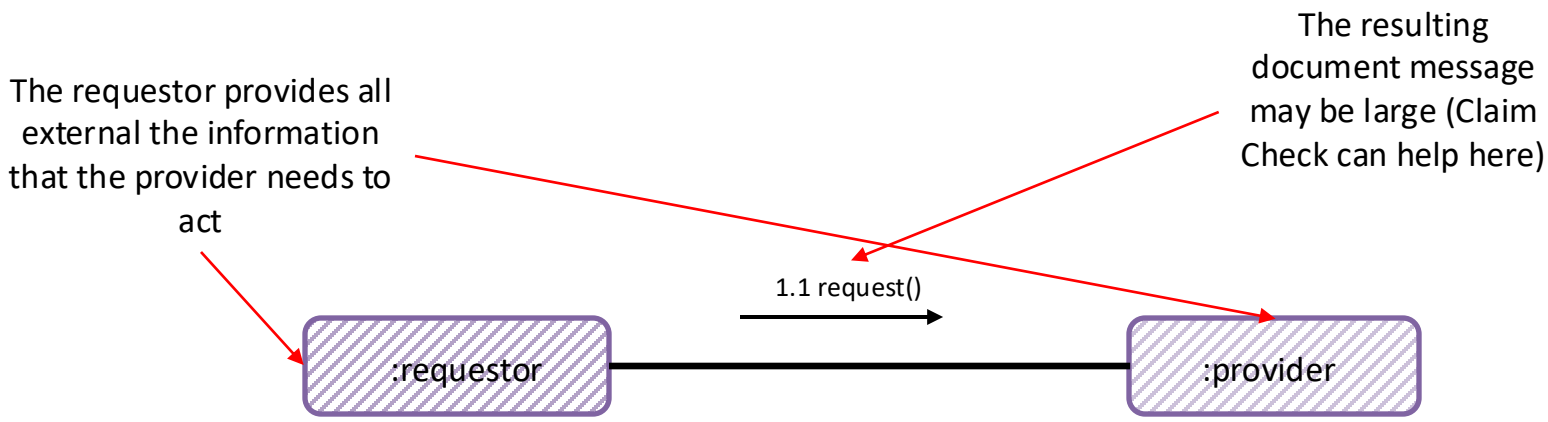


If the customer finishes shopping and pays for the basket before the time limit, then it commits to ask the warehouse to allocate the stock.

If the basket does not complete, then it rolls back to ask the warehouse to free the stock.

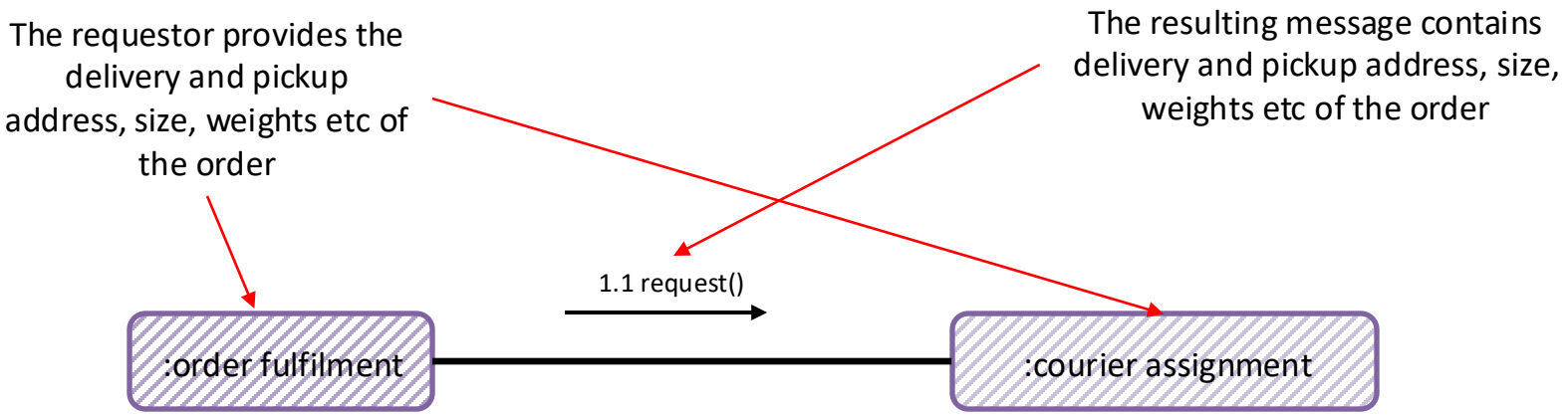
Fat & Skinny

Fat Message



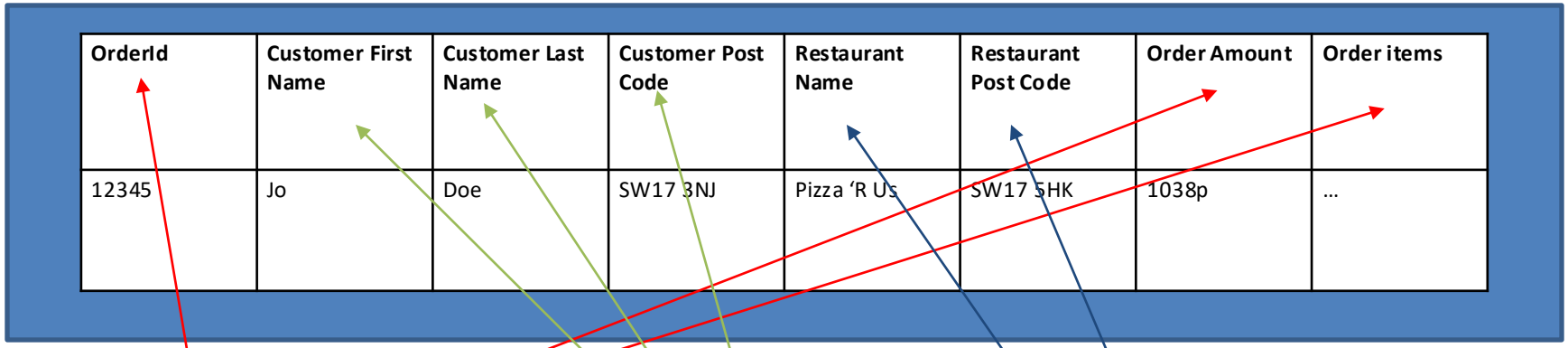
With a **Fat Message** the requestor sends across all the external information a provider may need to perform the operation.

Fat Message Example



Fat Message, Transitive Dependencies

Purchase Order Message



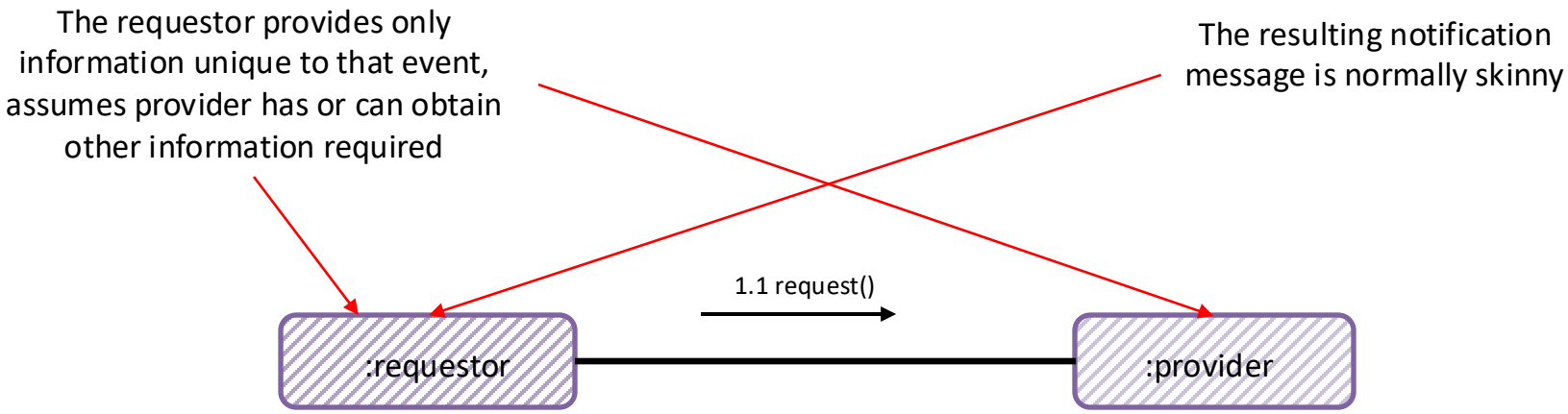
OrderId	Customer First Name	Customer Last Name	Customer Post Code	Restaurant Name	Restaurant Post Code	Order Amount	Order items
12345	Jo	Doe	SW17 3NJ	Pizza 'R Us	SW17 5HK	1038p	…

This data has a lifetime of the message i.e. the purchase order and their schema changes if the purchase order changes

This data is a **transitive dependency**, it has a lifetime of the Customer and their schema changes if the Customer schema changes – which may force us to change the message for Customer changes

This data is a **transitive dependency**, it has a lifetime of the Restaurant and their schema changes if the Restaurant schema changes – which may force us to change the message for Restaurant changes

Skinny Message

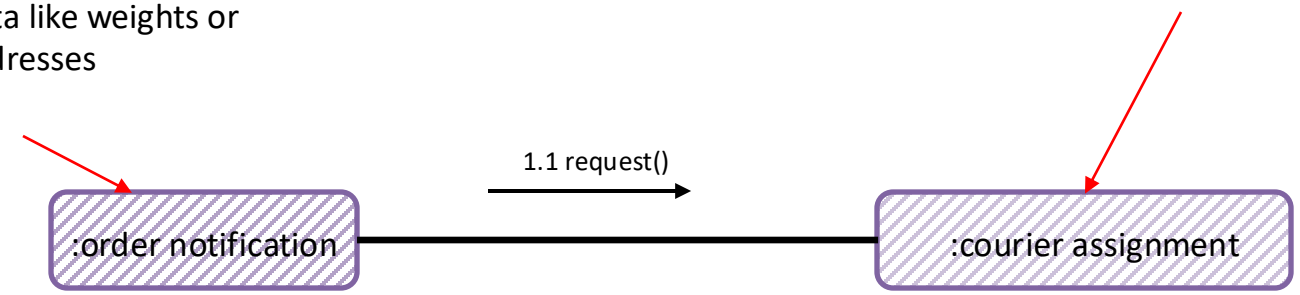


With a **Skinny Message** the requestor assumes the provider has necessary external information to perform the operation.

Skinny Request, Example

The requestor provides a notification that there is an order, but only inlines data that is unique to the order not reference data like weights or addresses

Courier assignment needs information that is not on the order: weights, addresses. It has to source that from elsewhere



Skinny Message, Normalized

Purchase Order Message

OrderId	Order Amount	Order items	CustomerId	RestaurantId
12345	1038p	…		

Customer

Customer First Name	Customer Last Name	Customer Post Code
Jo	Doe	SW17 3NJ

We use an Id where the data does not share the lifetime of the message; we assume the requestor obtains the data out-of-band

Restaurant

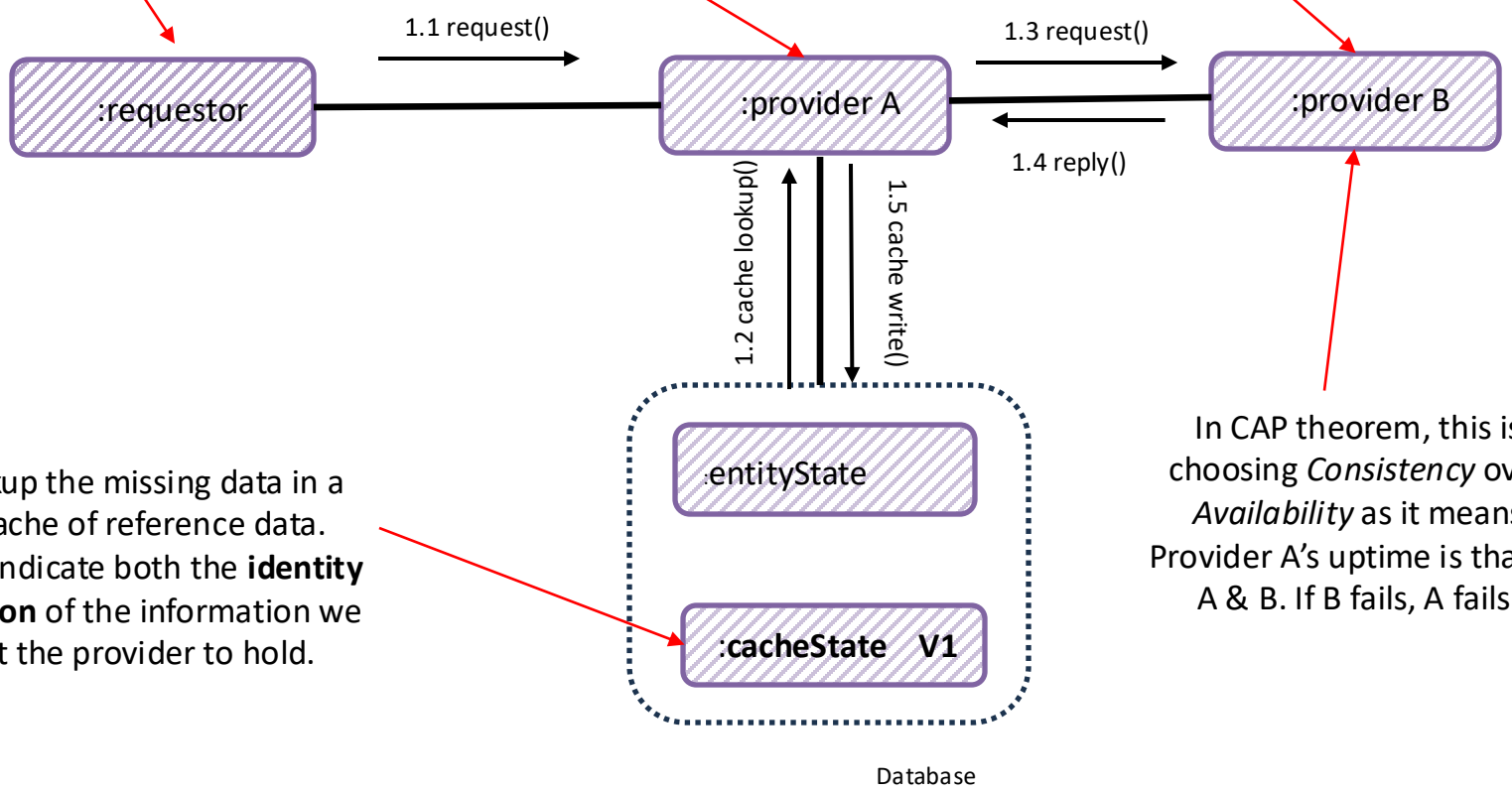
Restaurant Name	Restaurant Post Code
Pizza 'R Us	SW17 5HK

We need to lookup this id with other providers

Skinny Message via REST/RPC

The requestor provides only information unique to that event, assumes *provider A* has or can obtain information from *provider B*

On a cache miss we request the data from provider B, and store it in the cache. The cache is choosing *Availability* over *Consistency*, **except** on a miss.



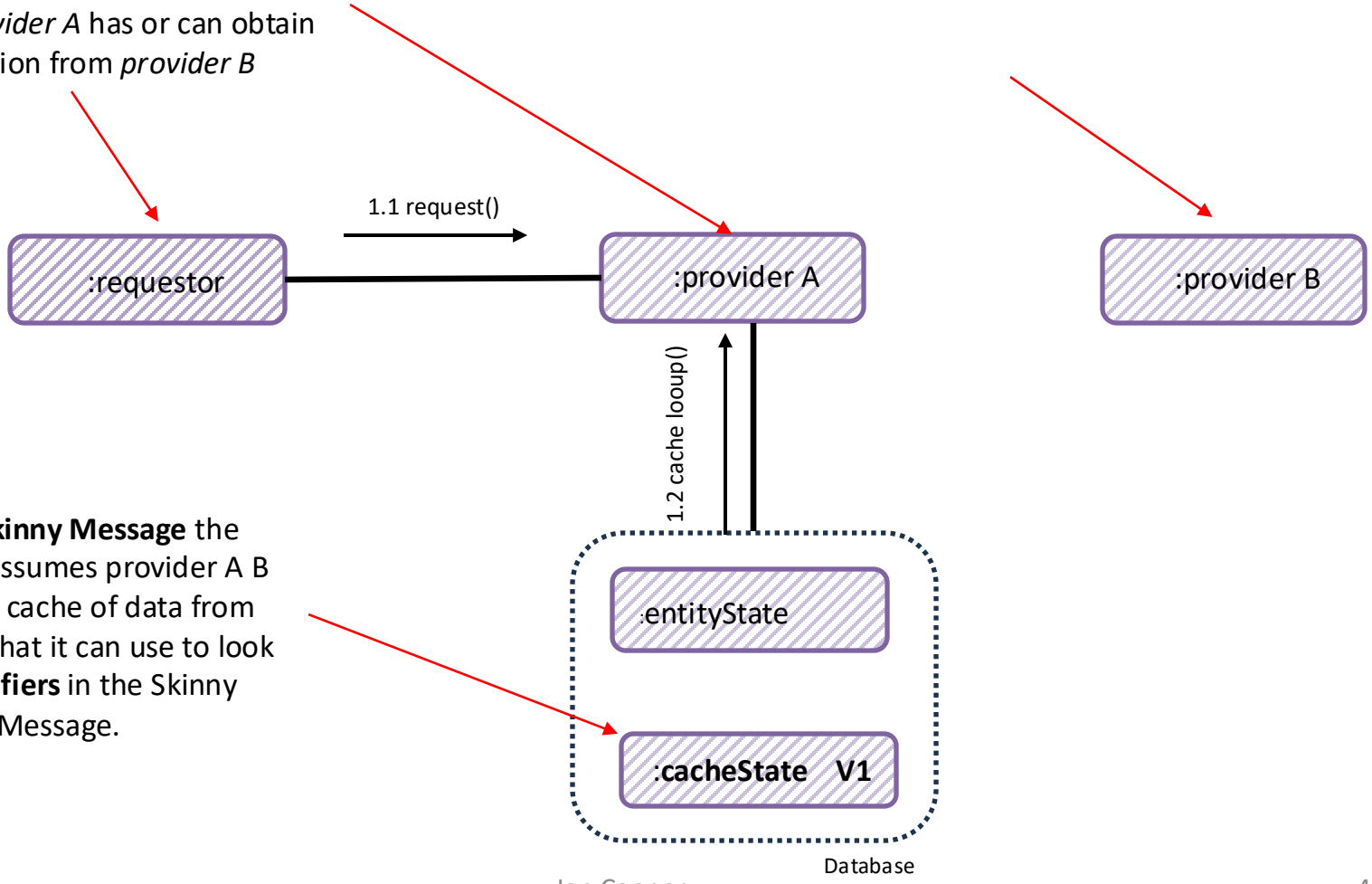
We lookup the missing data in a local cache of reference data. We may indicate both the **identity** and **version** of the information we expect the provider to hold.

In CAP theorem, this is choosing *Consistency* over *Availability* as it means Provider A's uptime is that of A & B. If B fails, A fails.

Skinny Message using Reference Data

Data that leaves Provider B via an API is **Reference Data**.
It is:
- *Immutable*
-*Versioned*
-*Stale*
It can be cached locally to provider A to prevent the need for frequent lookups

The requestor provides only information unique to that event, assumes *provider A* has or can obtain information from *provider B*

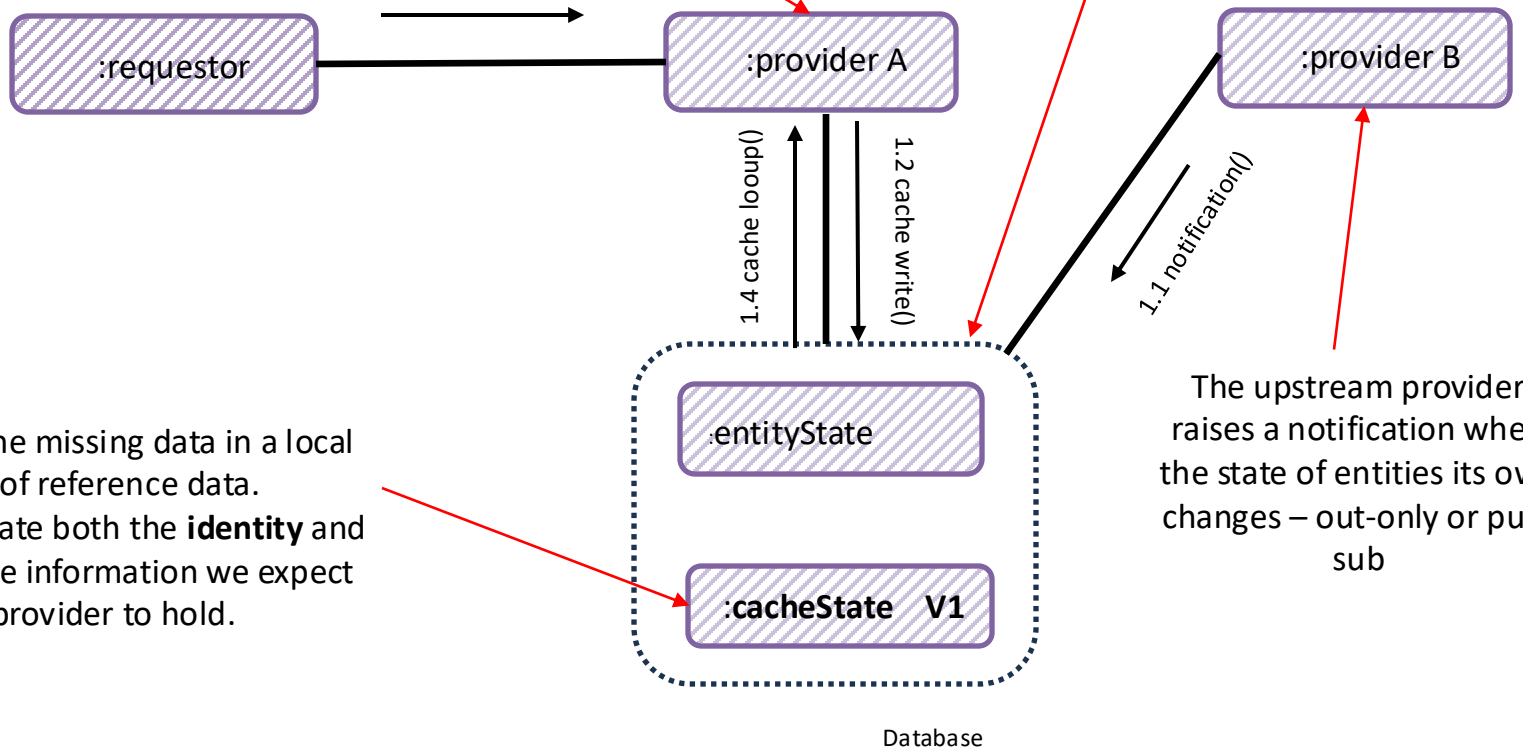


With a **Skinny Message** the requestor assumes provider A B has a local cache of data from provider B that it can use to look up **identifiers** in the Skinny Message.

Skinny Message, Reference Data via ECST (Event Carried State Transfer)

The requestor provides only information unique to that event, assumes *provider A* has or can obtain information from *provider B*

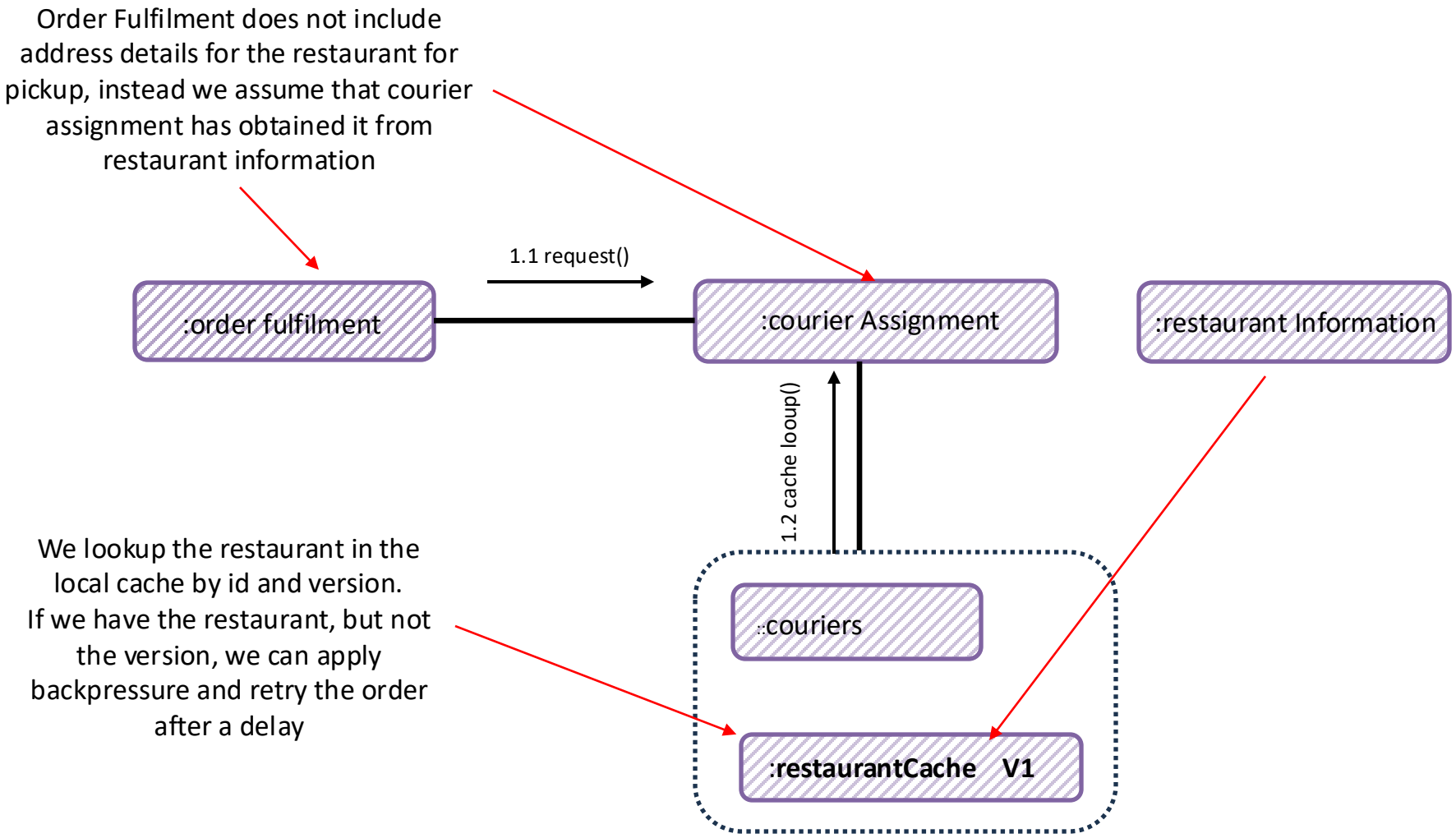
The downstream provider subscribes to these notifications and writes these to the local cache.
In CAP Theorem we choose *Availability* over *Consistency* – we choose to accept stale data over accepting the risk of failure due to a *Partition*.



We lookup the missing data in a local cache of reference data.
We may indicate both the **identity** and **version** of the information we expect the provider to hold.

The upstream provider raises a notification when the state of entities its own changes – out-only or pub-sub

Skinny Message using Reference Data

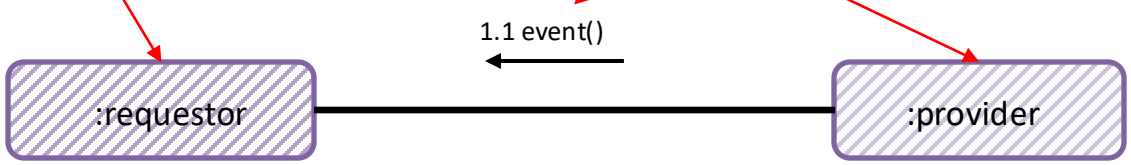


Domain, Summary & Sequence

Domain or Delta Event

An approach to Pub-Sub where the provider communicates **granular** state changes to its requestors

The resulting message is usually a past participle named after the command that caused the change i.e. *BasketItemRemoved*, *BasketItemAdded*



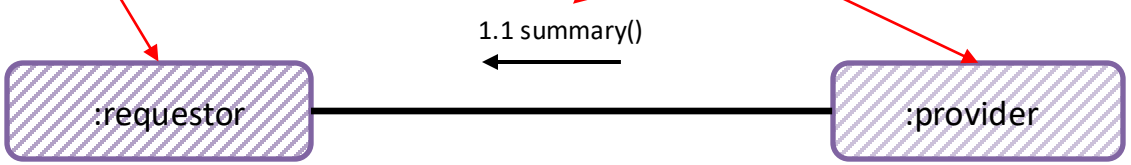
With a **Domain or Delta Event,** the provider cannot use a *DataType Channel* as the domain events for the observable must be ordered on the same channel, and have a different schema.

This puts an additional burden on the requestor to multiplex handling of the event

Summary or Snapshot Event

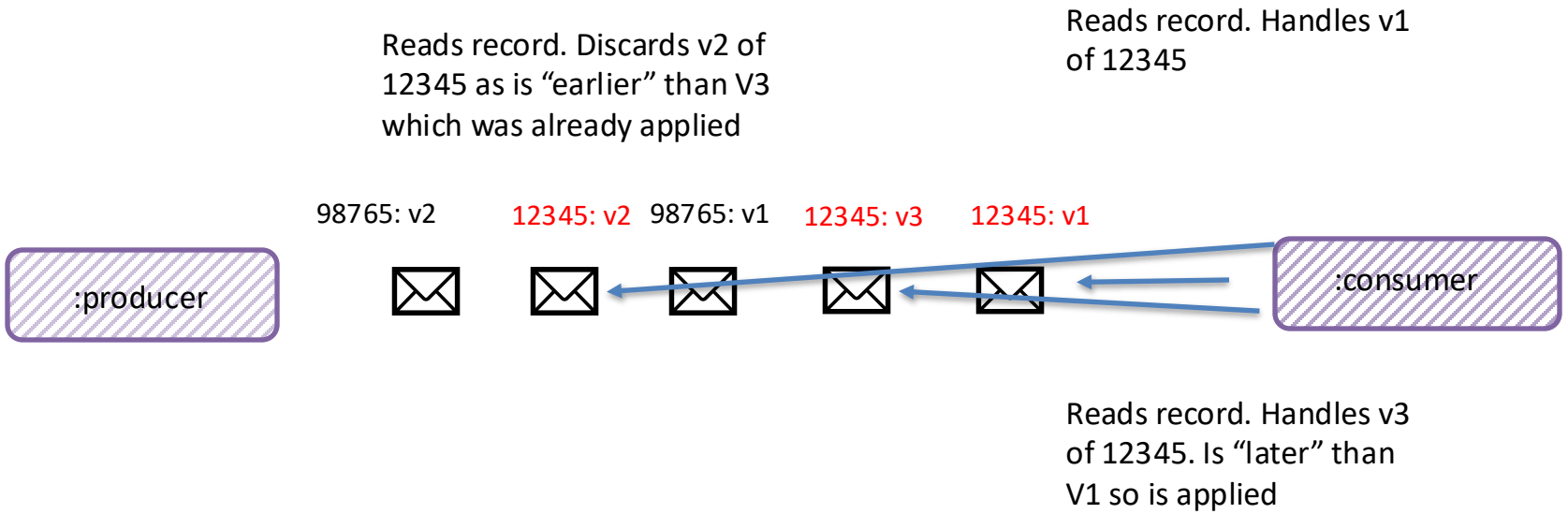
An approach to Pub-Sub where the provider communicates **a summary** of state changes to its requestors

The resulting message is **versioned** and contains *metadata describing the cause(s)* of any changes. It is usually named after the observable in the pub-sub i.e. *BasketChanged*



With a **Summary or Snapshot Event** the provider can use a *DataType Channel,* as there is just one schema used to communicate a snapshot of the observable.

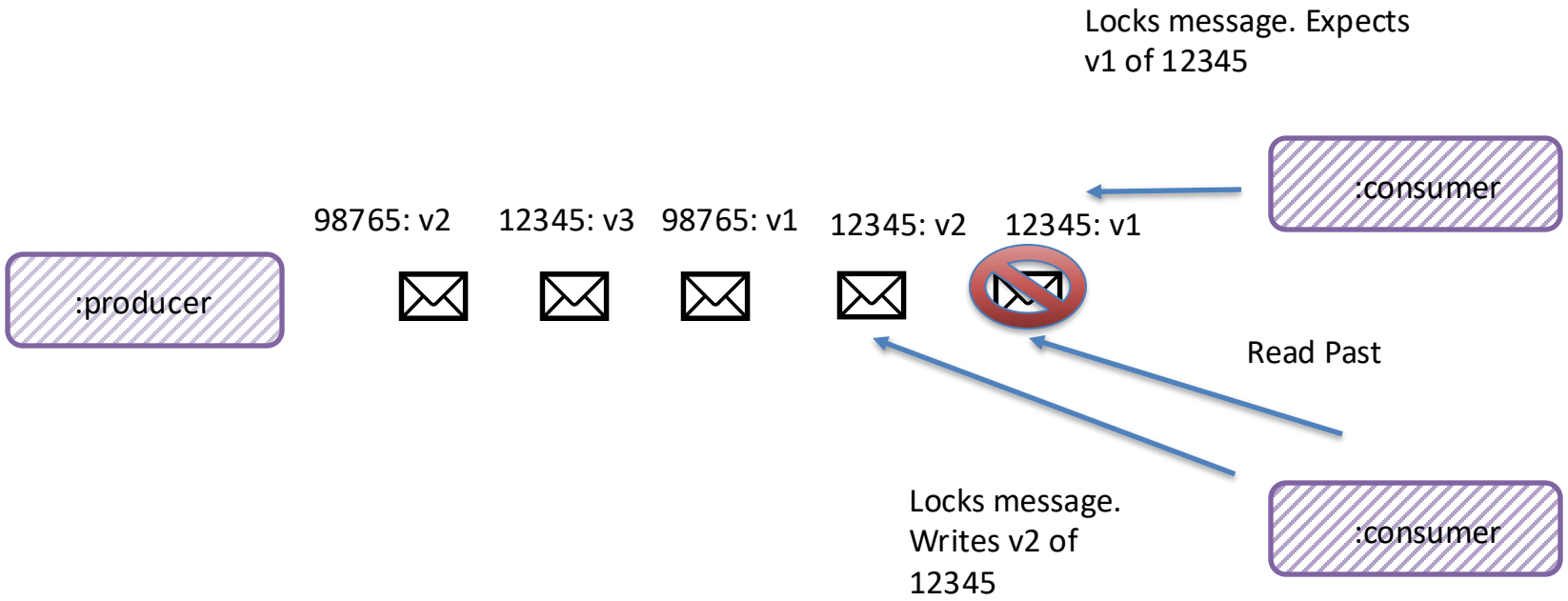
If Later, Stream



If Later allows us to use a versioned **Summary Event** to ignore ordering errors. Even on a stream, a non-blocking retry or guaranteed delivery via an outbox may result in out-of-order messages. We can also shed load

We cannot use If Later with a **Domain Event,** as they all must be applied. We can only use a blocking retry with a Domain Event and cannot shed load.

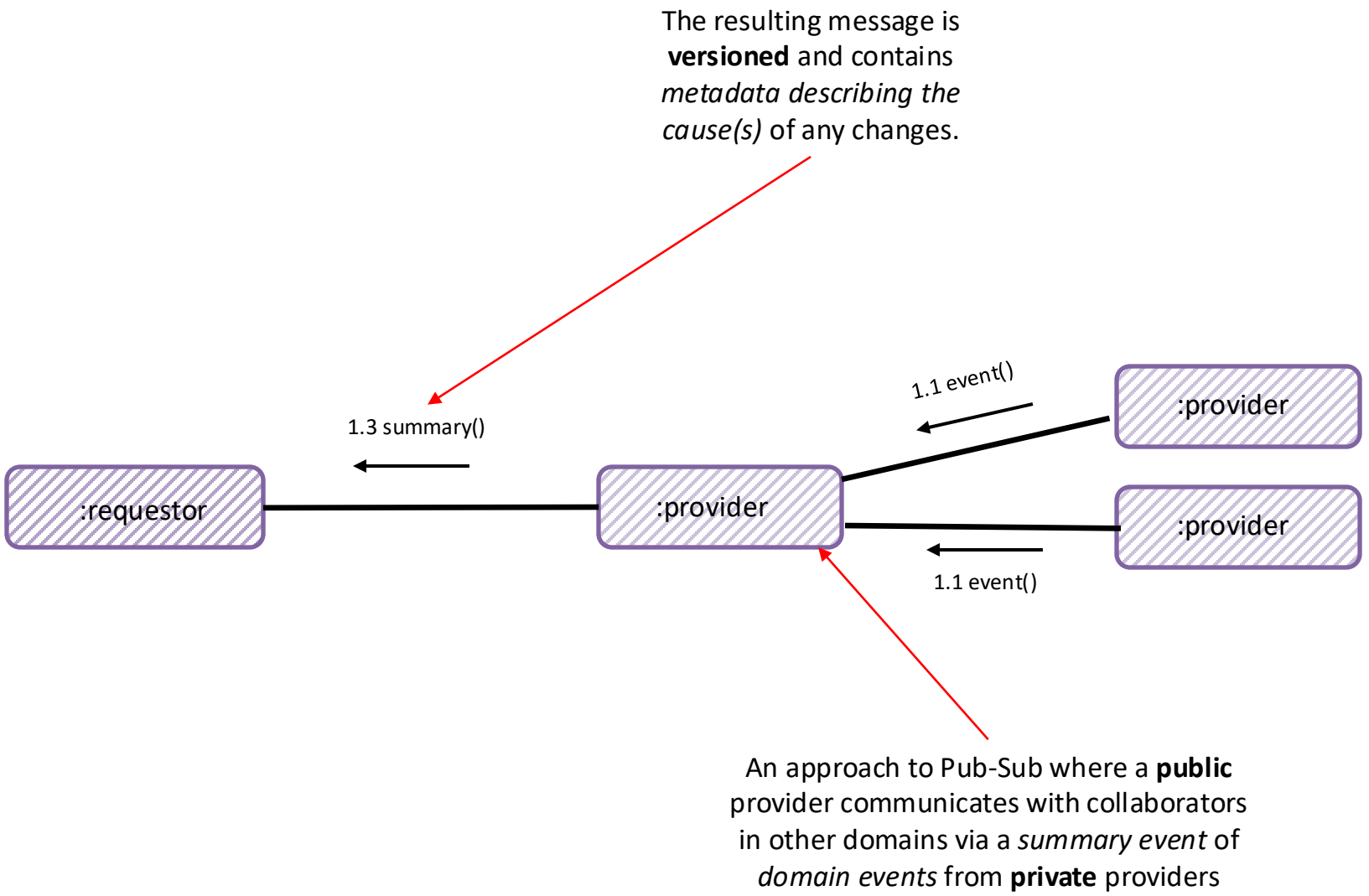
If Later, Queue



If later allows message to processed be out-of-order with a queue and competing consumers. A queue normally processes messages, and not events, so this only applied where we are using a queue.

If a message must be ordered, such as a series of commands, we must use requeue with delay, or a sequencer to re-order.

Public and Private Providers



FLOW

Paper Workflows

"My life looked good on paper - where, in fact, almost all of it was being lived."
- Martin Amis

Messaging/Discrete Event

Discrete, Immediately Actioned

Skinny

Series Event (Document)

Series, Supports Action

Fat


OUT
IN
gettyimages
Comstock Images


Walmart.com

OE DIVISION
OEØ1 ROUTING SHEET

TITLE: EIB-E15 (23 July 84)

NAME	REQD	DATE IN	DATE OUT	INITIAL
ET1 BARMAN	R		8/12/84	AFB
ET2 SWEET	I			
ET2 BALABUSHKA	A			
ET2 MURLBUTT	I			
ET2 JENNINGS	I			
ET3 MORGAN	I			
ET3 JONES	A			
ET3 JACKSON	I			
ET3 MYERS	I			

REQD legend; I - Information
A - Action
R - Retain

Upon Completion, Return to ET1 Barman

Destroy
File

OPS/OE Form No. XXXXXXXXX


FAX
brother

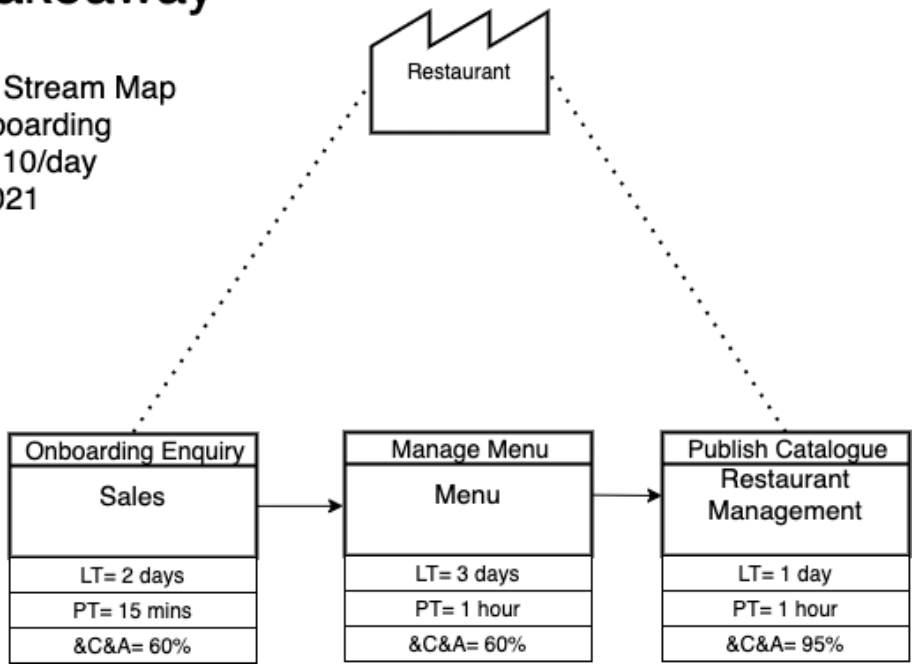
What is a microservice?

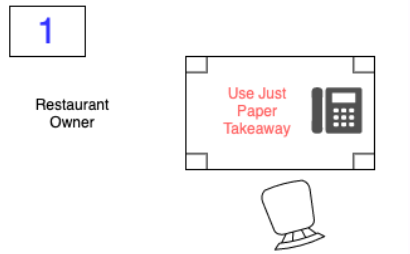
SOA is focused on business *processes*. These *processes* are performed in different steps (also called *activities* or *tasks*) on different systems. The primary goal of a **service** is to represent a “natural” step of business functionality. That is, according to the domain for which it’s provided, *a service should represent a self-contained functionality that corresponds to a real-world business activity*.

Josuttis, Nicolai M.. SOA in Practice: The Art of Distributed System Design . O'Reilly Media. Kindle Edition.

Just Paper Takeaway

Current State Value Stream Map
Restaurant Onboarding
Demand Rate 10/day
29 SEP 2021

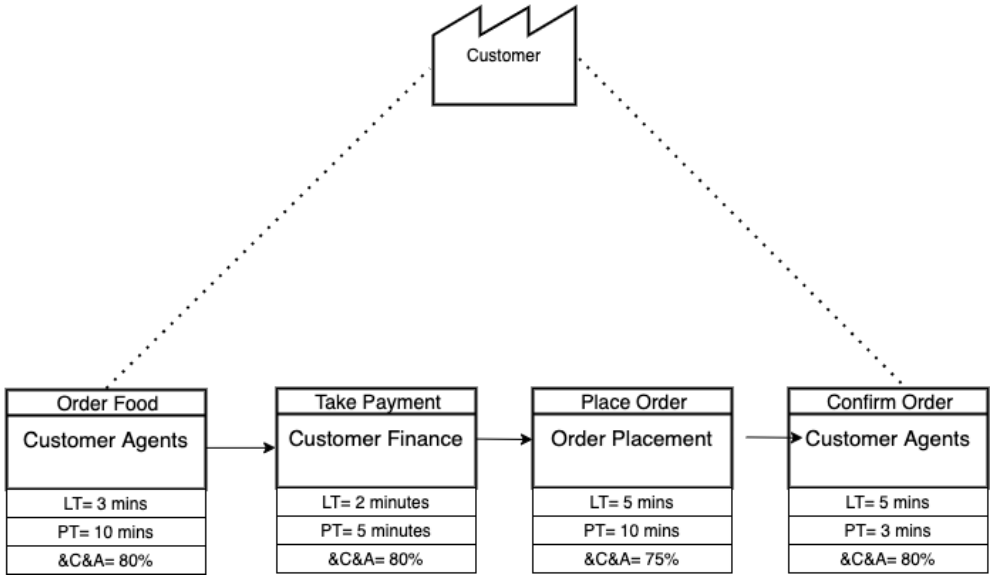


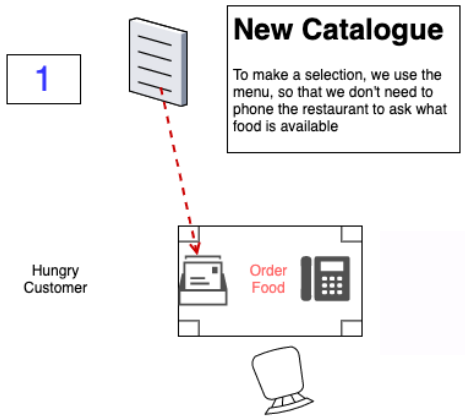


Restaurant Onboarding

Just Paper Takeaway

Current State Value Stream Map
Order Flow
Demand Rate 10/day
29 SEP 2021

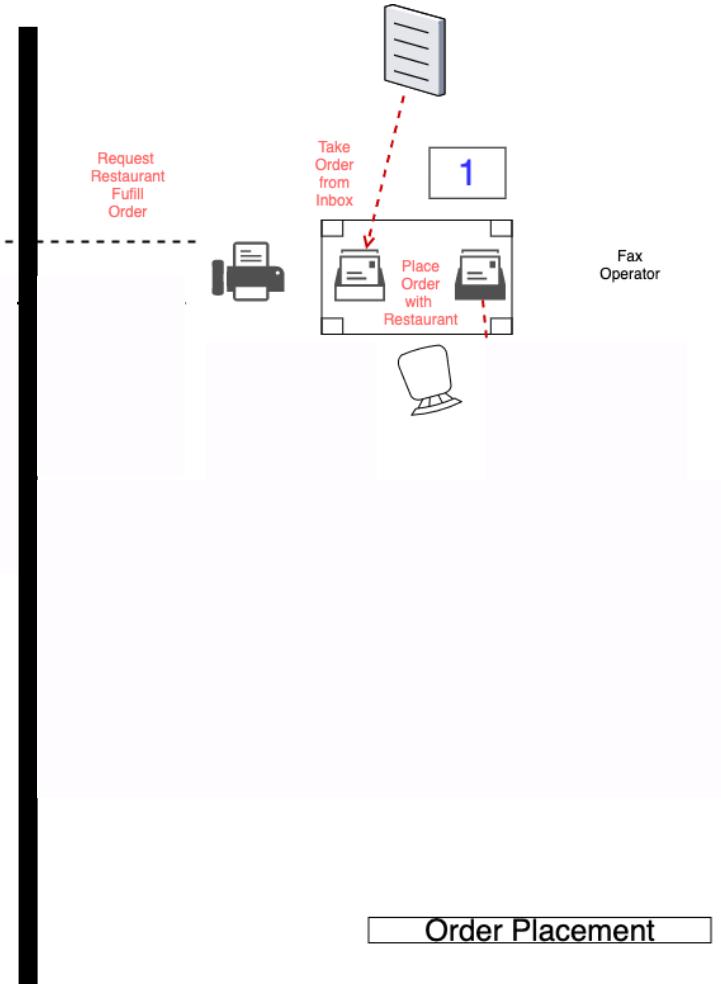


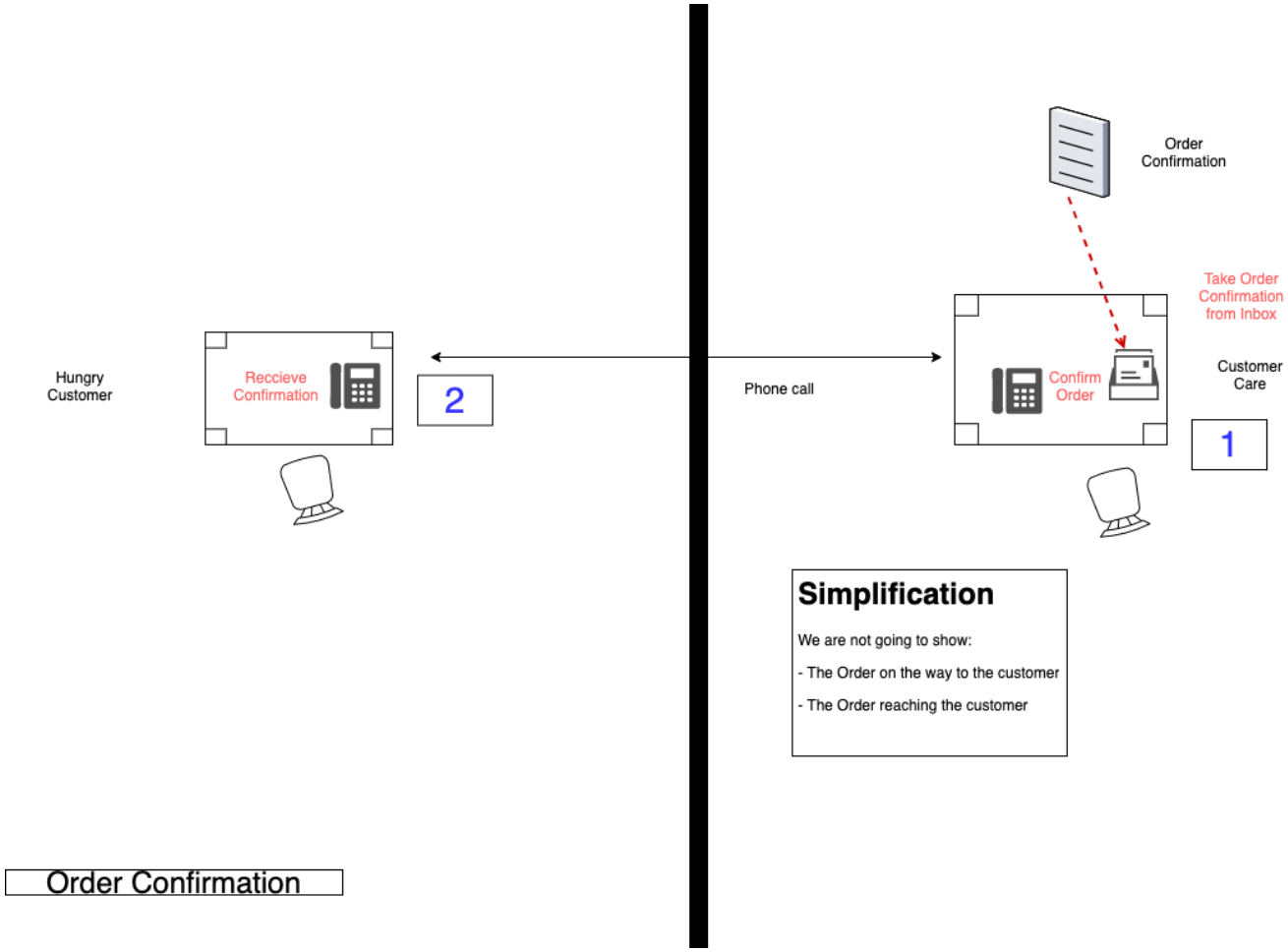


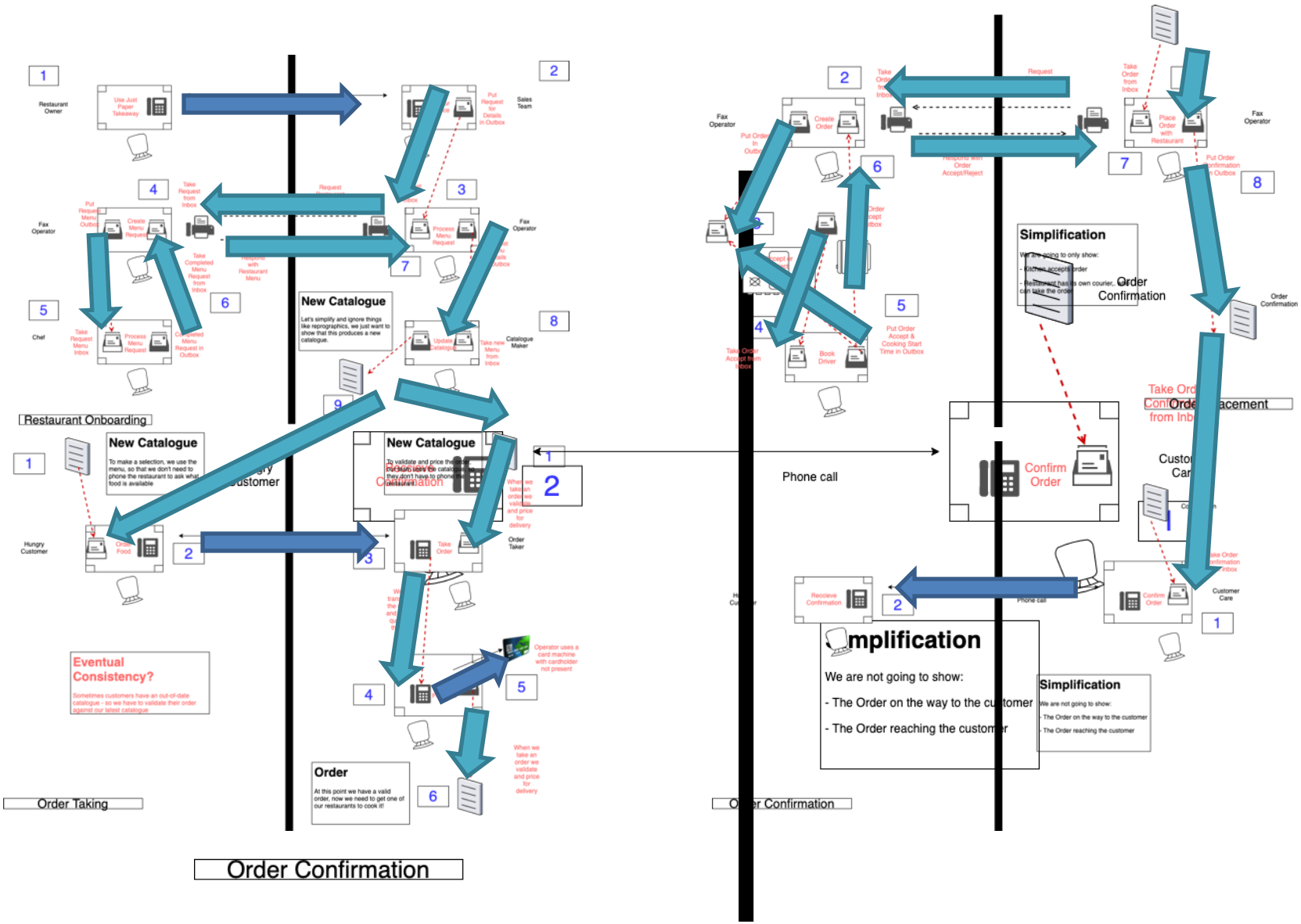
Order Taking

Order Wheel

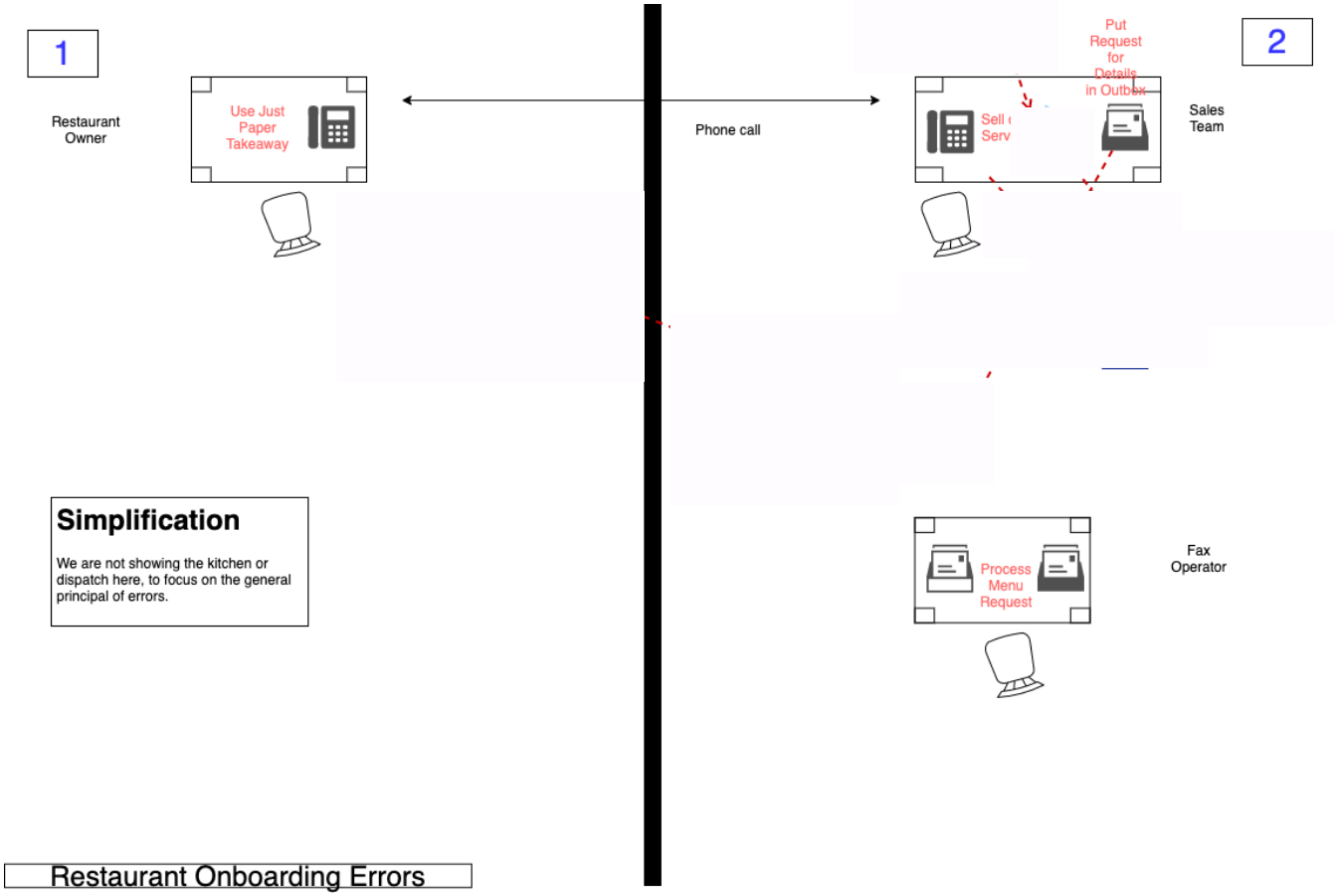

Jon 104 2
484812
Car GF
Wedge Des
1 Gr. Salmon MR
2 NY Strip M
Thank You!


Request
Restaurant
Fufill
Order
Take
Order
from
Inbox
1
Place
Order
with
Restaurant
Fax
Operator
Order Placement


Order Confirmation
Take Order Confirmation from Inbox
Hungry Customer
Reccieve Confirmation
2
Phone call
Confirm Order
Customer Care
1
Simplification
We are not going to show:
- The Order on the way to the customer
- The Order reaching the customer
Order Confirmation


Restaurant Onboarding
New Catalogue
Let's simplify and ignore things like reprographics, we just want to show that this produces a new catalogue.
New Catalogue
To make a selection, we use the menu, so that we don't need to phone the restaurant to ask what food is available
New Catalogue
Eventual Consistency?
Sometimes customers have an out-of-date catalogue - so we have to validate their order against our latest catalogue
Order
At this point we have a valid order, now we need to get one of our restaurants to cook it!
Order Taking
Order Confirmation
Phone call
Confirm Order
Simplification
We are not going to show:
- The Order on the way to the customer
- The Order reaching the customer
Order Confirmation

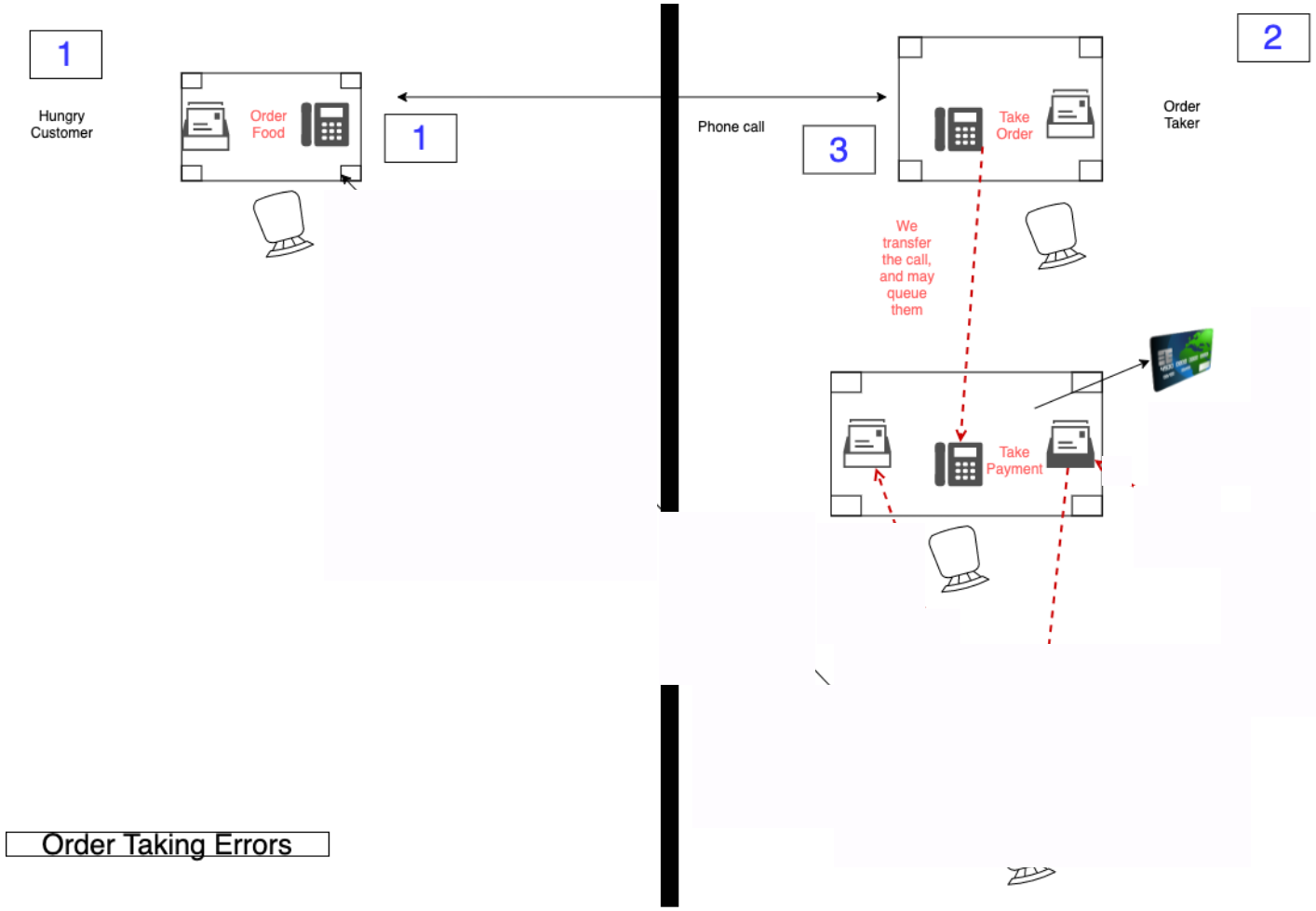

DO NOT USE!
Scrap Cost


1
Restaurant Owner
Use Just Paper Takeaway
Phone call
Put Request for Details in Outbox
2
Sales Team
Simplification
We are not showing the kitchen or dispatch here, to focus on the general principal of errors.
Process Menu Request
Fax Operator
Restaurant Onboarding Errors

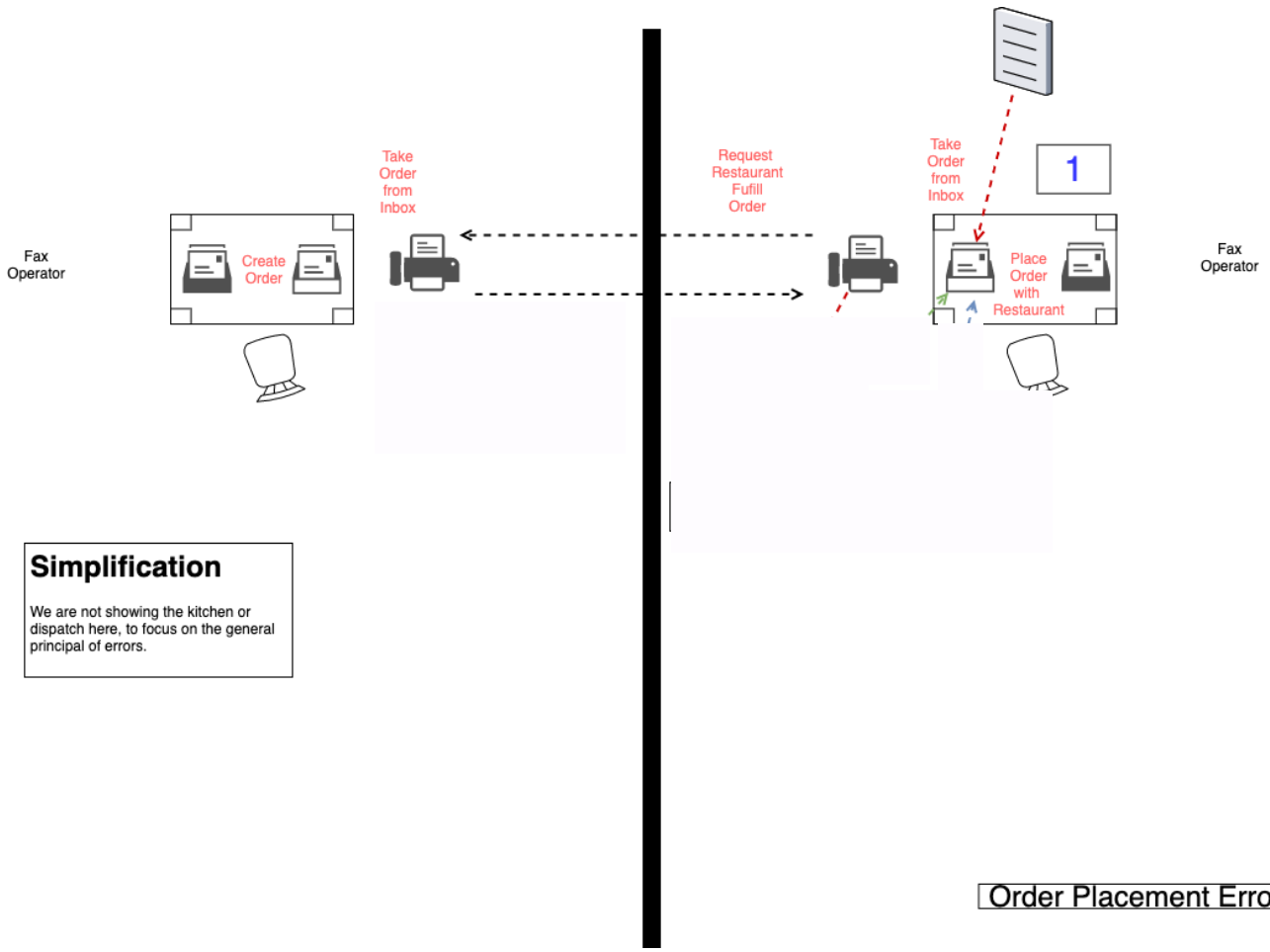
Fax Call Log

Monday, 2010-11-08 11:19

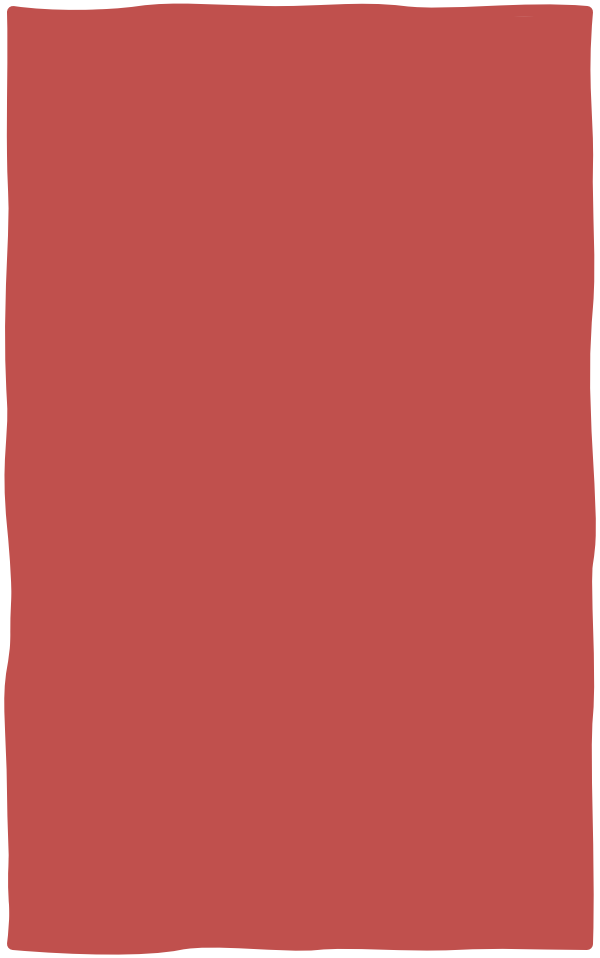
Date	Time	Type	Job #	Length	Speed	Station Name/Number	Pages	Status
2010-06-18	08:31	SCAN	92	0:25	28800	[illegible]	0	E-705 V.34 1M31
2010-03-22	11:39	RECV	59	0:20	26400	[illegible]	1	OK -- V.34 BM31
2010-03-22	11:45	RECV	60	0:20	26400	[illegible]	1	OK -- V.34 BM31
2010-03-22	12:29	RECV	61	0:21	26400	[illegible]	1	OK -- V.34 BM31
2010-09-14	14:46	SCAN	129	1:40	9600	[illegible]	2	E-606 V.29 AR30



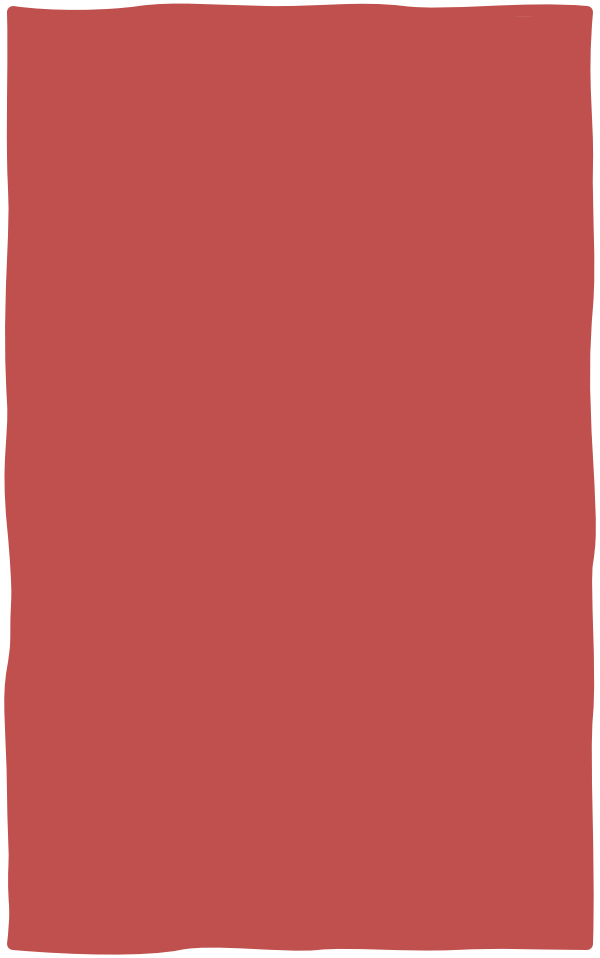
Order Taking Errors

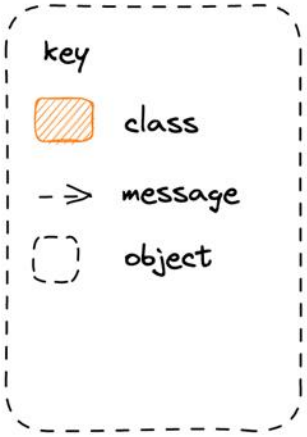

Fax
Operator
Create
Order
Take
Order
from
Inbox
Request
Restaurant
Fufill
Order
Take
Order
from
Inbox
1
Place
Order
with
Restaurant
Fax
Operator
Simplification
We are not showing the kitchen or dispatch here, to focus on the general principal of errors.
Order Placement Errors

REACTIVE PROGRAMMING

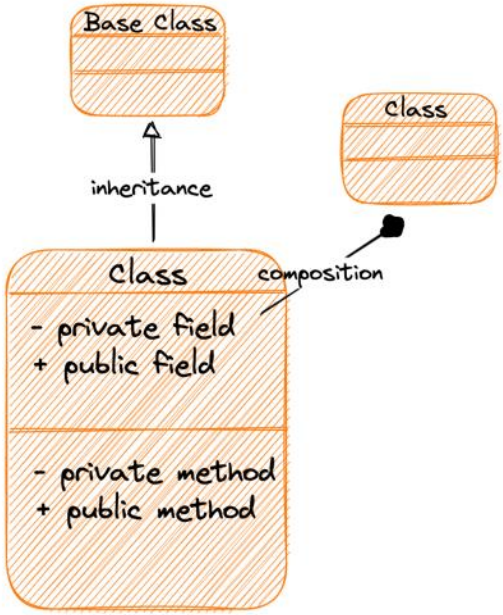


Object Oriented Programming





Object Orientation

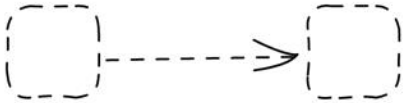


Data and Behaviour

A class couples data and the behaviour that depends upon that data. This allows encapsulation the data can be hidden and just behaviour exposed

Dynamic Dispatch

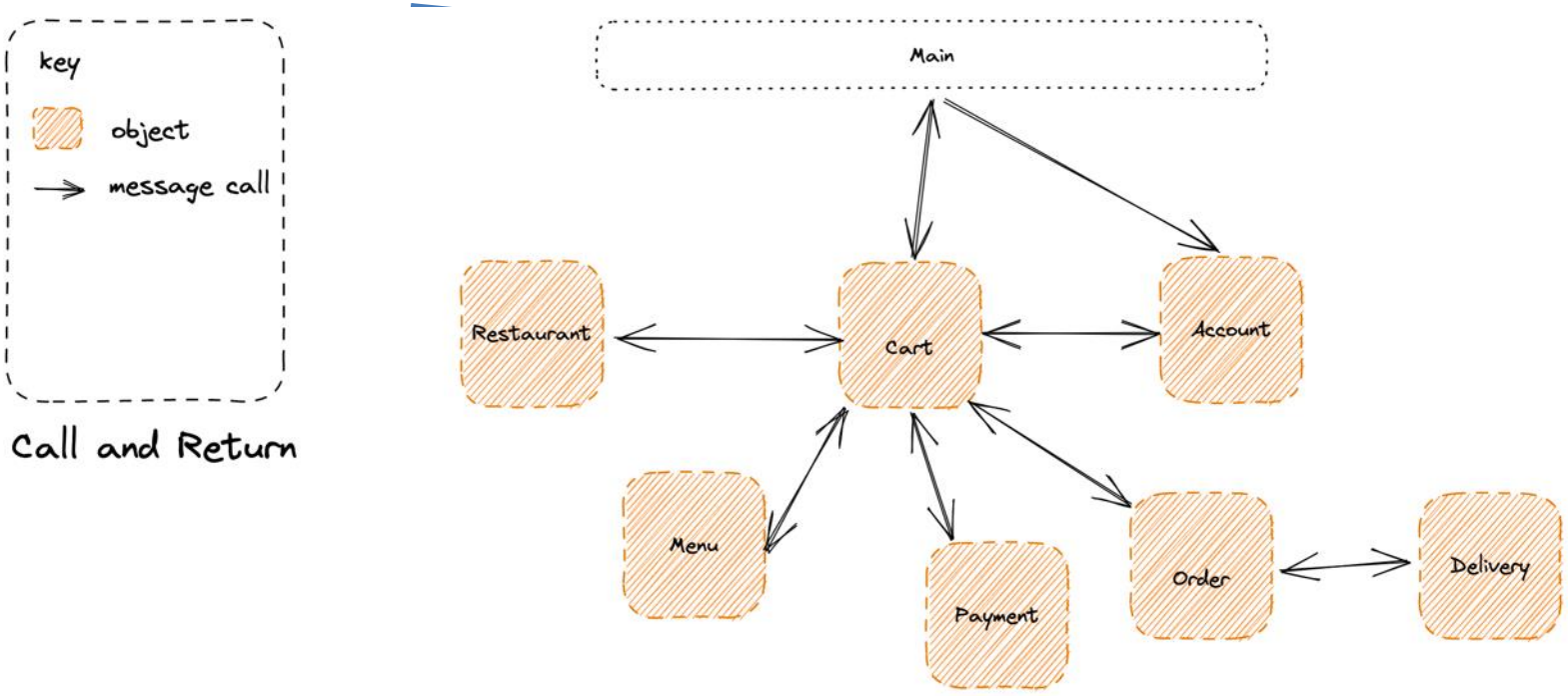
Due to inheritance the method chosen in response to a message may come from the base class of an object



Message Passing

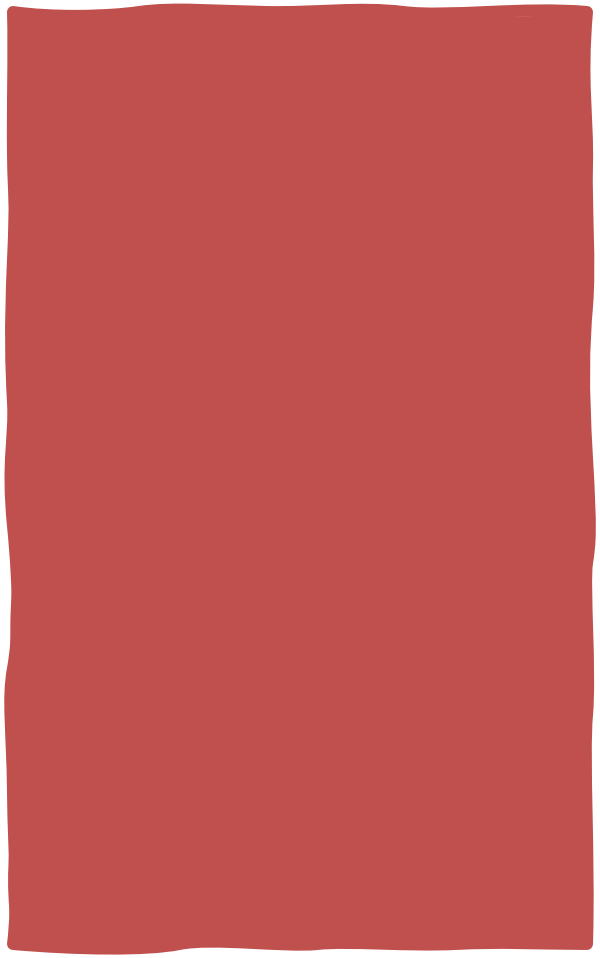
Objects communicate by message passing. A message is the method to call, and the parameters to that method.

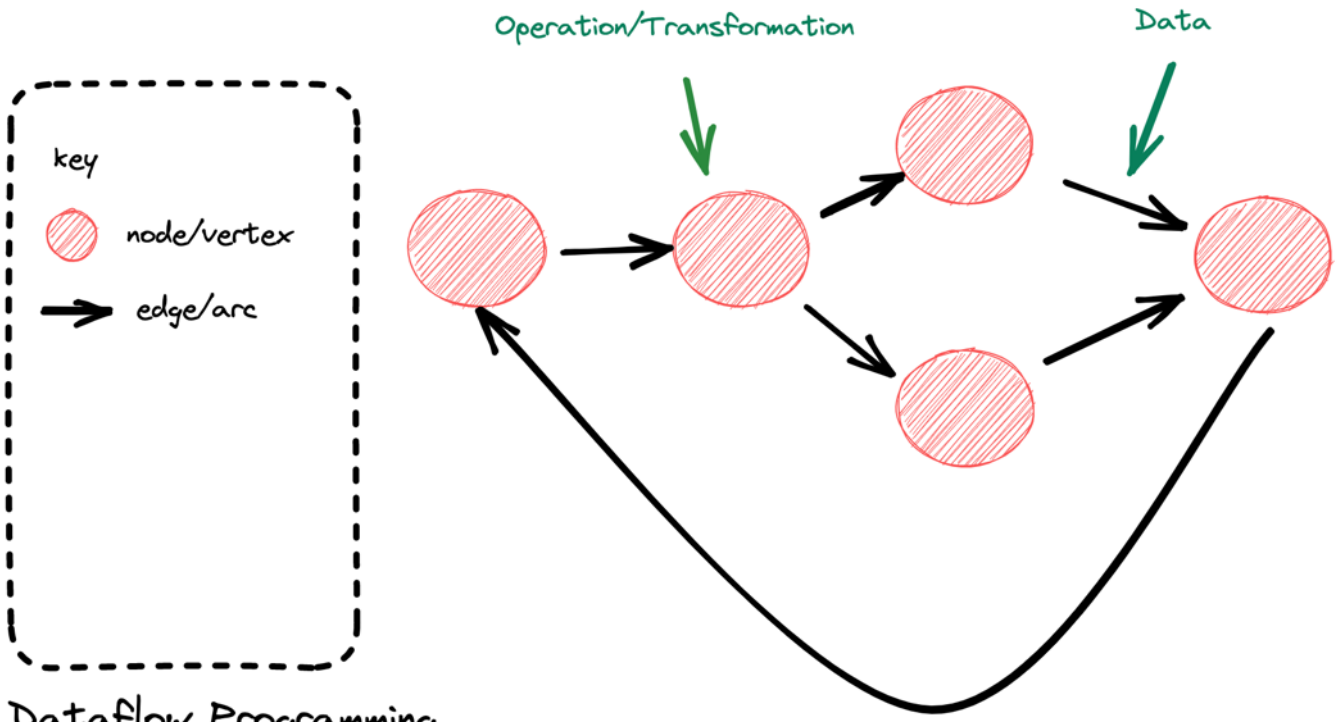
Call and Return: The main method represents our entry point and it uses message passing to invoke objects, which in turn invoke other objects, and return to their caller



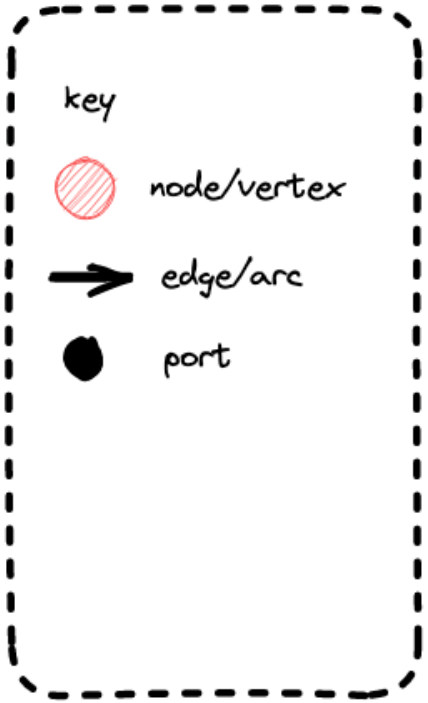
God Object: A danger here is that we end up with a ”god” object, such as Cart here, that controls all the other objects. This is a high-degree of behavioral coupling

Data Flow Programming



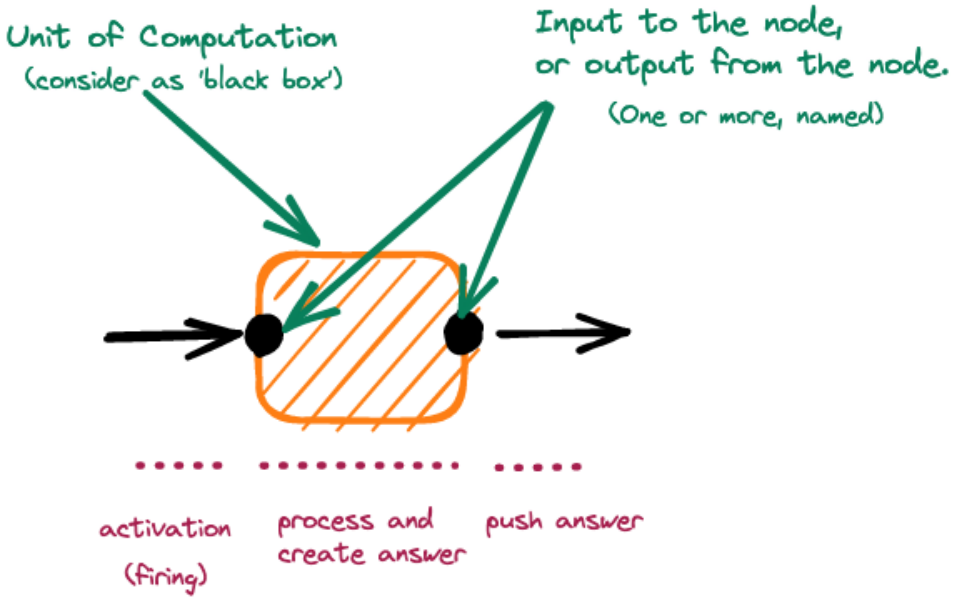


Data Flow Programming conceptualizes a program as a directed graph where operations are represented as nodes which are connected via arcs through which data flows. A node performs an operation when input data is available.



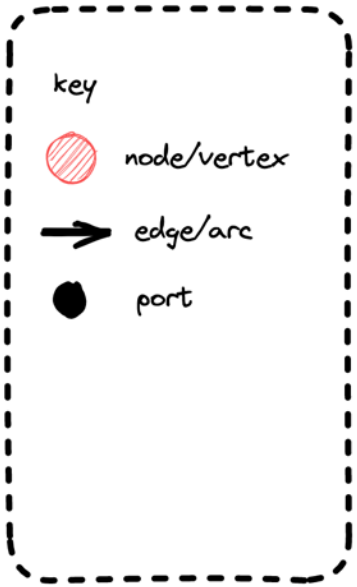
Dataflow Programming

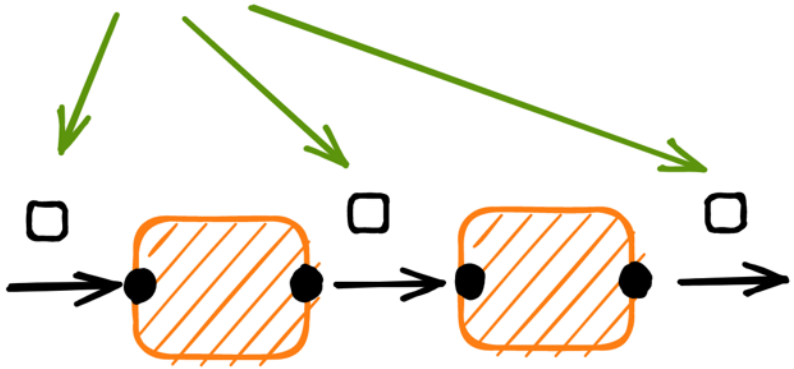
Jack Dennis et al.



Data Packet

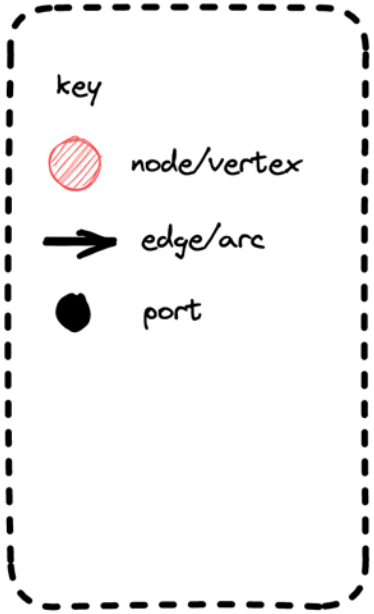
primitive/compound, structured/unstructured values, and even include other data packets

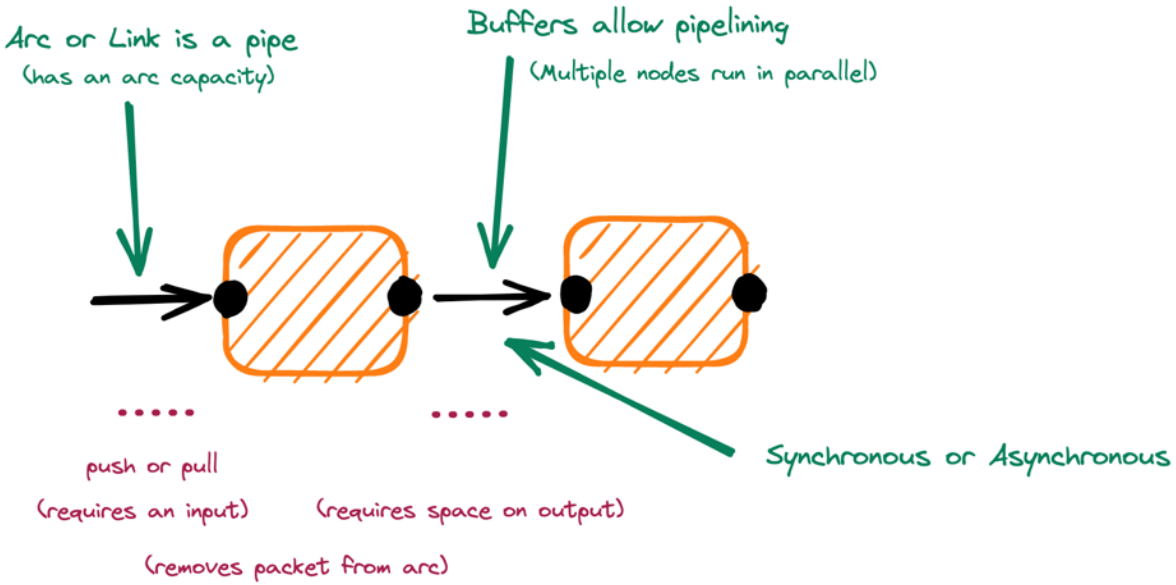




Dataflow Programming

Jack Dennis et al.





Dataflow Programming

Jack Dennis et al.

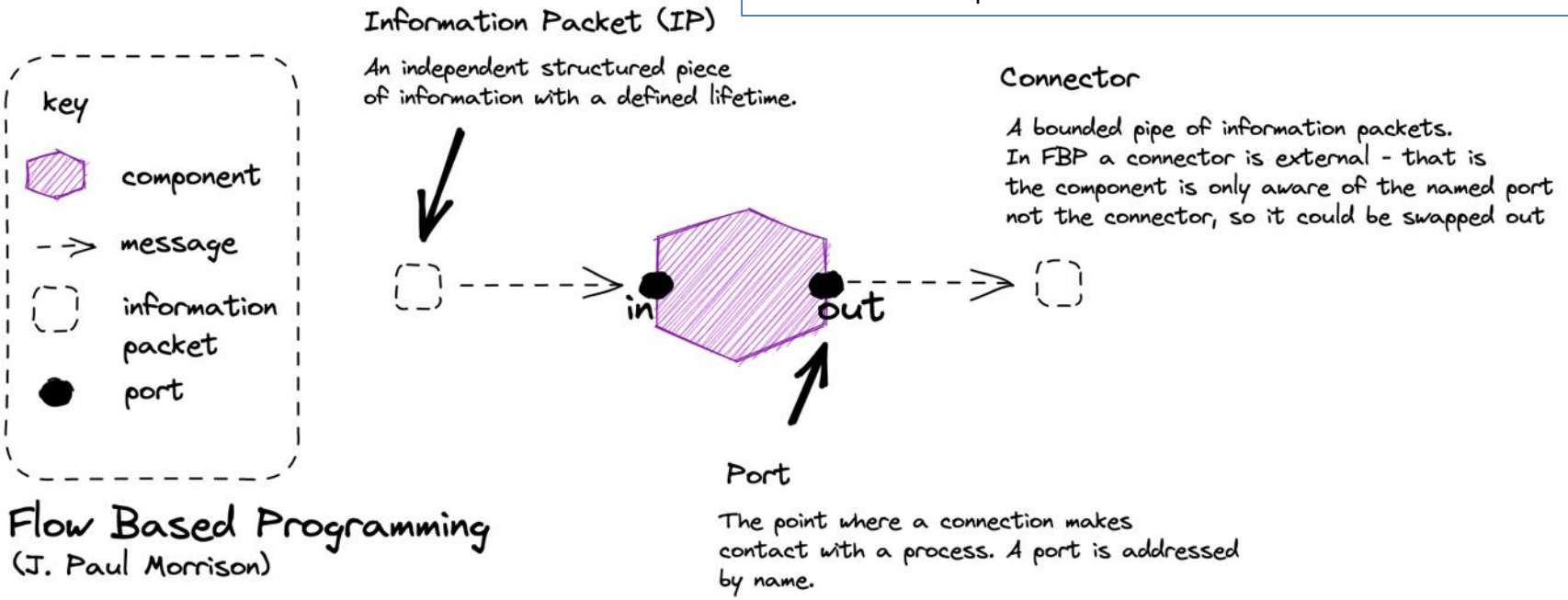
Node Lifetime: In 'classic' dataflow programming the lifetime of a node is from activation when there is work to do (push) or work is requested (pull) until it has pushed the answer.

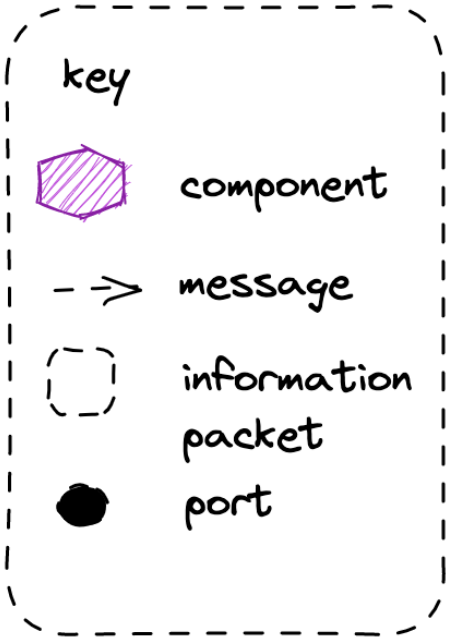
Capacity: In dataflow programming we do not activate a node to read from the input, if there is no space on the output. (Infinite capacity links exist only in theory). This creates **backpressure**.

Flow Based Programming

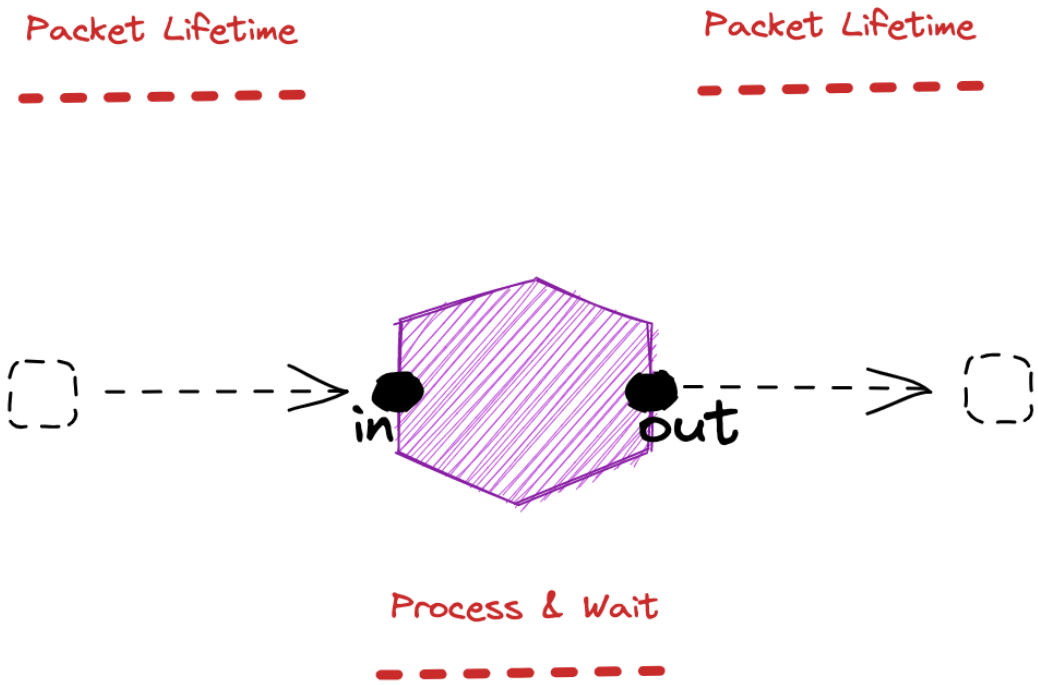
"Everything Flows and Nothing Stays." - Heraclitus

Flow-Based Programming (FBP) is a subclass of DFP. While DFP can be synchronous or asynchronous, FBP is always asynchronous. FBP allows multiple input ports, has bounded buffers and applies back pressure when buffers fill up.

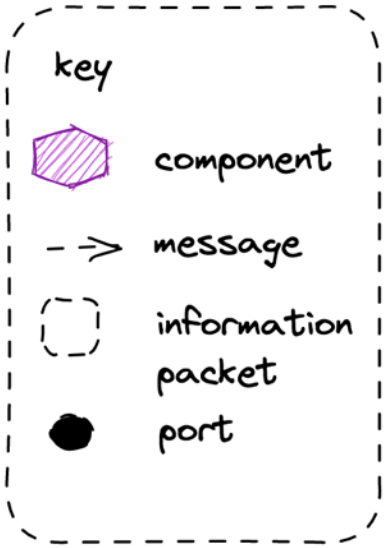




Flow Based Programming
(J. Paul Morrison)



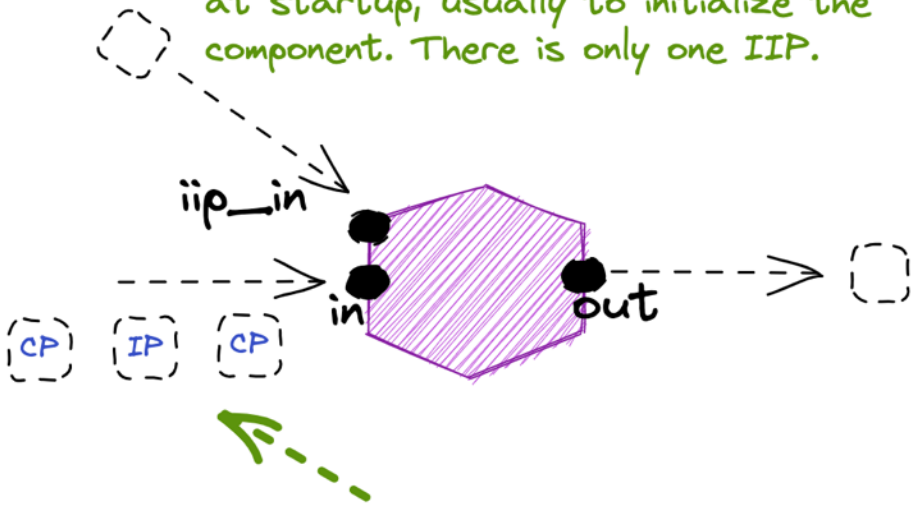
Node Lifetime: In flow base programming the a node can remain running whilst there is work on an input queue, can can suspend instead of terminate if there is no work on its connector.



Flow Based Programming
(J. Paul Morrison)

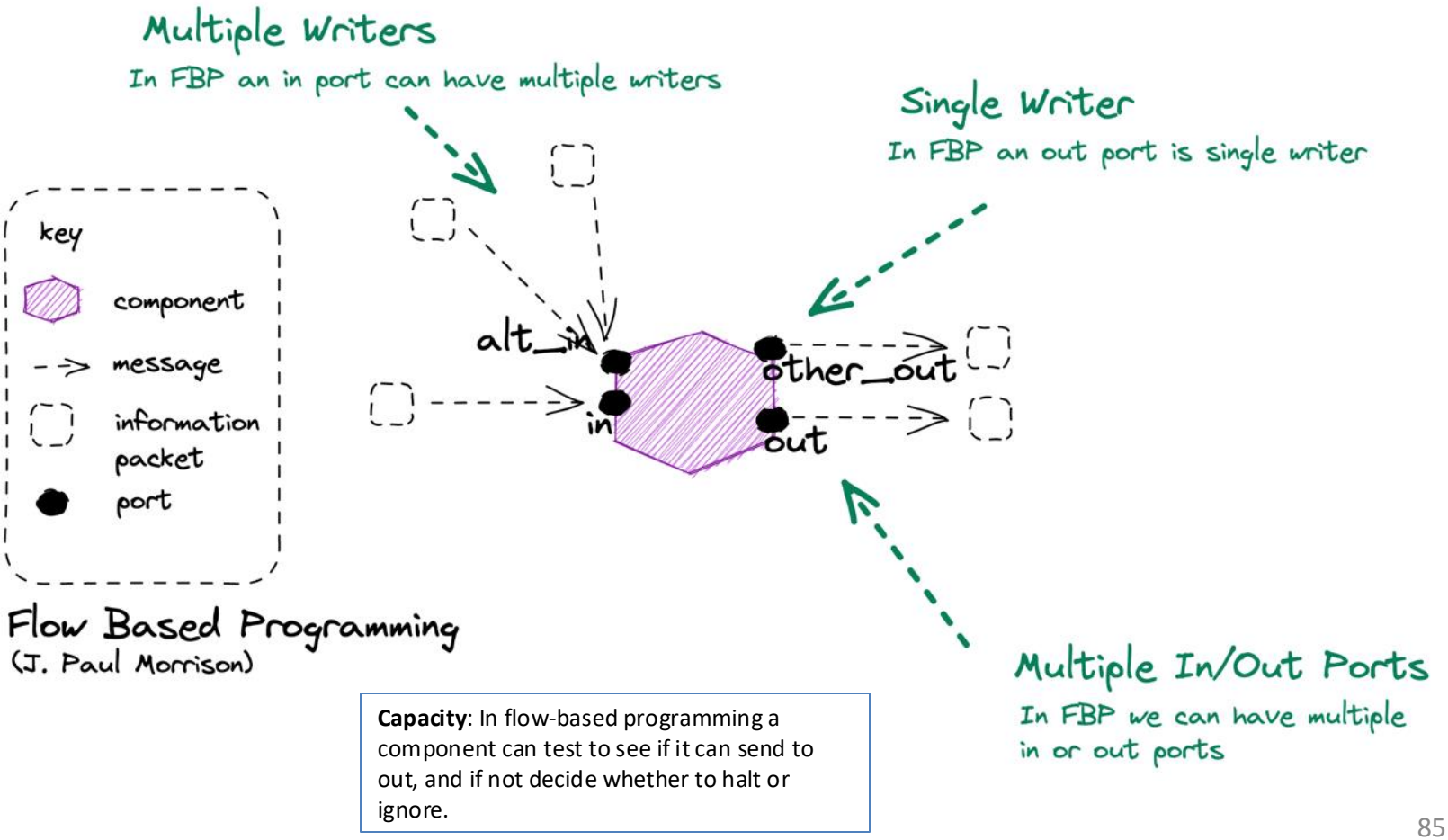
Initial Information Packet

A packet that the component reads at startup, usually to initialize the component. There is only one IIP.

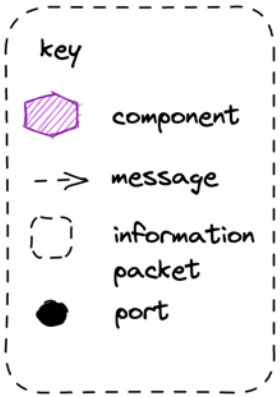


Control Packets

Control Packets can be used to 'bracket' groups

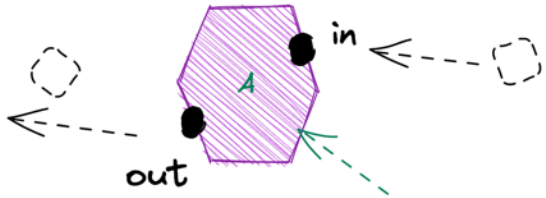


Capacity: In flow-based programming a component can test to see if it can send to out, and if not decide whether to halt or ignore.

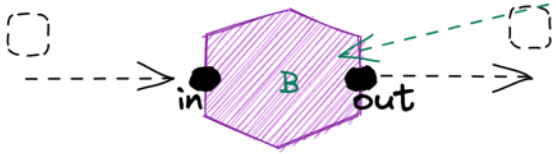


Flow Based Programming

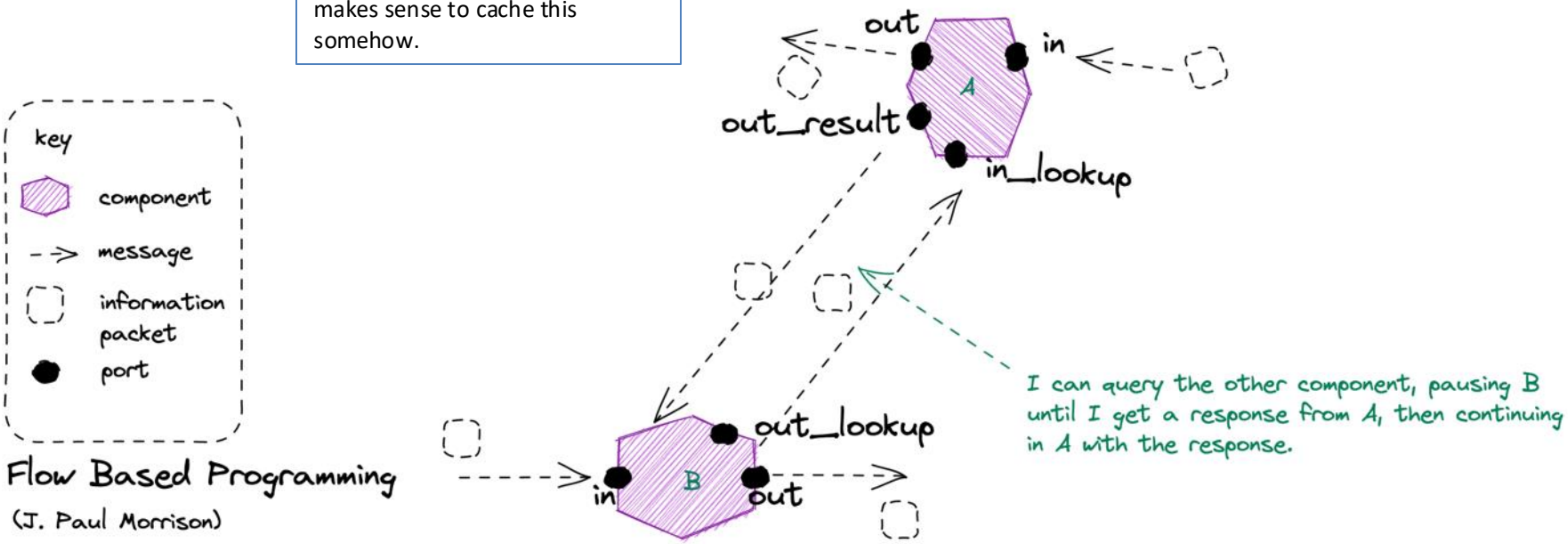
(J. Paul Morrison)

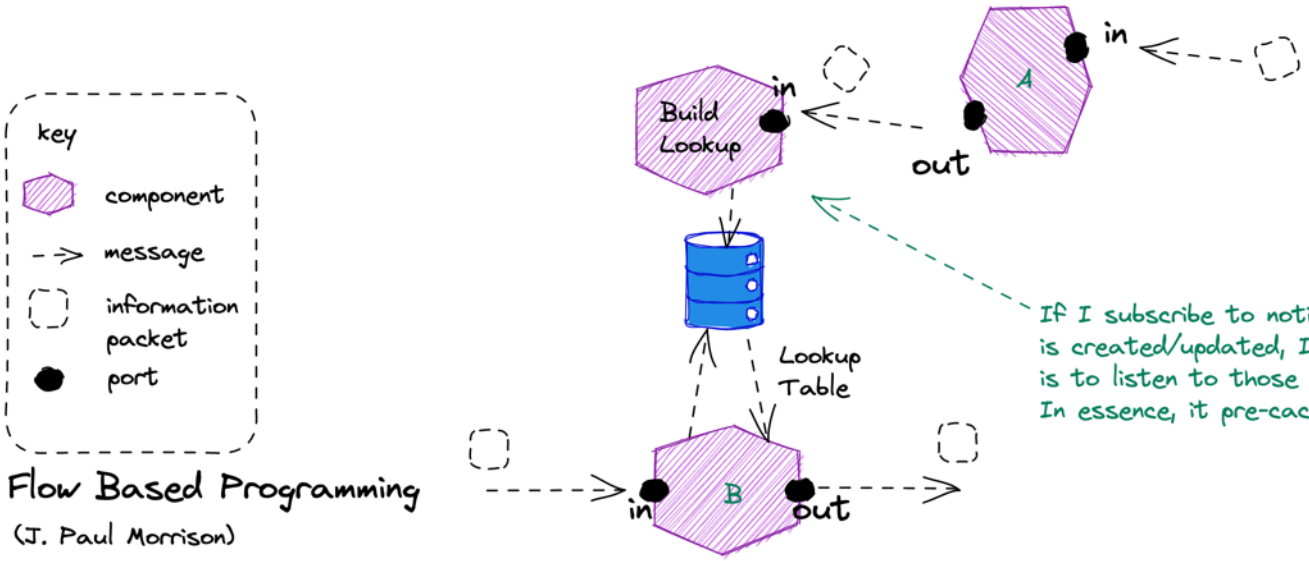


What happens if I need data from one node, to react in another node?



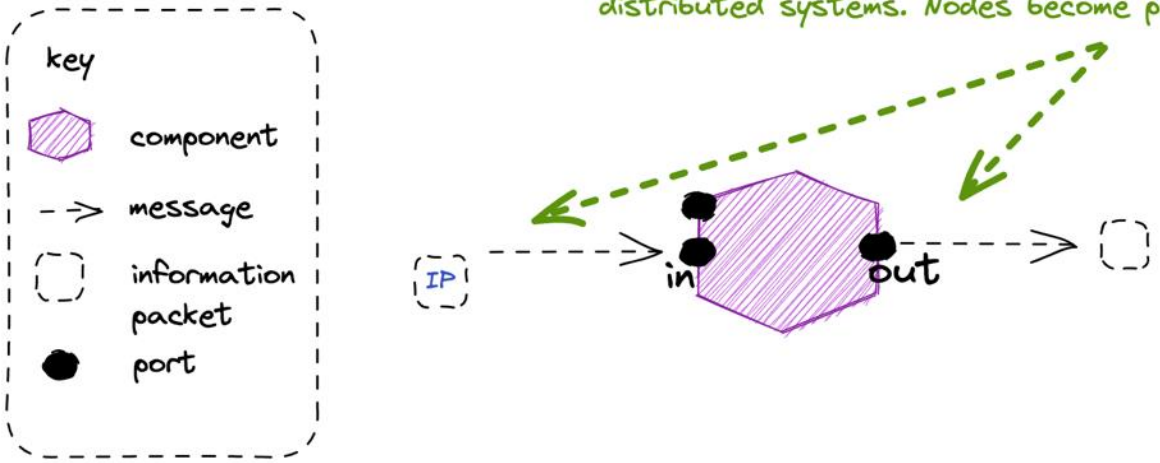
Walk of Shame: The trouble is that we may keep asking, so it makes sense to cache this somehow.




key
component
message
information packet
port
Flow Based Programming
(J. Paul Morrison)
Build Lookup
in
out
A
in
Lookup Table
in
B
out
If I subscribe to notifications from A, as new data is created/updated, I can write a node whose responsibility is to listen to those messages and builds a lookup table for B. In essence, it pre-caches the results for B.

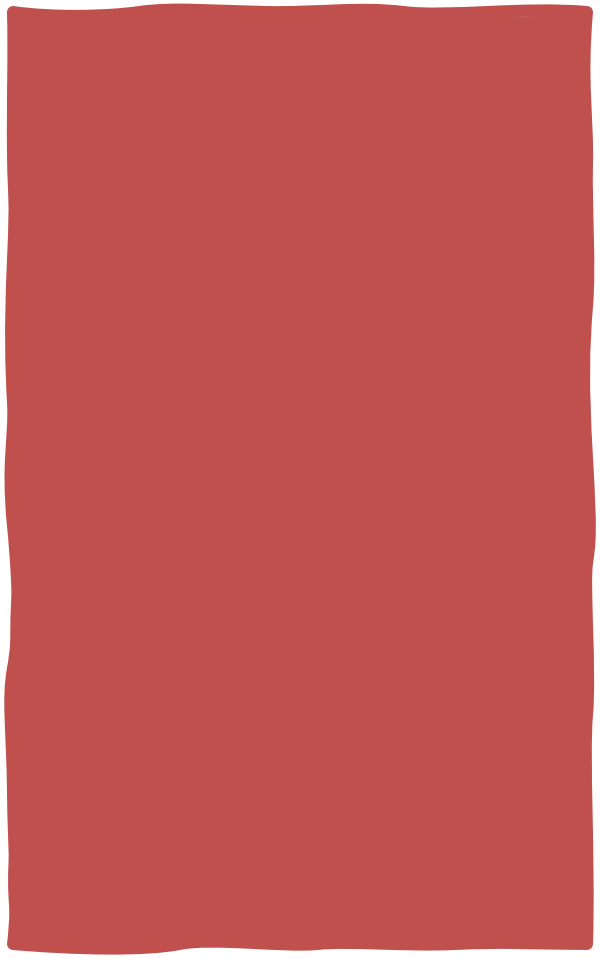
Message Oriented Middleware (MoM)

Because FBP allows for externally defined connectors
in the 2010 update JPM outlined that it could be used in
distributed systems. Nodes become processes and connectors use MoM.

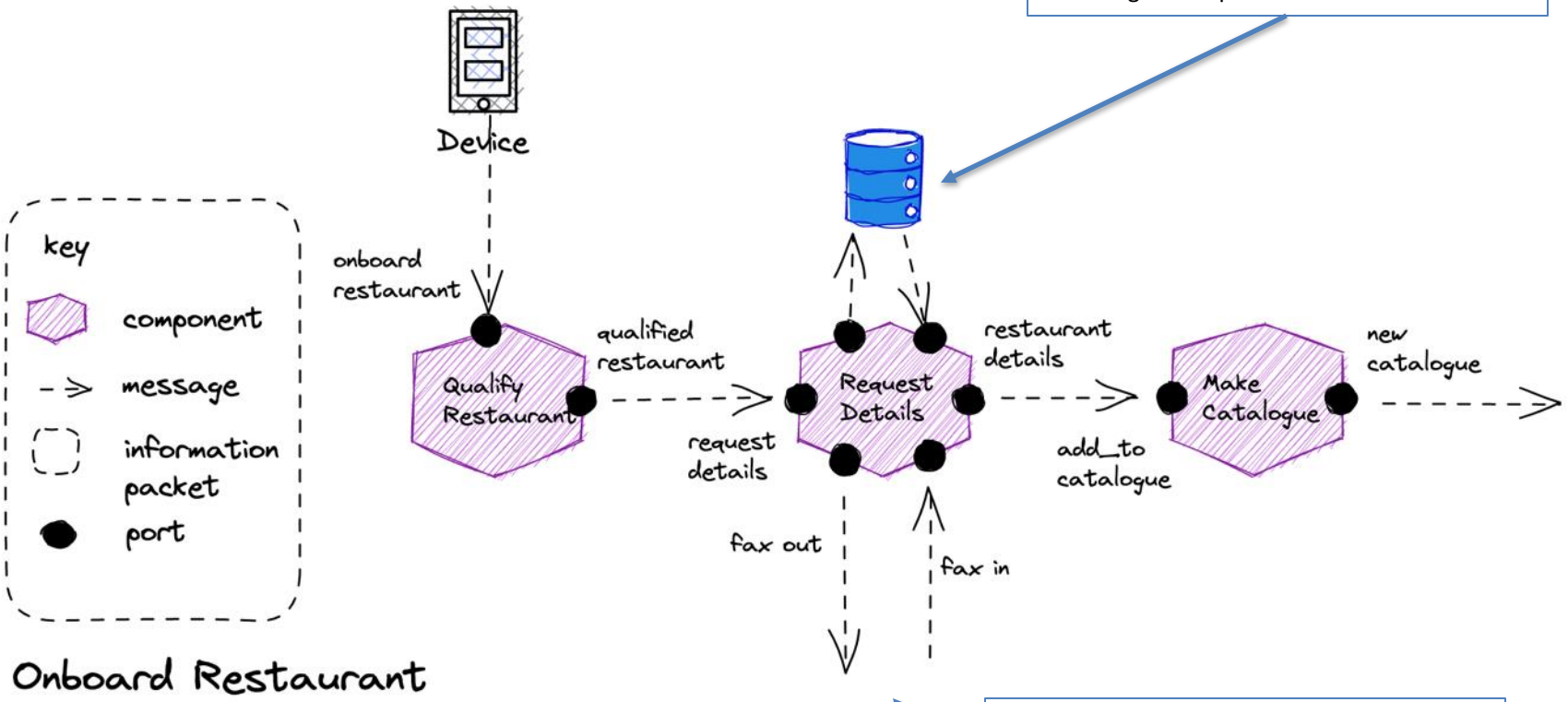


Flow Based Programming - 2010 Update
(J. Paul Morrison)

Example

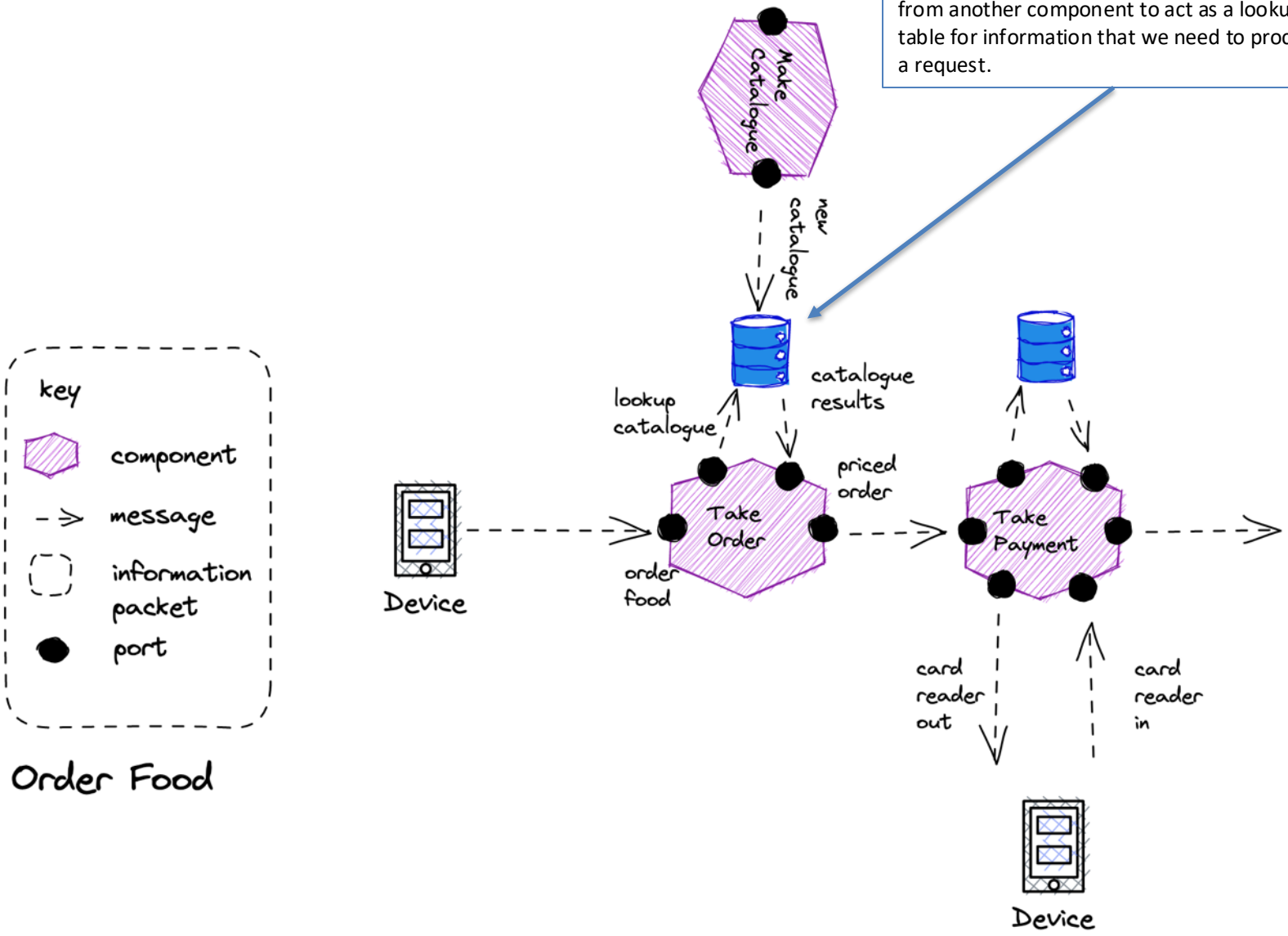


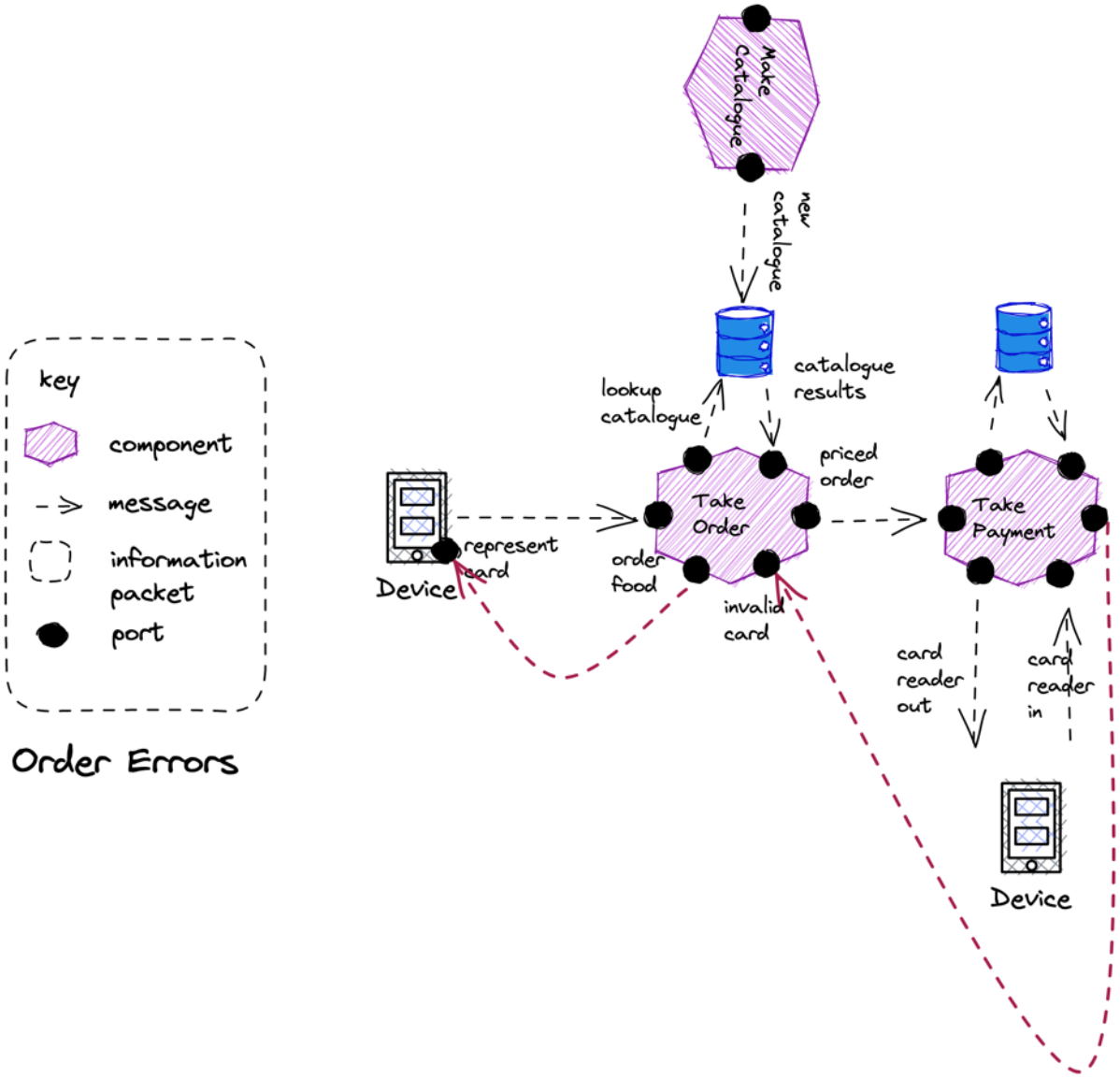
Storage: Request details may need storage to be reliable, it takes work from the **request_details** port, but cannot forward it to the **restaurant_details** port without waiting for an answer from **fax_in**. We store the state of the workflow in case we crash before receiving the response.

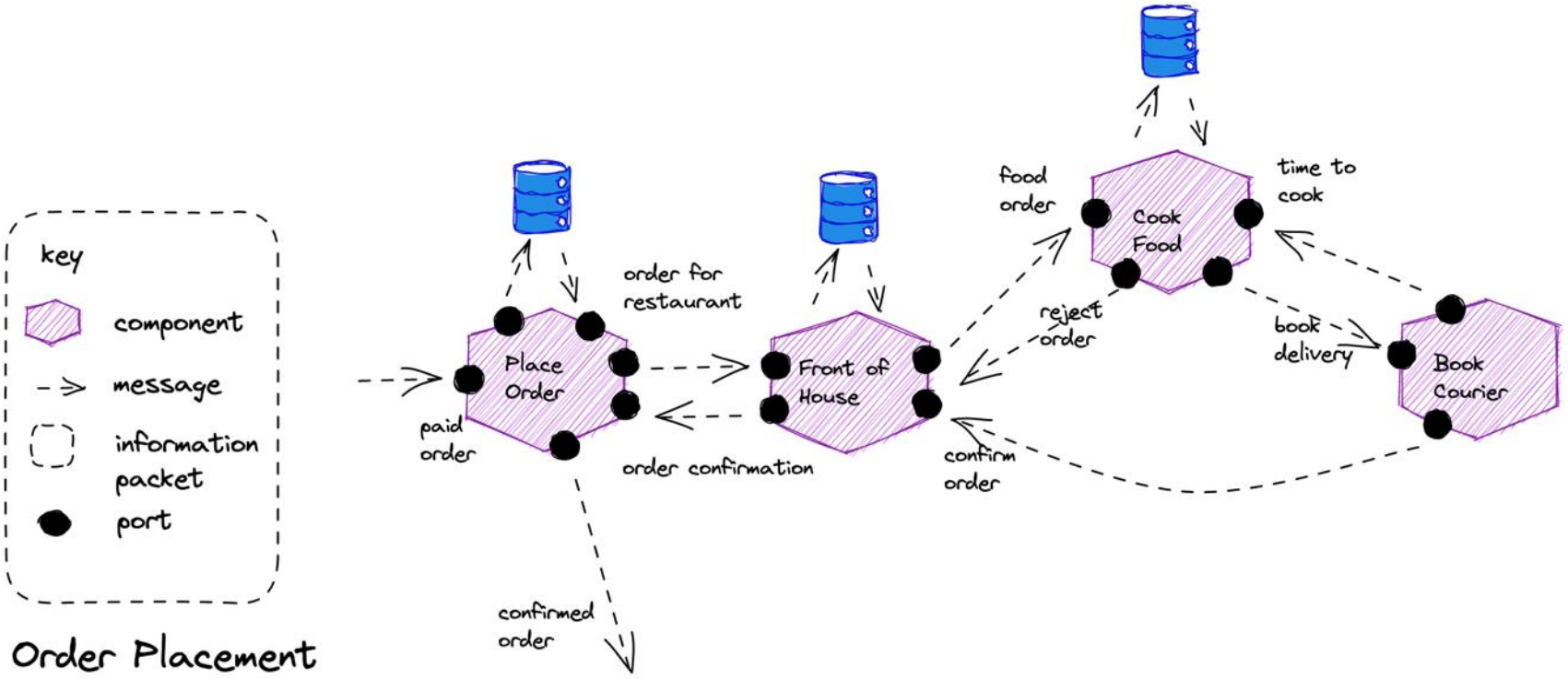


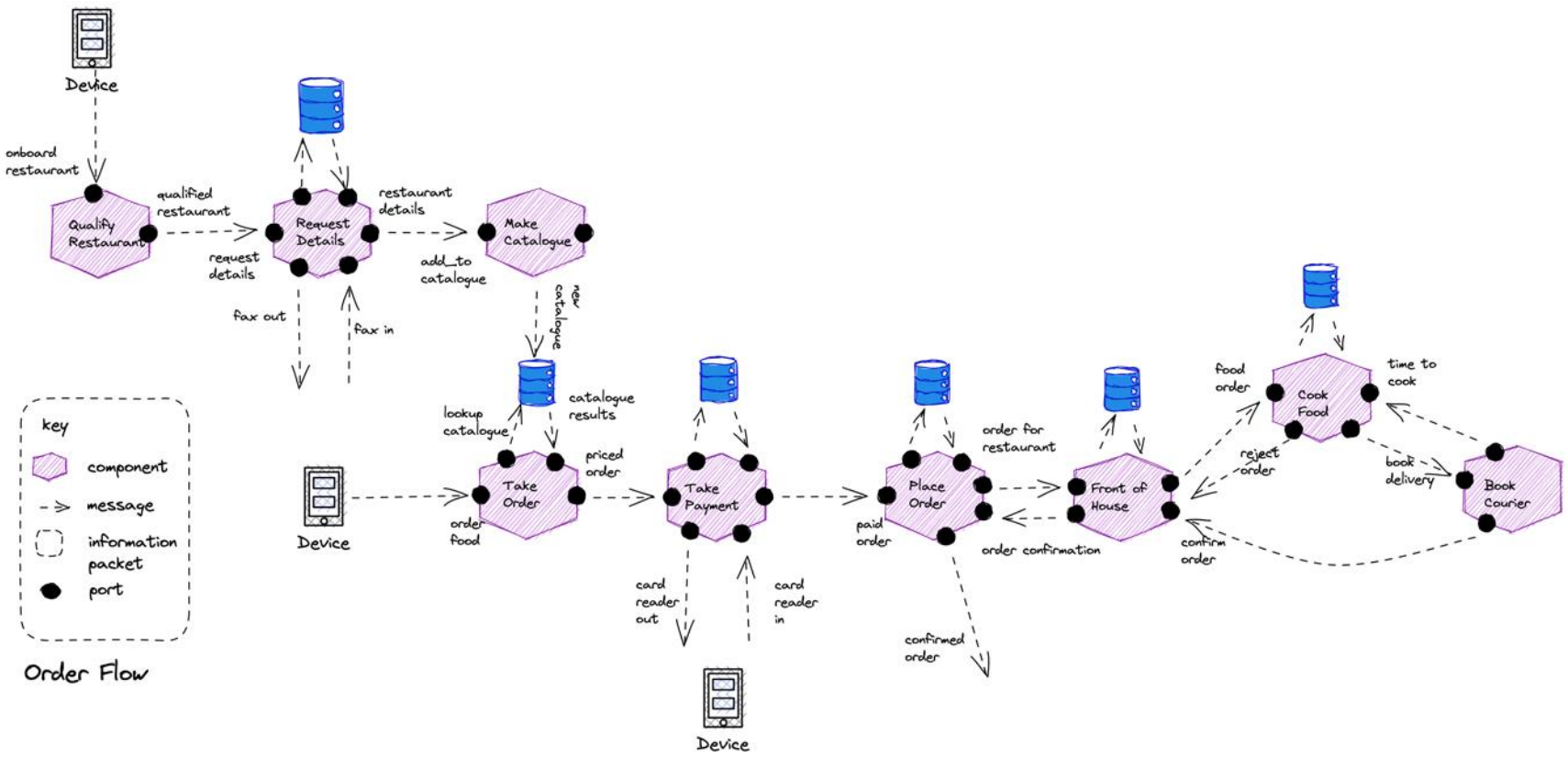
Correlation: To allow us to match the response to the saved state we use a *correlation id*. We send it on the ip to fax_out, and the restaurant includes it on the ip from fax_in. We use the ip to look up the workflow we stored.

Lookup: We can build a store from IP raised from another component to act as a lookup table for information that we need to process a request.

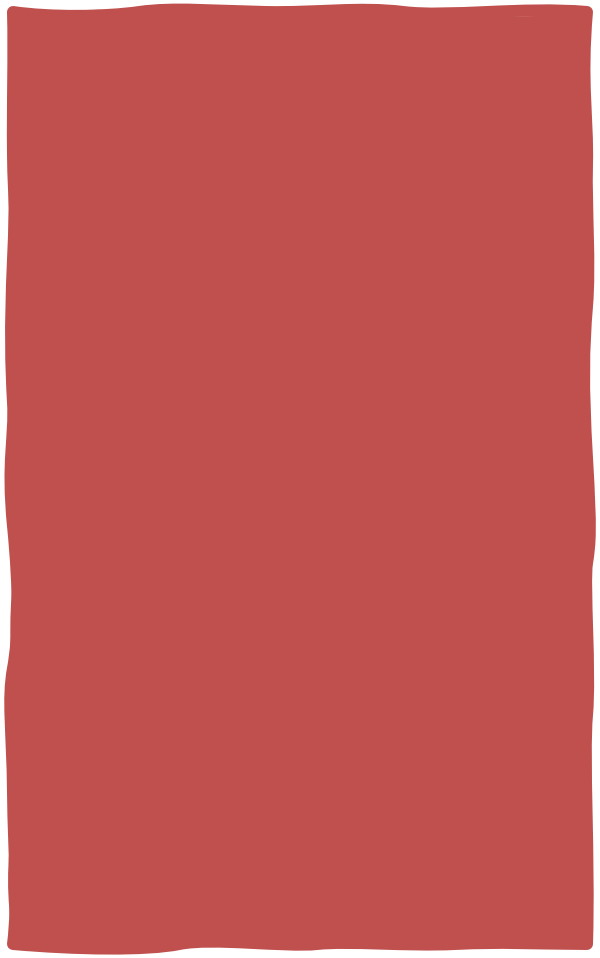


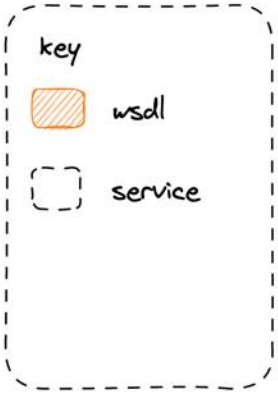

key
component
message
information packet
port
Order Errors
Make Catalogue
new catalogue
lookup catalogue
catalogue results
priced order
Take Order
Take Payment
represent card
Device
order food
invalid card
card reader out
card reader in
Device


key
component
message
information packet
port
Order Placement
Place Order
Front of House
Cook Food
Book Courier
order for restaurant
paid order
order confirmation
confirmed order
food order
time to cook
reject order
book delivery
confirm order


Device
onboard restaurant
Qualify Restaurant
qualified restaurant
Request Details
request details
restaurant details
Make Catalogue
add_to catalogue
new catalogue
fax out
fax in
lookup catalogue
catalogue results
Take Order
order food
priced order
Device
Take Payment
card reader out
card reader in
Device
paid order
Place Order
confirmed order
order for restaurant
order confirmation
Front of House
food order
Cook Food
time to cook
reject order
book delivery
Book Courier
confirm order
key
component
message
information packet
port
Order Flow

SOA Architectures

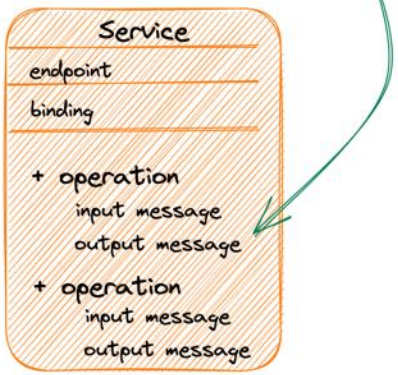




Service Orientation

Messages

A message is the parameter to an operation or the return type from it

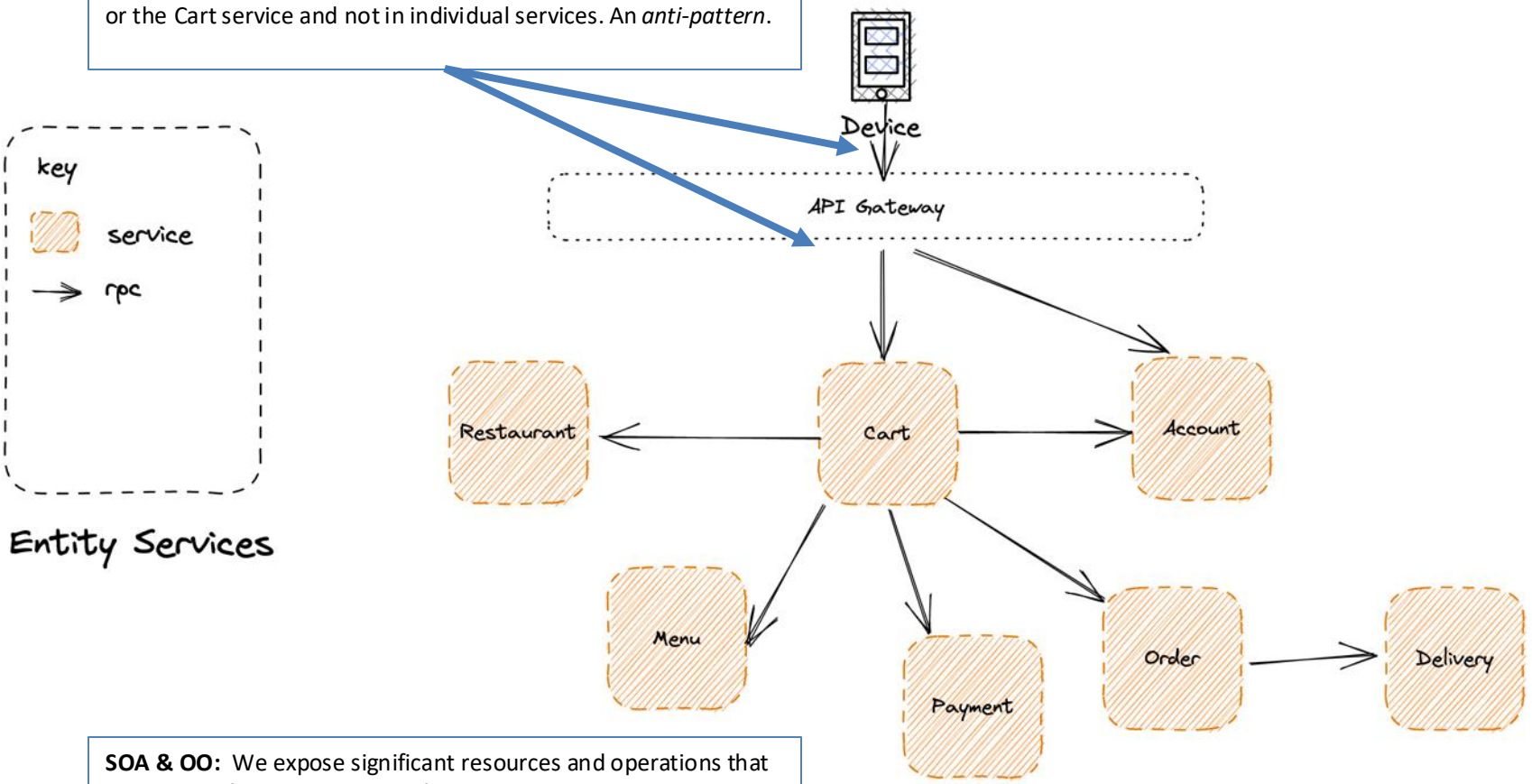


Data and Behaviour

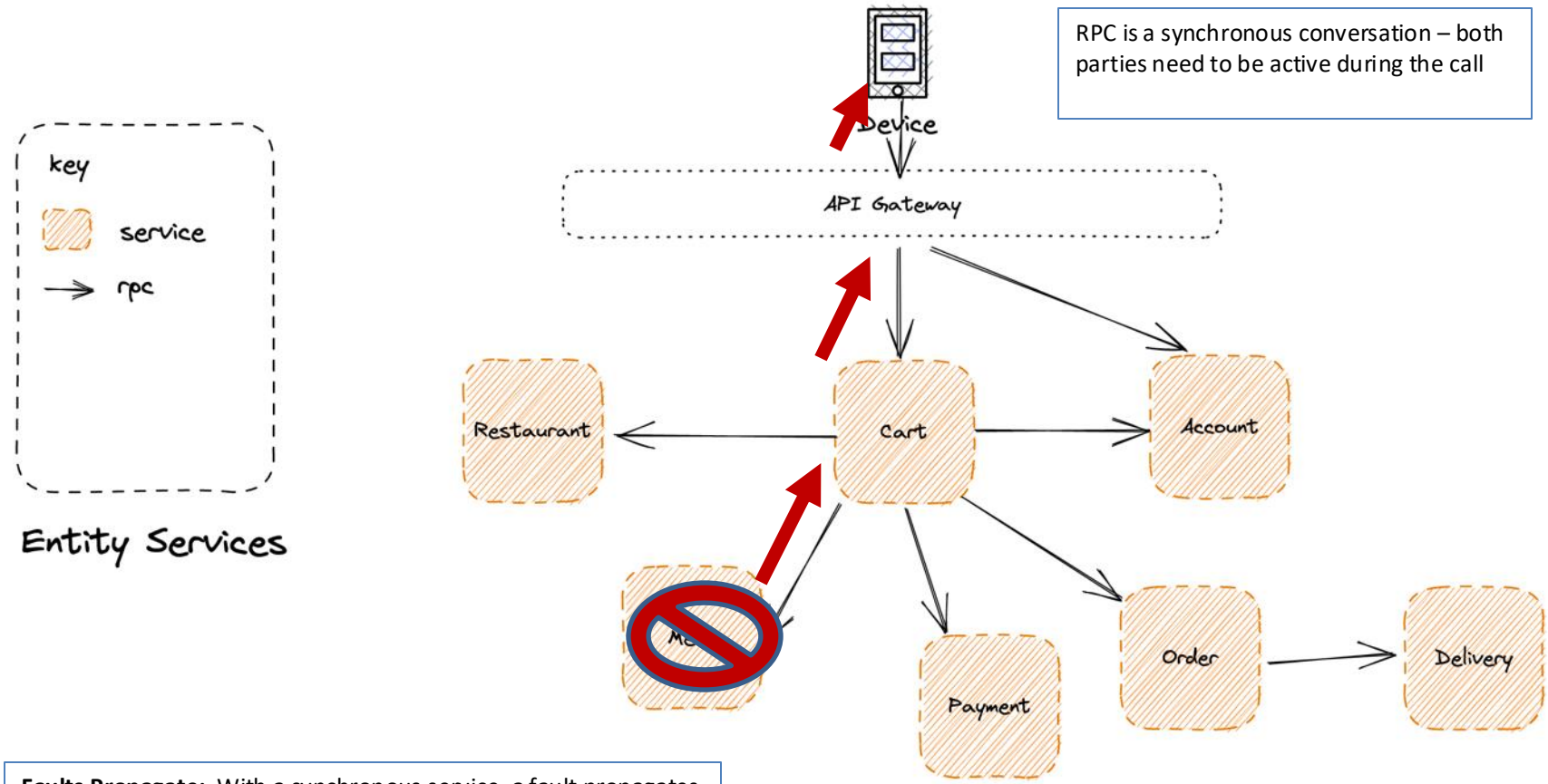
A service couples data and the behaviour that depends upon that data. This allows encapsulation the data can be hidden and just behaviour exposed

SOA & OO: SOA creates OO-like components, they encapsulate their data and expose behavior that is coupled to that data. This is the Web Services approach to services.

Feature Envy: Because domain logic needs to co-ordinate across these resources, it ends up either on the client, the API Gateway, or the Cart service and not in individual services. An *anti-pattern*.



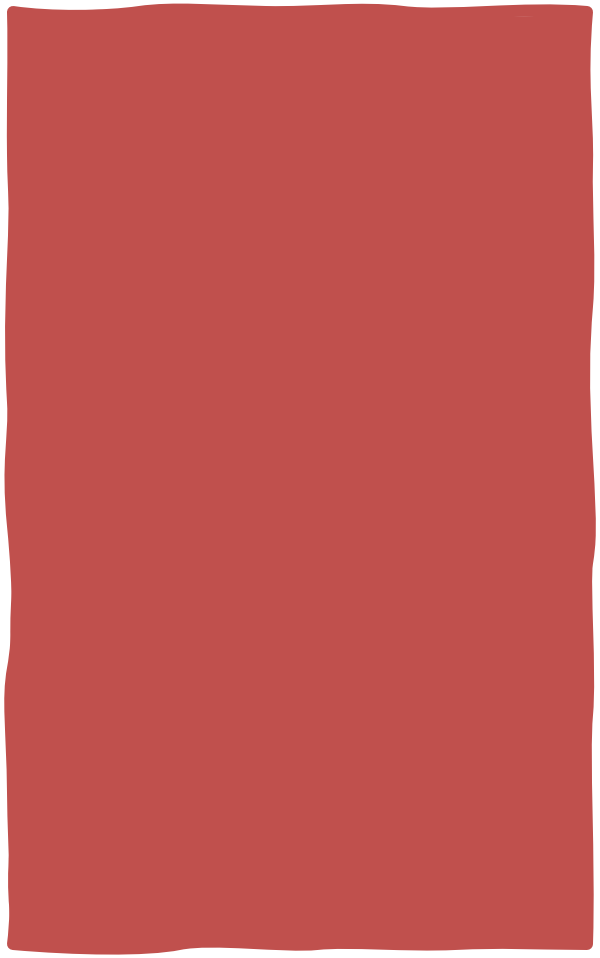
SOA & OO: We expose significant resources and operations that you can perform upon them, often CRUD.



RPC is a synchronous conversation – both parties need to be active during the call

Faults Propagate: With a synchronous service, a fault propagates

Reactive Architectures



The Reactive Manifesto

Published in September 2014
Author(s): Jonas Bonér (Erik Meijer, Martin Odersky, Greg Young, Martin Thompson, Roland Kuhn, James Ward and Guillaume Bort)

Defines an architectural style: **Reactive Applications**

Write applications that:

react to events: their event-driven nature enables the following qualities

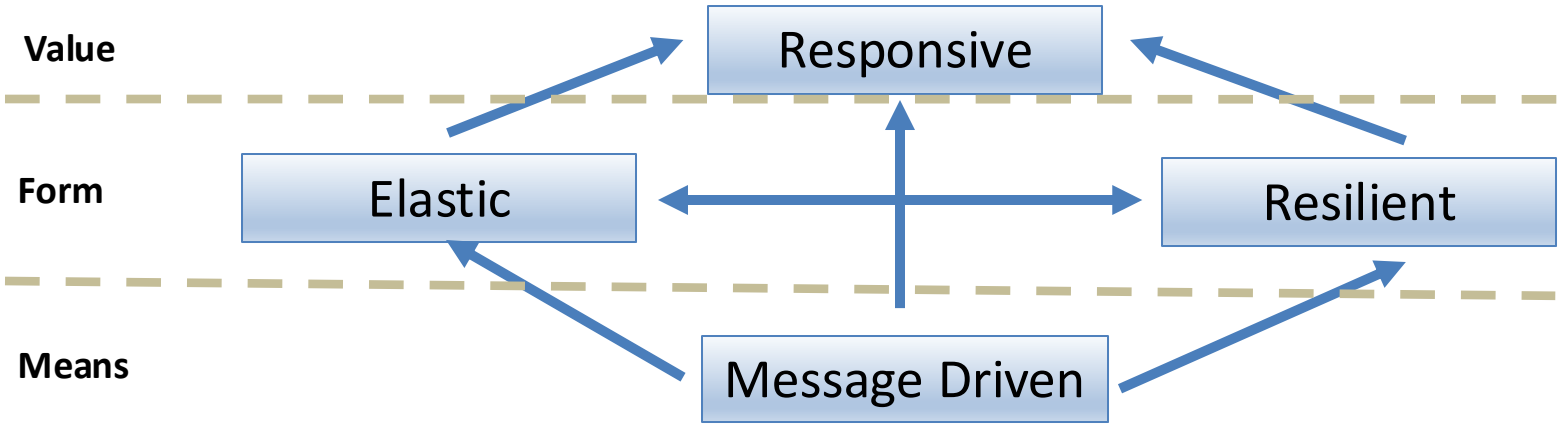
react to load: focus on scalability rather than single-user performance

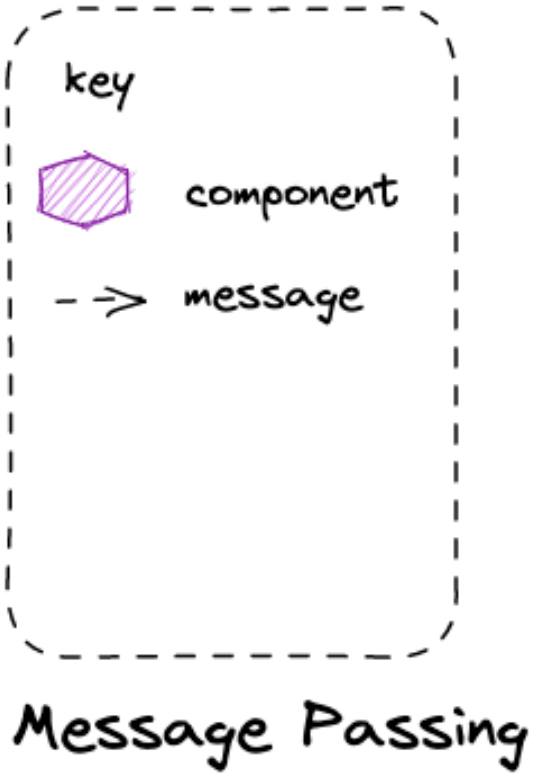
react to failure: resilient systems with the ability to recover at all levels

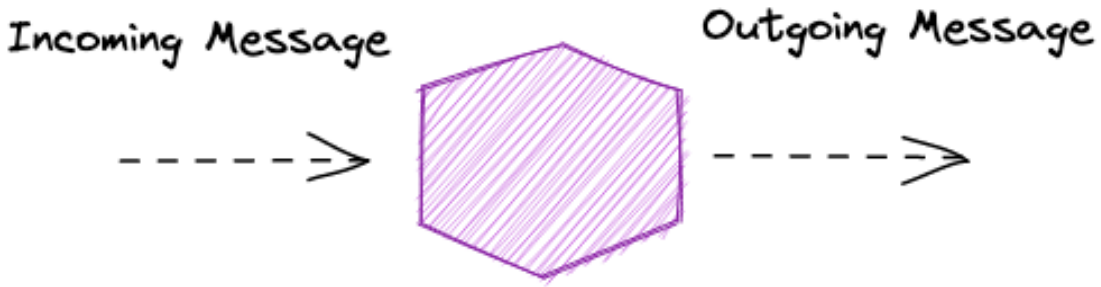
react to users: combine the above for an interactive user experience

Responsive: The system responds in a timely manner.
Resilient: The system stays responsive in the presence of failure.
Elastic: The system stays responsive under varying workload.
Message Driven: Rely on asynchronous message passing

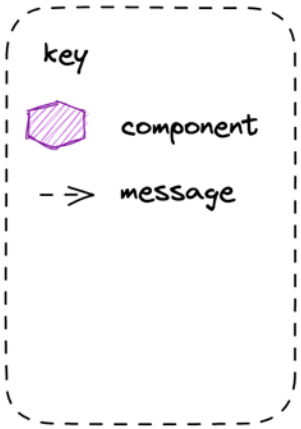
https://www.reactivemanifesto.org/


Value
Form
Means
Responsive
Elastic
Resilient
Message Driven



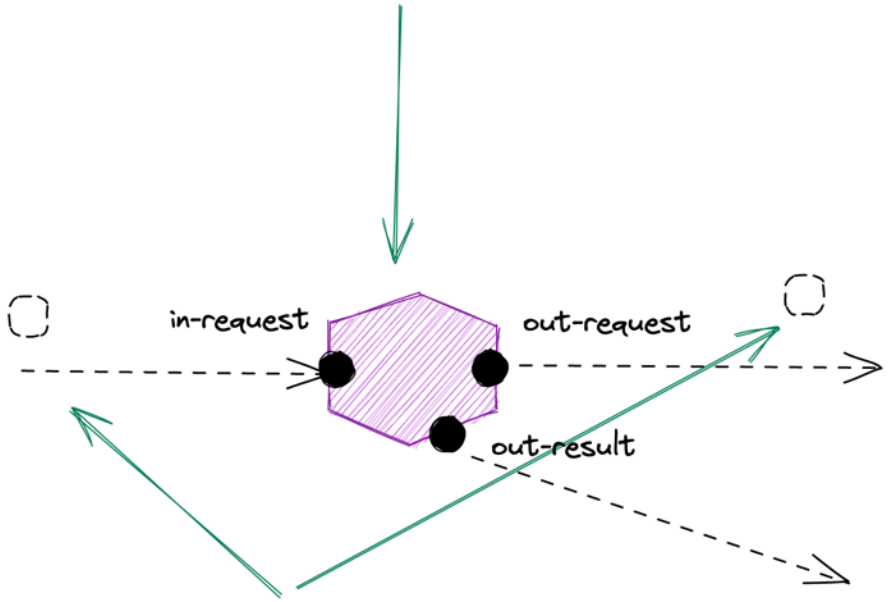


Message Passing is an asynchronous method of communication – both parties do not have to be simultaneously present for communication to occur, instead mail is delivered to 'mailbox' of some form for later retrieval.



Message Passing

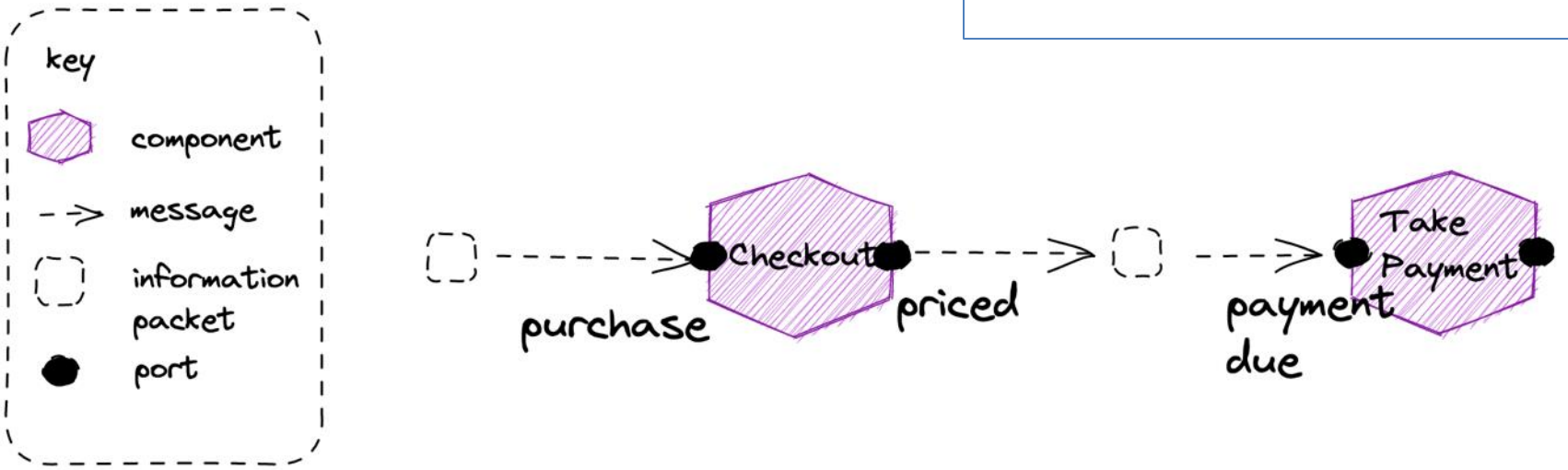
Service should focus on a transformation to the data
Usually a verb + noun combination
Place Order or Onboard Restaurant



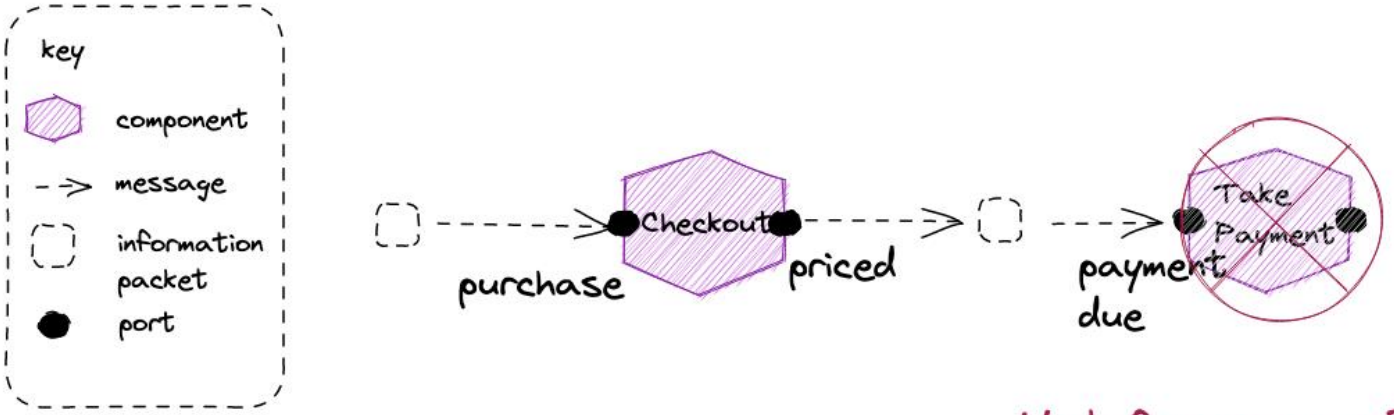
Things I am being told, a command

Things I know, facts, an event

Coordinate Dataflow A reactive *principle* where we "orchestrate a continuous steady flow of information" focusing on division by behavior, not structure



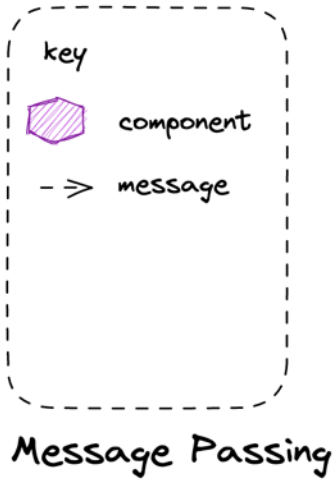
Partitioning Partition to take advantage of parallelism in the system, tasks that could be done concurrently with each other

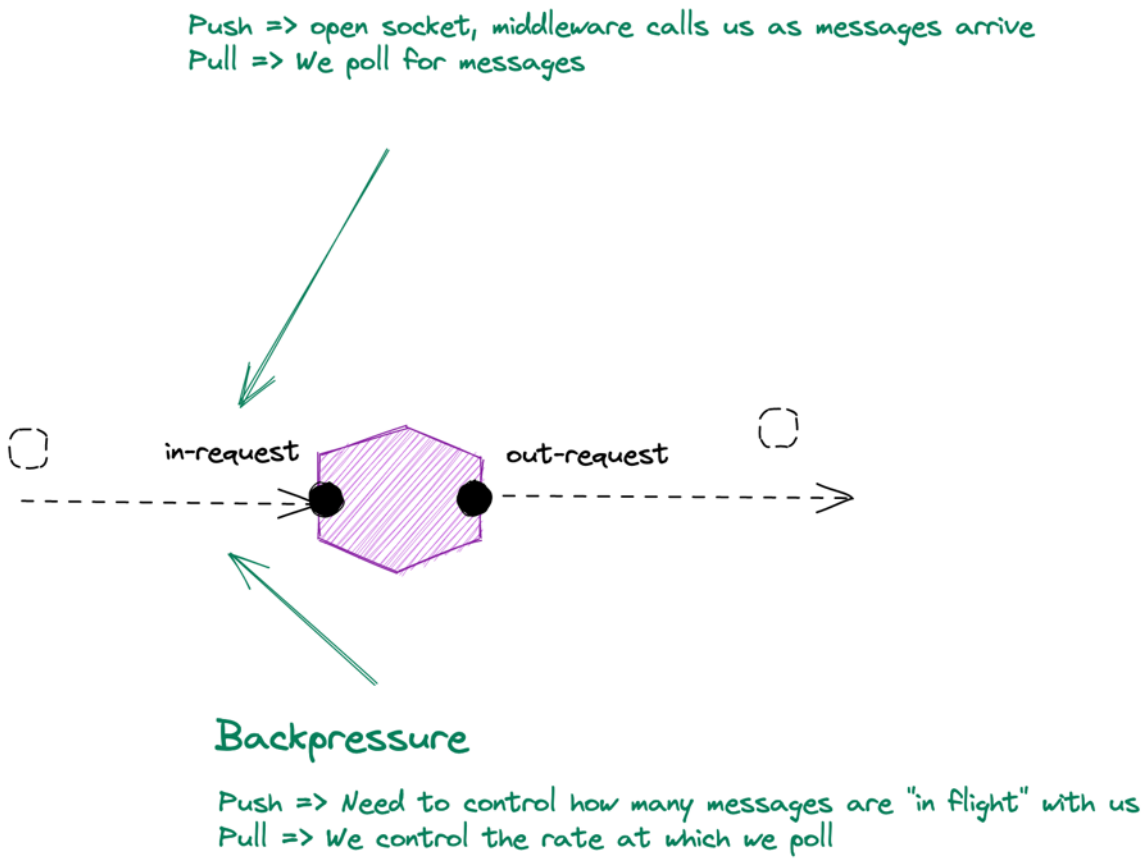


Work Queues on a Fault

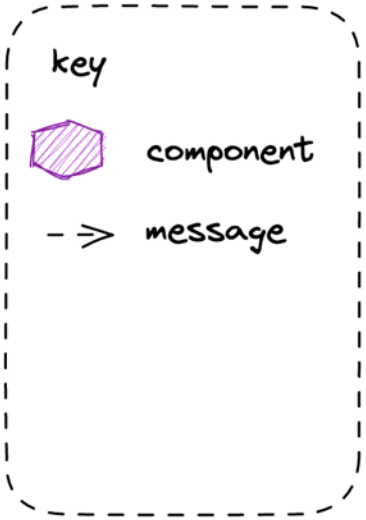
If Take Payment is not available, we can queue requests. When it starts up, Take Payment can process those requests and respond as normal.

Bulkheads: In an asynchronous conversation a fault does not propagate back up the chain – we have a bulkhead that protects us against failure

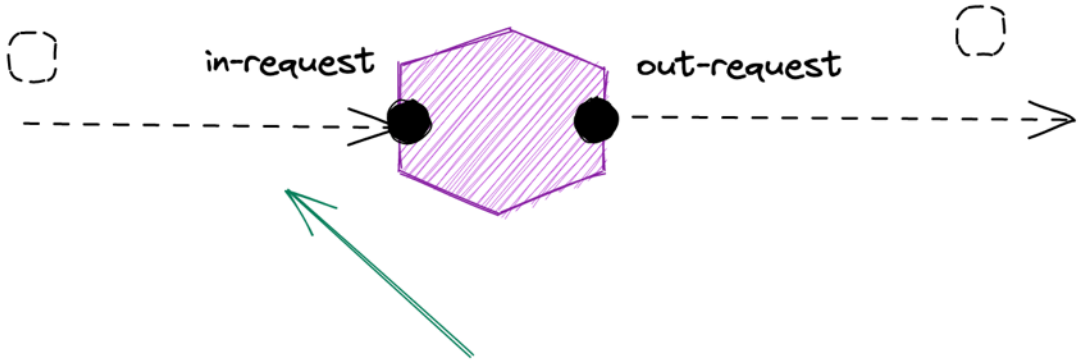




Blocking Retry: Blocking Retry – when we keep trying to process a message, such as when we cannot connect to a DB and retry, creates backpressure as we slow consumption.

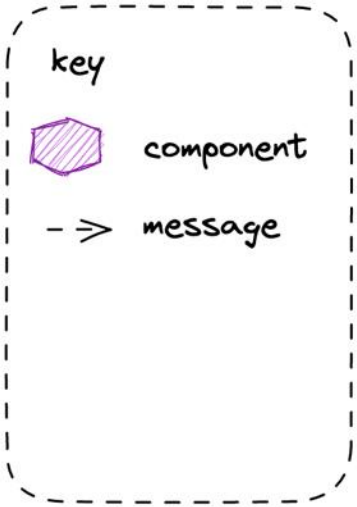


Message Passing

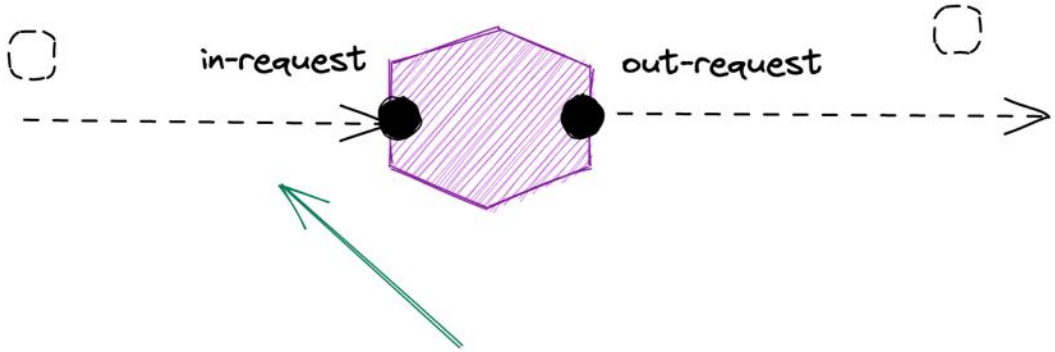


Load Shedding

Sometimes we may prefer to simply drop messages or shed load, when there is a fault.

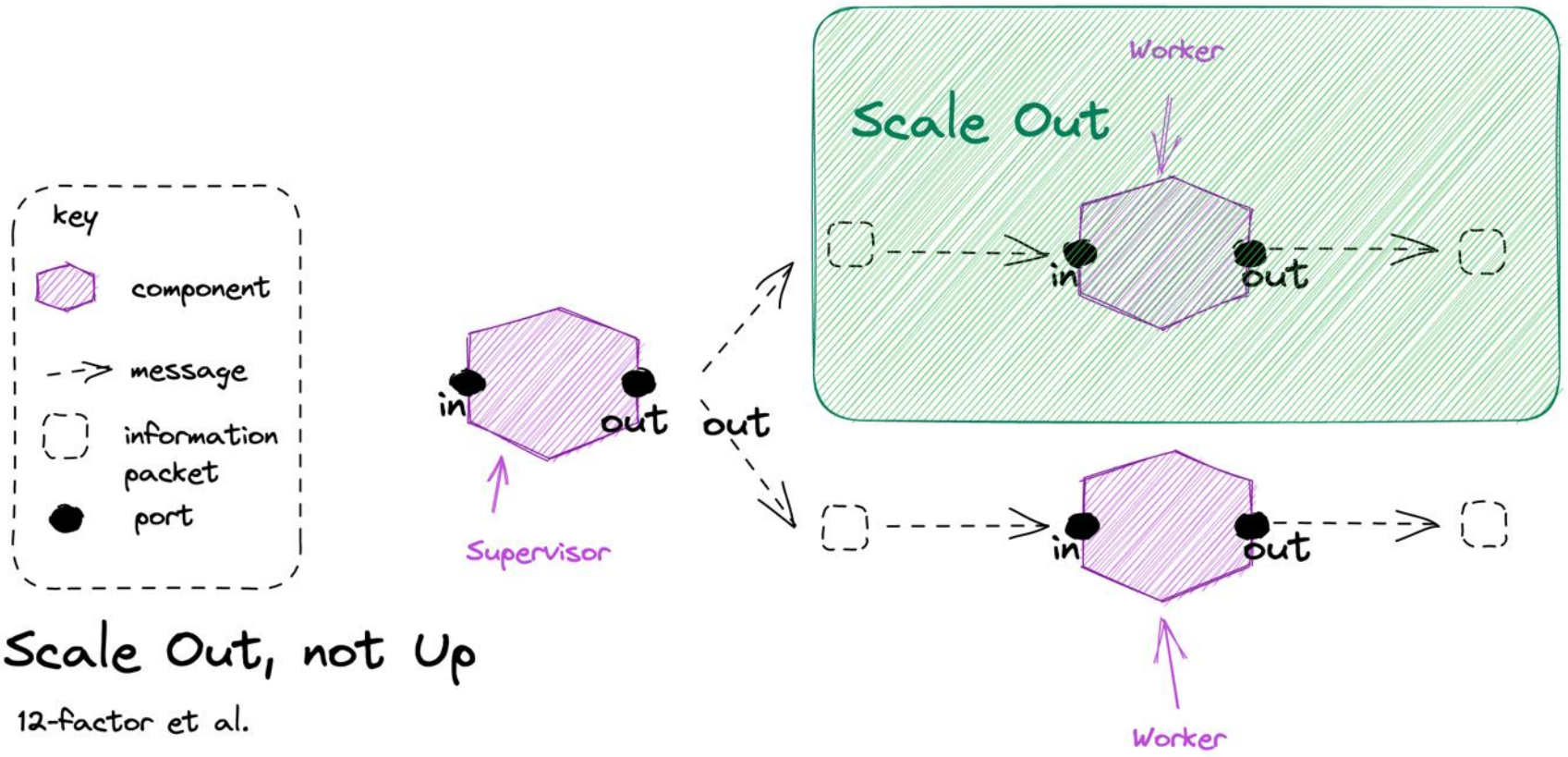


Message Passing



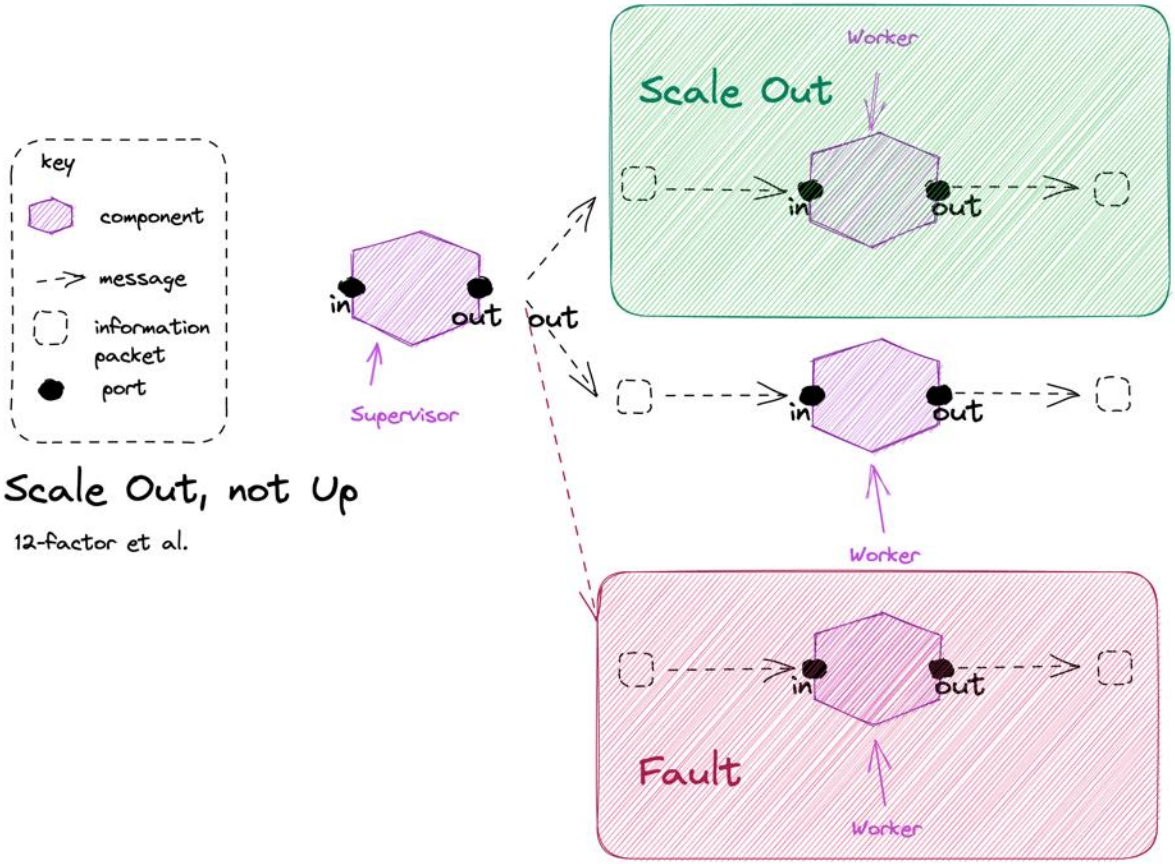
Circuit Breaker

We may want to stop consuming, due to a fault. We can periodically let a message through to see if the fault has cleared.



Scale Out, not Up

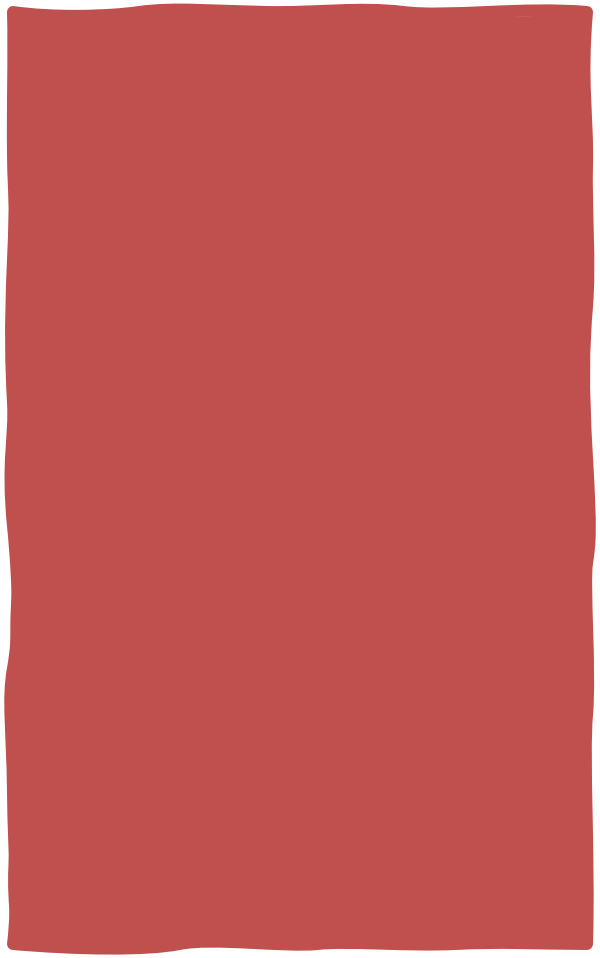
12-factor et al.



Scale Out, not Up

12-factor et al.

Putting It Together



EXERCISE MATERIAL

Flow

- Readme
- Slides

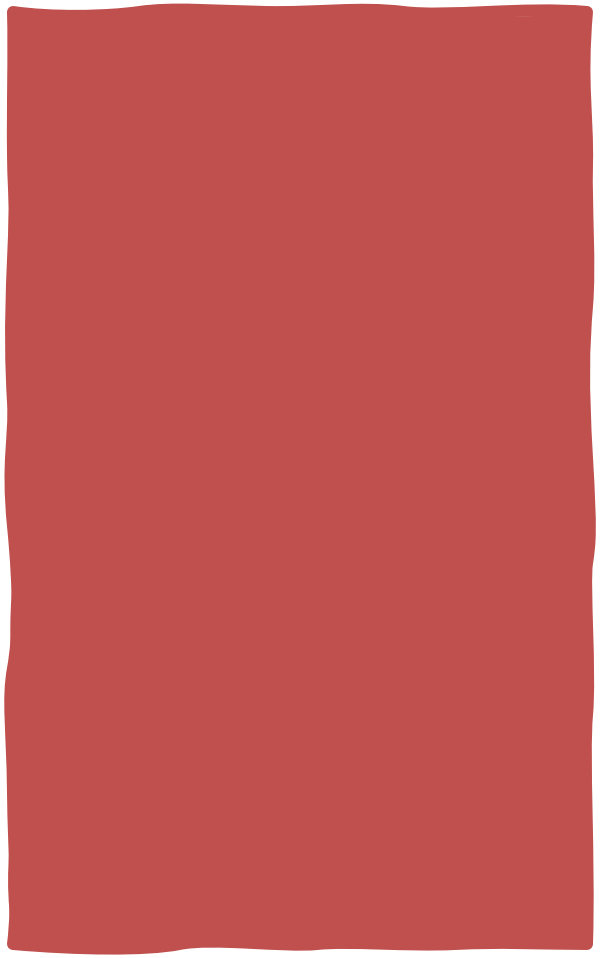
PROCESS AUTOMATION

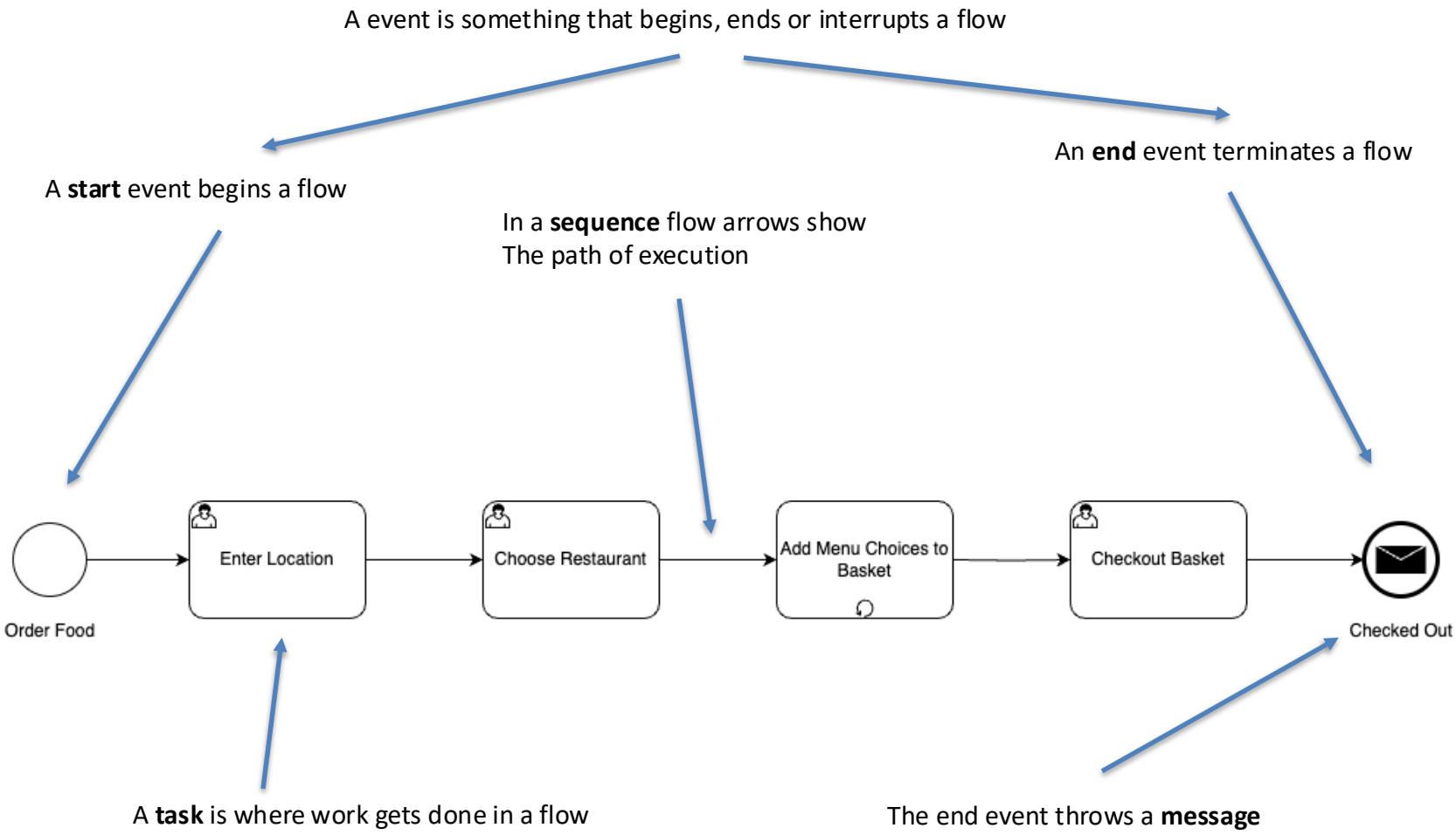
What is a microservice?

SOA is focused on business *processes*. These *processes* are performed in different steps (also called *activities* or *tasks*) on different systems. The primary goal of a **service** is to represent a “natural” step of business functionality. That is, according to the domain for which it’s provided, *a service should represent a self-contained functionality that corresponds to a real-world business activity*.

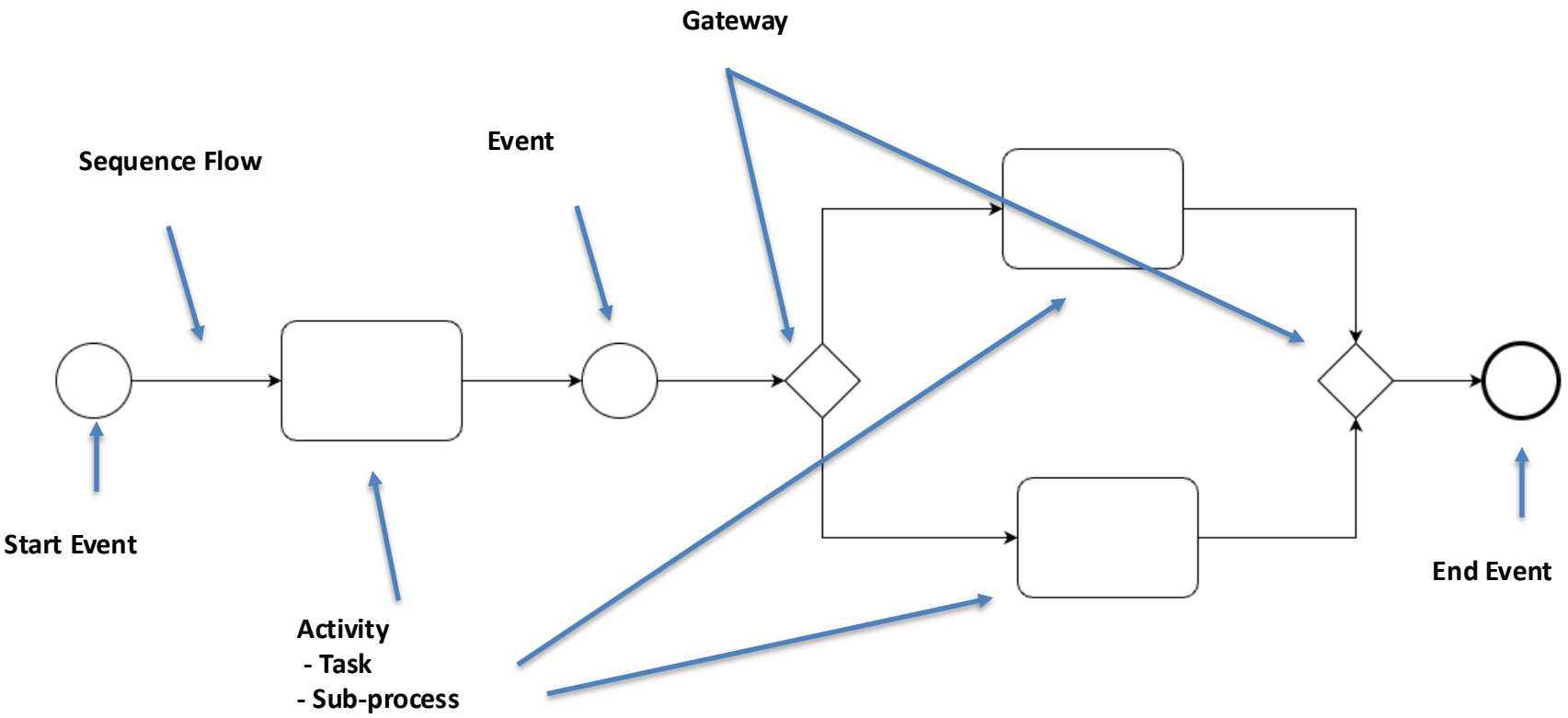
Josuttis, Nicolai M.. SOA in Practice: The Art of Distributed System Design . O'Reilly Media. Kindle Edition.

BPMN



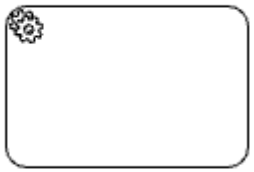


BPMN stands for *Business Development Management and Notation*. It is a visual language for diagramming a business processes in a clear, standardized and comprehensive way.



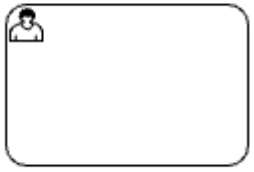
A BPMN diagram consists of a set of basic elements

Generic Task: basic task



Service Task: uses a Service

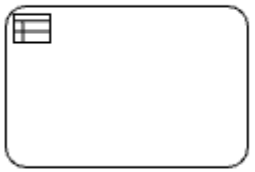
Receive Task: waits to receive a message from another participant



User Task: waits for human completion of task (via software)

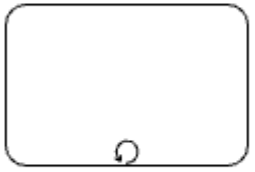
Send Task: sends a Message to another participant

Manual Task: waits for human completion of task (not via software)



Business Rule Task: executes a business rules engine

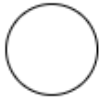
Script Task: executes code



Loop: this task loops

Transaction: this task is a transaction

Task: an atomic activity. By implication, cannot be broken down further.



None Event: typically start or end, no defined trigger

Message Event: a message is received from another participant

Time Event: a date or time triggers a process

Escalation Event: triggers a sub-process to expedite completion of an activity

Compensation Event: triggers a compensation sub-process

Conditional Event: triggers when a conditional expression becomes true

Signal Event: a broadcast event triggers flows

Parallel Event: multiple events required to trigger a sub-process

Cancel Event: used to trigger a sub-process with the abnormal termination of a transaction

Event: something that happens within a flow: starts it, ends it, or interrupts it.

Exclusive: creates alternative paths based upon an expression, only one taken

Inclusive: creates alternative paths based upon an expression, may be taken in parallel

Parallel: split of join paths

Complex: a complex expression determines what paths are taken

Event: instead of an expression, an event determines what path is taken

Gateway: controls how sequence flow branches and converges

Sequence Flow 

Message Flow 

Connecting Object: connects activities

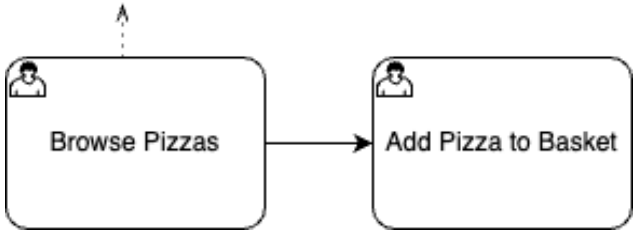
Workflow Patterns

- Workflow Patterns: *W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, A.P. Barros, 2000*
  - **5 Basic Patterns**
  - 15 Advanced Patterns

- Workflow Control Flow Patterns: A Revised View: *W.M.P. van der Aalst, Nataliya Mulyar, Nick Russell, A.H.M. ter Hofstede, 2007*
  - 23 New Patterns

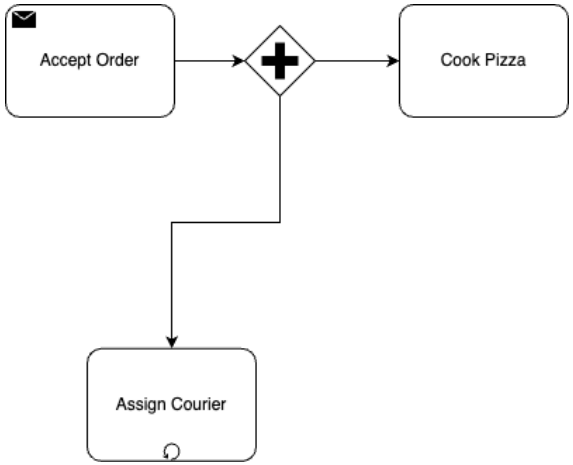
Pattern 1 – Sequence

- **Description**: One activity follows another.
- **BPMN**: Connect two tasks with a **sequence flow arrow**.
- Easy to model and execute.
    - **Example (Customer)**: Browse Pizza → Add Pizza to Basket.



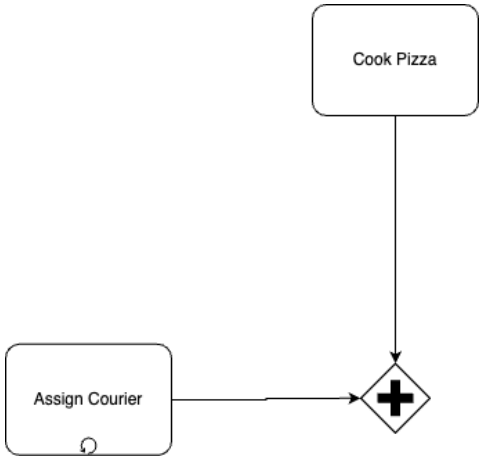
Pattern 2 – Parallel Split

- **Description**: One path splits into two or more **concurrent branches**.
- **BPMN**: Use a **Parallel Gateway** (+) to fork.
  - **Example (Pizza Shop)**: Assign Courier + Cook Pizza.



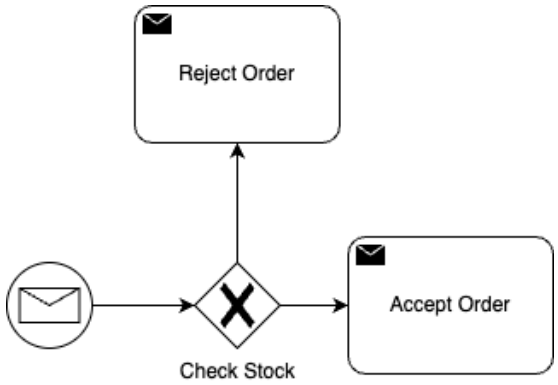
Pattern 3 – Synchronization (Join)

- **Description**: Wait until multiple concurrent branches complete.
- **BPMN**: Use a **Parallel Gateway** again to join.
  - **Example (Pizza Shop)**: Assign Courier + Cook Pizza.



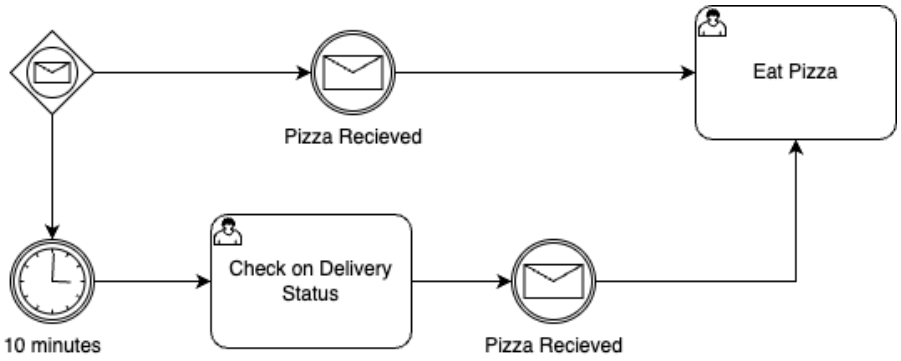
Pattern 4 – Exclusive Choice

- **Description**: Choose one path based on a condition.
- **BPMN**: Use an **Exclusive Gateway** (X) with condition expressions.
  - **Example (Pizza Shop)**: Check Availability → Accept Order X Reject Order.

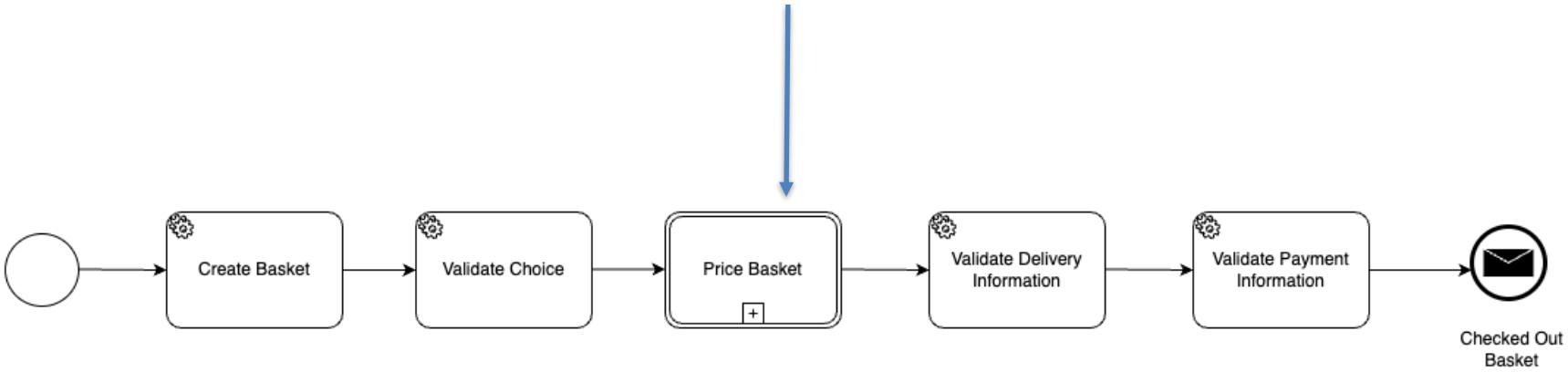


Pattern 5 – Simple Merge

- **Description**: Merge **non-concurrent** paths back into one.
- **BPMN**: Use a **converging sequence flow; no gateway is** needed if no synchronization is required.
  - **Example**: Multiple paths lead to Eat Pizza.



A **Process** describes a sequence or flow of **Activities** for carrying out work. In **BPMN** a **Process** is depicted as a graph of Flow Elements, a **Sequence Flow,** which is a set of **Activities**, **Events**, **Gateways**



A **Process** is an ***Orchestration****:*

- The **Sequence Flow** represents control of a process
- The **token** represents state associated with an instance
- In implementation, generally a process orchestration lives within an address space i.e. is not distributed, either an embedded workflow or external process manager.

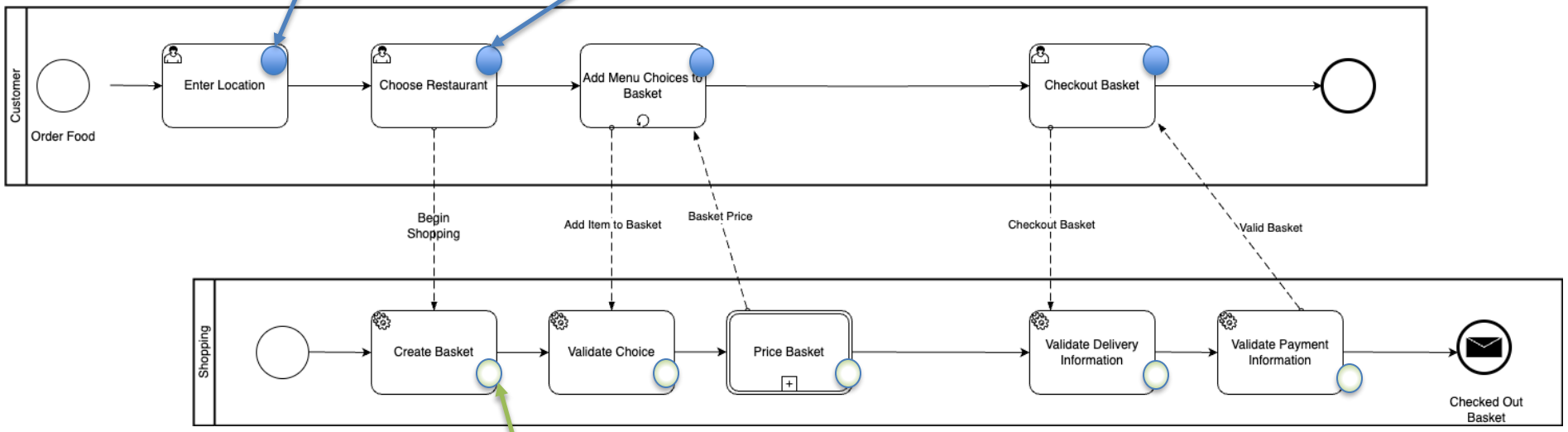
What is Orchestration?

Focused on a **single participant's perspective**

- Like writing *your own script*
- Includes control flow, state, and decisions
- All logic is local to the orchestrator
- Often implemented via state machines or workflow engines.

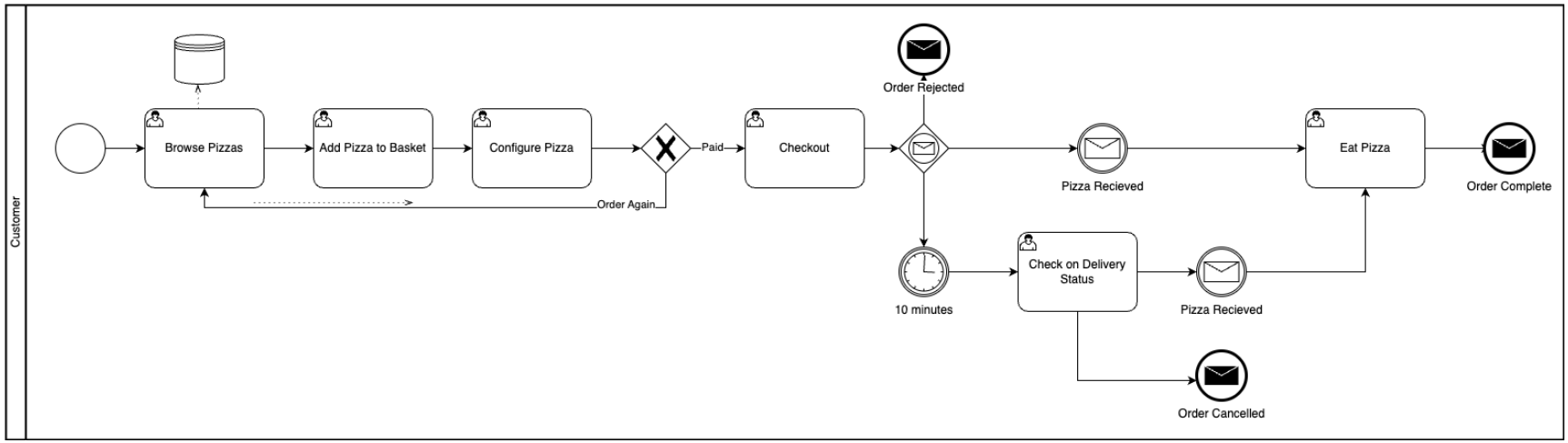
A **token** represents an instance of a process

A **token** flows down the process; it is the state of the process for that instance



A **token** flows down the process; it is the state of the process for that instance

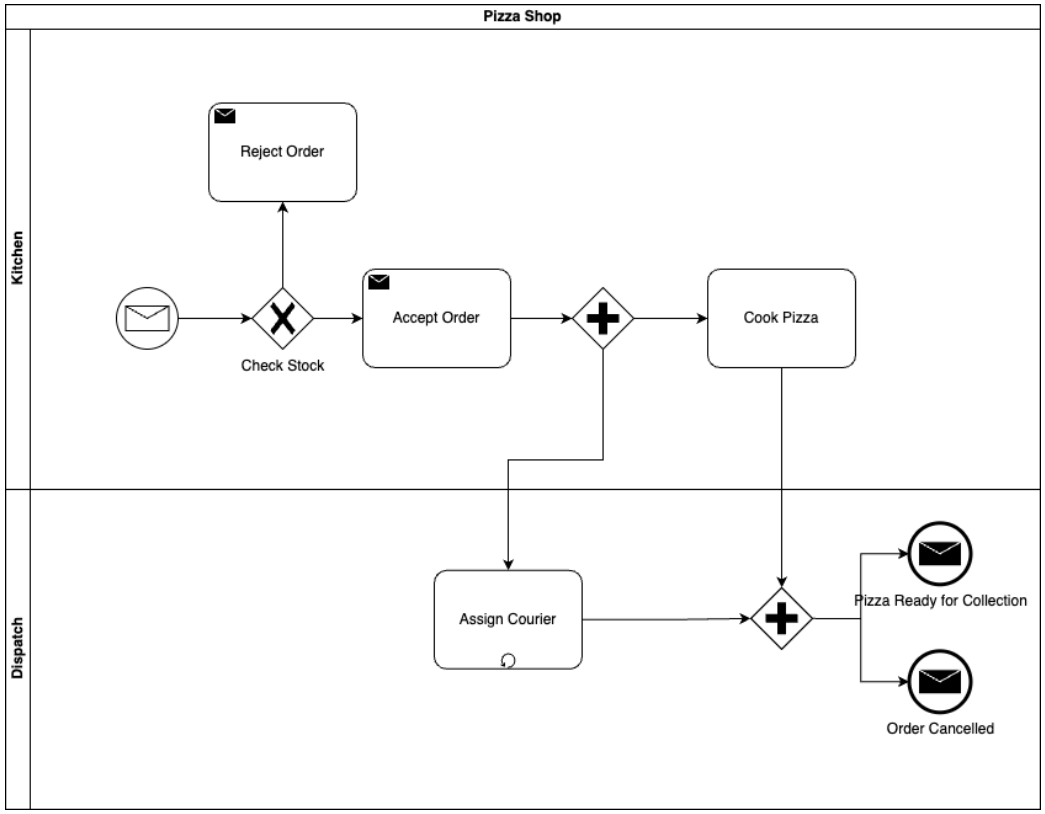
Pizza Example – BPMN Orchestration



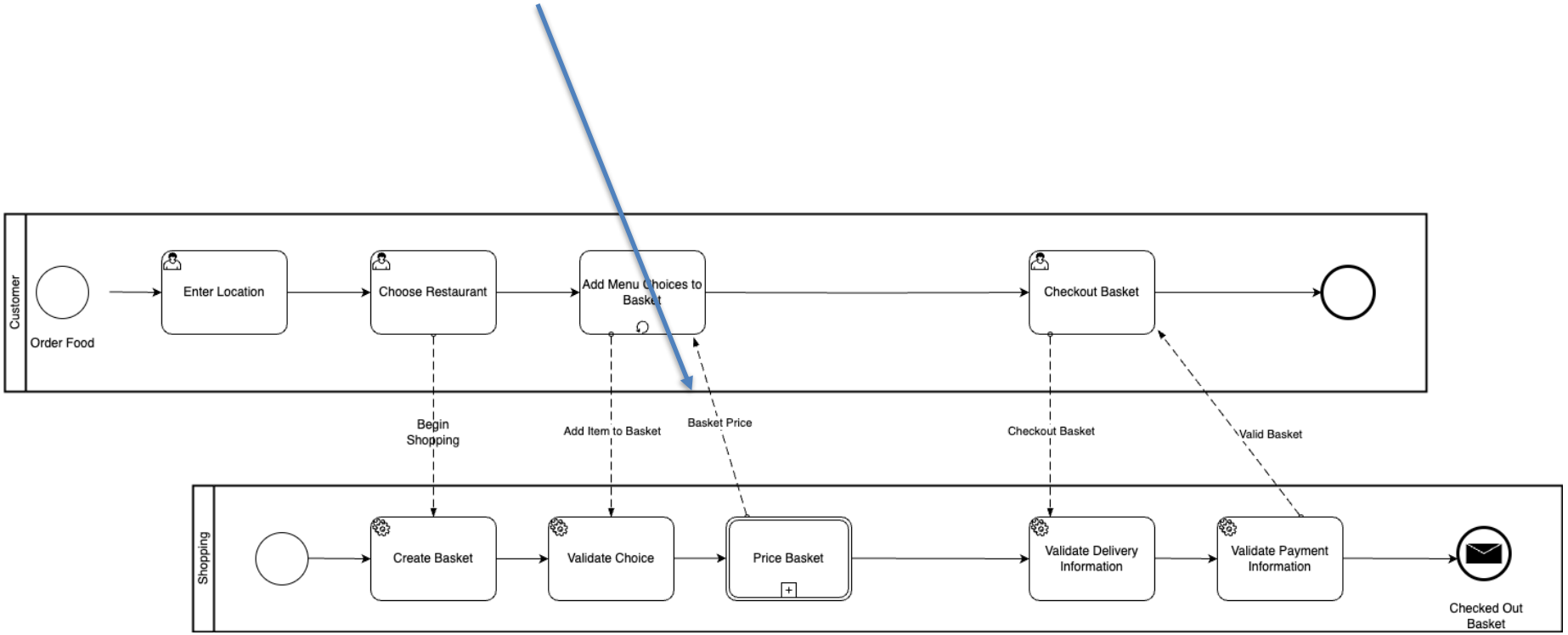
Customer Pool

Pizza Example – BPMN Orchestration

Pizza Shop Pool



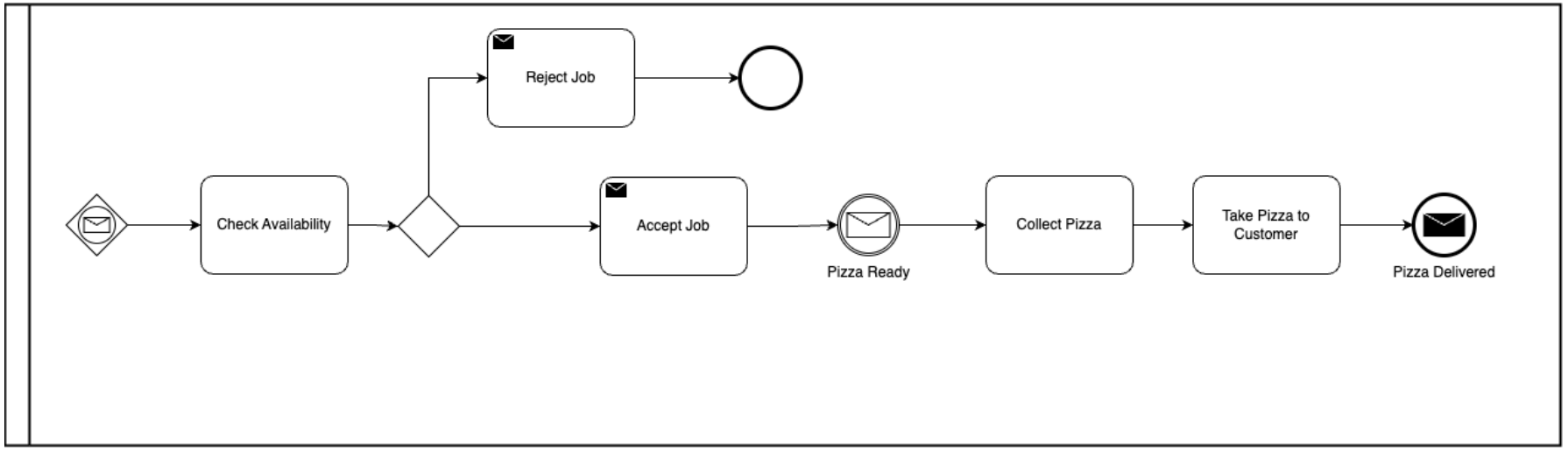
A **Collaboration** is where we have multiple participants.



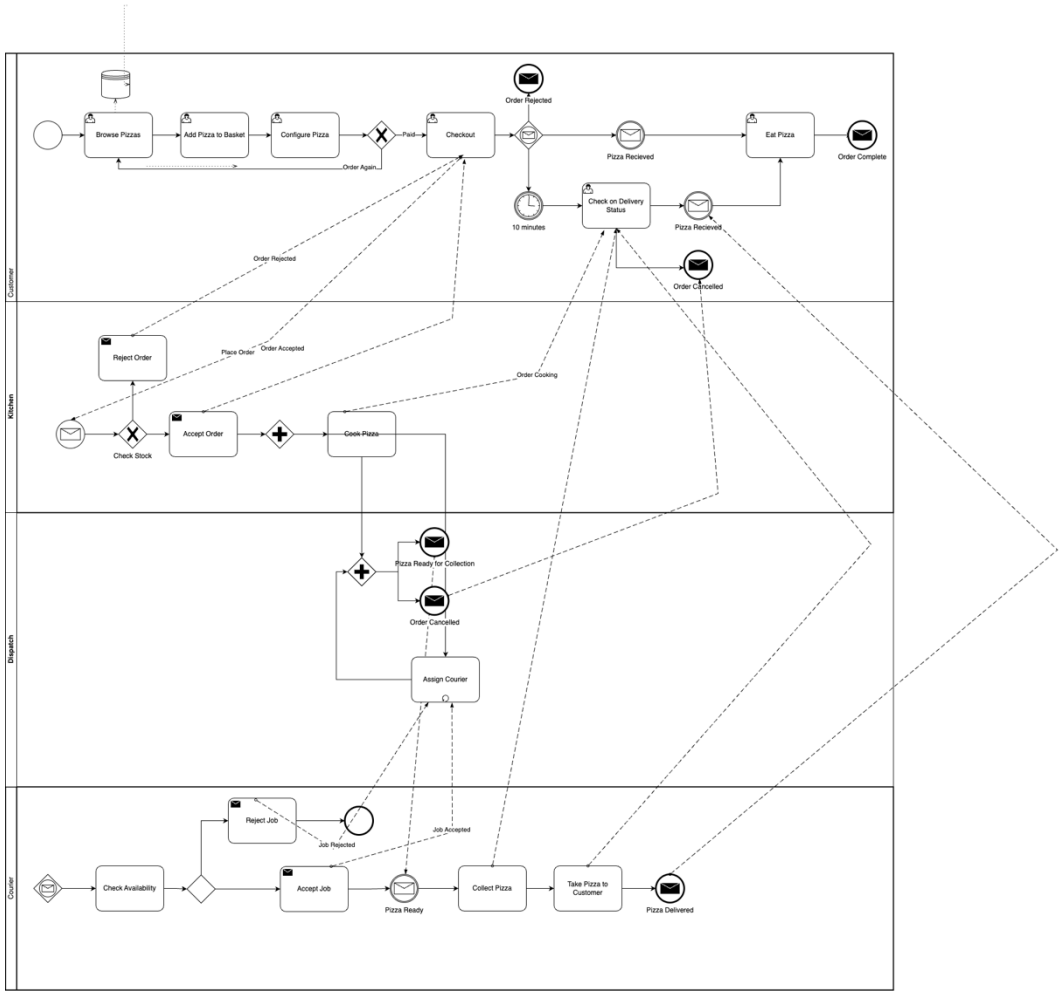
The message exchange between participants in a **Collaboration** is a ***Choreography***

- In implementation, generally, a choreography is the flow of messages between participants. Within each orchestrated process, there are message events that are caught or raised
- But Processes react to what happens in the Choreography, no one owns it (it does not have its own tokens or state)

Pizza Example – BPMN Orchestration



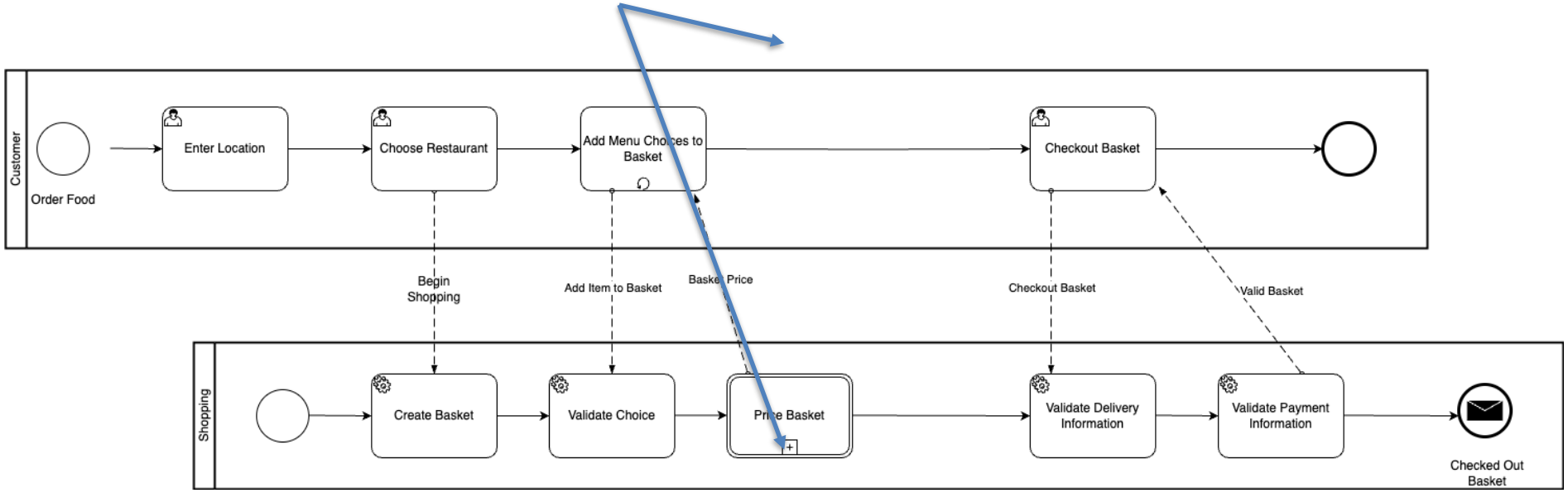
Courier Pool



Pizza Example – Pools and Lanes

A **Pool** is participant in a collaboration

- A participant can be a role or an organization
- Not required for the Main Internal Pool => modeler's own organization
- Required for other organization's processes
- A Lane distinguishes sequences of Activities in the Pool
- White Box - can see process; Black Box - cannot see process
- Activities within the Pool organised by sequence flow



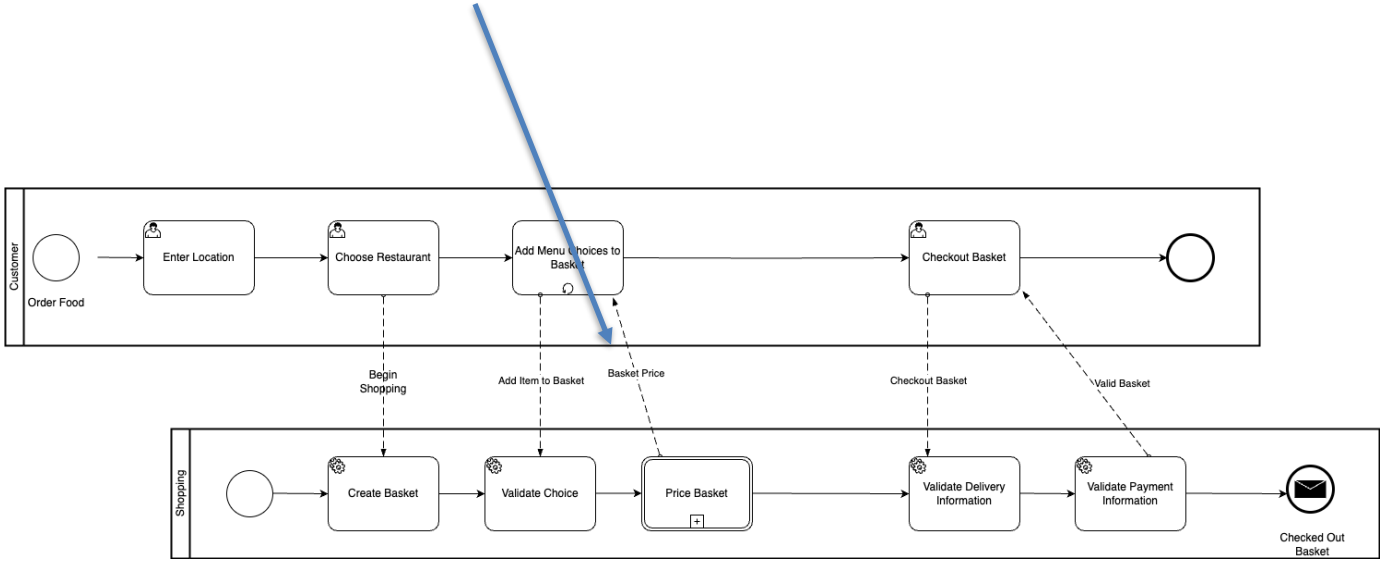
What is Collaboration

Focused on how **participants interact**

- Interaction might be within a workflow engine.
  - But this creates coupling, and both participants must run in the workflow engine.

- Interaction might be via an API
  - From an event-driven perspective, we are most interested in this model.

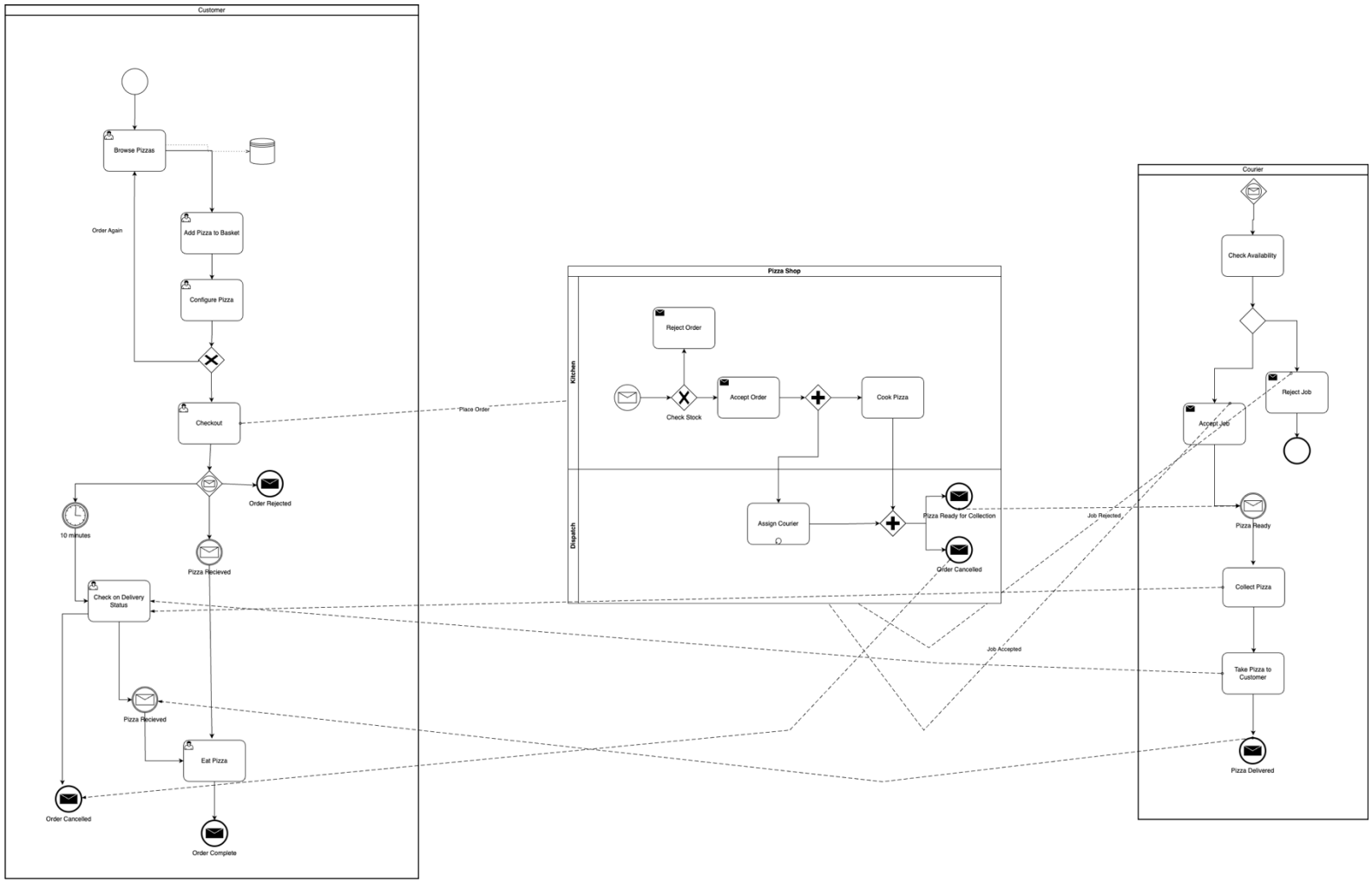
A **Collaboration** is where we have multiple participants.



The message exchange between participants in a **Collaboration** is a ***Choreography***

- In implementation, generally a choreography is the flow of messages between participants. Within each orchestrated process there are message events that are caught or raised
- But Processes react to what happens in the Choreography, no one owns it (it does not have its own tokens or state)

Pizza Shop Collaboration

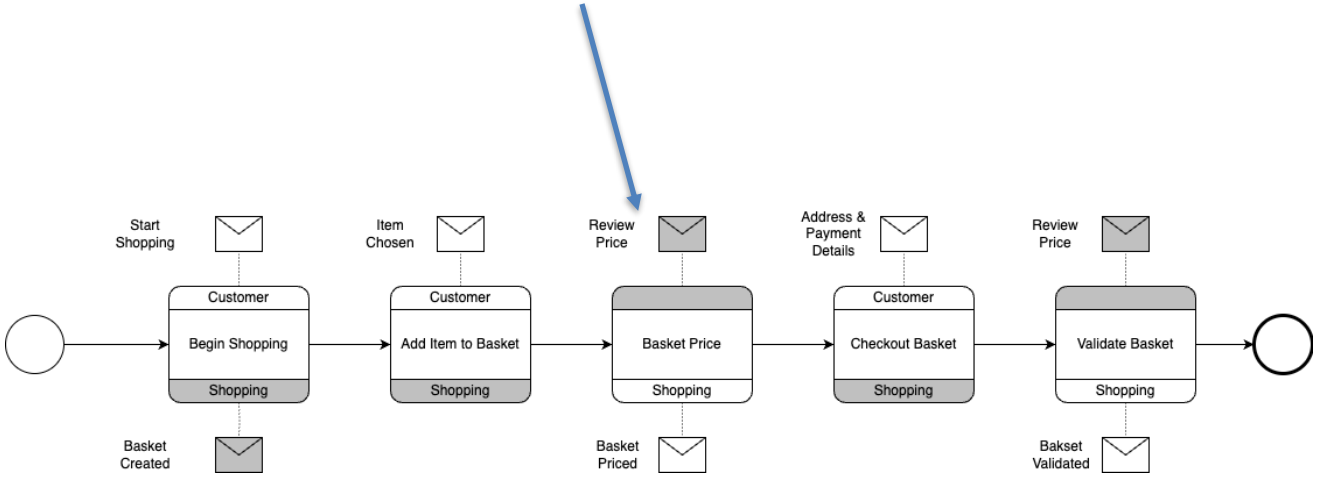


What is Choreography?

Focused on **interactions** across multiple participants

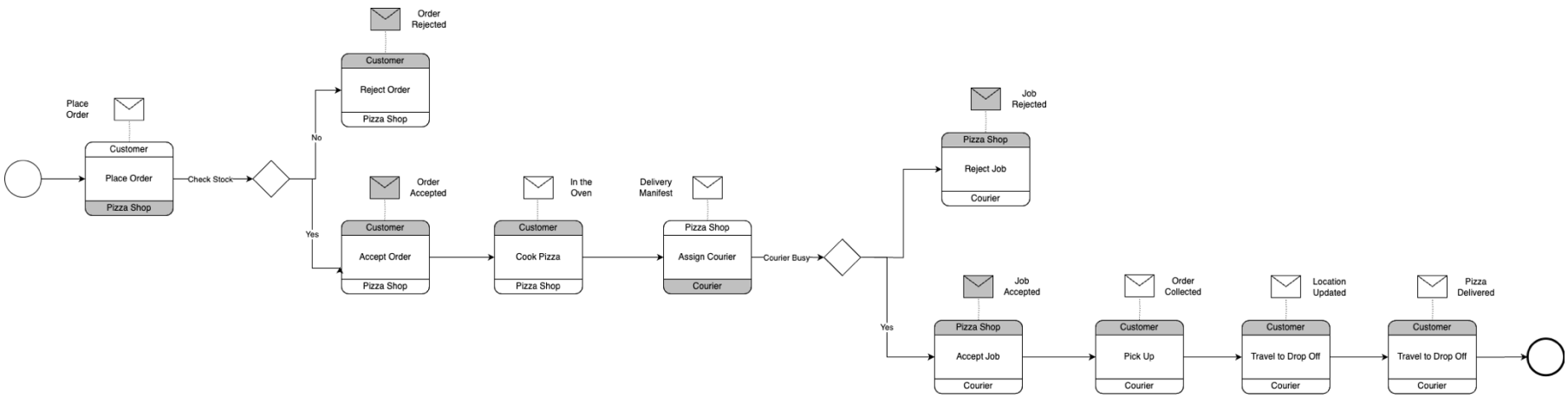
- Like describing a **dance,** no single owner of the flow.
- Defines *who talks to whom, in what order*
- No centralized control
- Cannot access shared internal data/state

A **Choreography** describes a sequence or flow of **Activities** between two different participants. In **BPMN** a **Choreography** is depicted as a graph of Flow Elements, a **Message Flow,** which is a set of **Activities**, **Events**, **Gateways**



A Conversation is a logical association of messages that can all be correlated
- Correlation Keys are used to associate messages that form part of the same Conversation
- This may be existing message data
- The first task in a conversation MUST populate the conversation id

Pizza Example - BPMN Choreography



Pizza Order Choreography

Messaging and Eventing

A common notion within EDA:

- Messaging (commands) is orchestration; Eventing is choreography

- It is not a helpful rule.

- An orchestration is a sequence, with attendant session state [token]
  - Frequently, it is not distributed and *is within* an executable process
  - A routing slip is a distributed orchestration that utilizes message attributes as the engine for application state.

- Choreography is the flow of control and information between orchestrations.
  - Frequently, it is out of process.
  - Can use messaging or eventing as required.

Event Chaining

A sensible balance of orchestrated bounded contexts and choreography between will reduce event chaining.

- However, Event Pinball can still be a problem between orchestrated services.

- In-Out (or Robust In-Only)
  - Use of In-* patterns with explicit transfer of control reduces the “pinball effect” in choreography
  - Creates **causality** in workflows by awaiting a response; you don’t send an event, and wait for a response, you send a command and wait for a response.

- Out-Only
  - Often better at the termination of a workflow, when the work is done, or to broadcast intermediate states with no expectation of response.

Allocating Responsibility for Collaborations

Commands/Messaging – Transfer of Control/Data.

- Typically, this involves delegating responsibility for part of a workflow.
- In-Out. Expected to Return
- In-Only. Not Expected to Return. But May on Error (Robust In-Only).

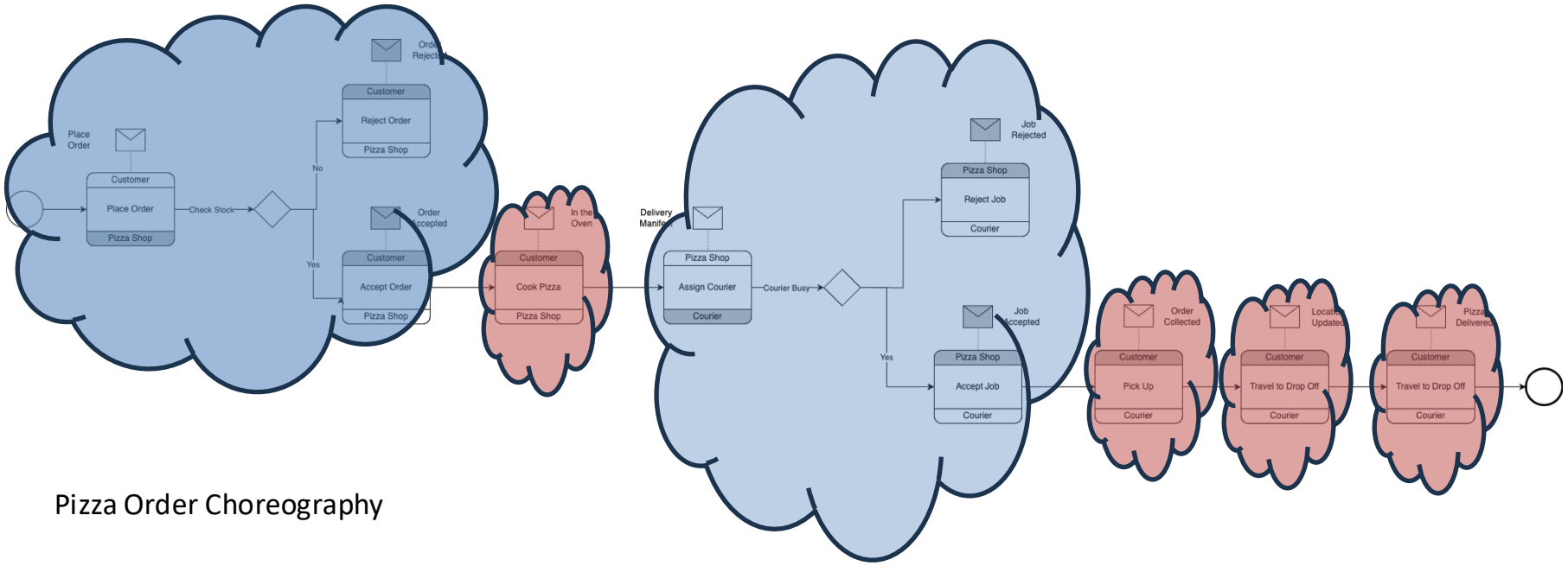
Events – Communicate the Outcome of a Workflow

- Typically, a workflow is complete or a milestone is broadcast for observation.
- Out-Only. No Expected Return.

Workflow code is Glue.

- Glue does not execute the domain; it is not the behaviour
- Instead, glue calls something that has the behaviour

Pizza Example – Messaging and Eventing



Pizza Order Choreography

Aspect	Orchestration	Choreography
Control	Centralized (one participant)	Decentralized (shared)
BPMN Shape	Inside a Pool	Between Pools
Perspective	One party’s process	Multi-party message exchange
Data Access	Internal	Shared messages
Execution Tool	API calls, Handlers, Workflow engine	Messaging protocol
Example	Pizza Shop coordinates cook + courier	Customer + Shop + Courier interact

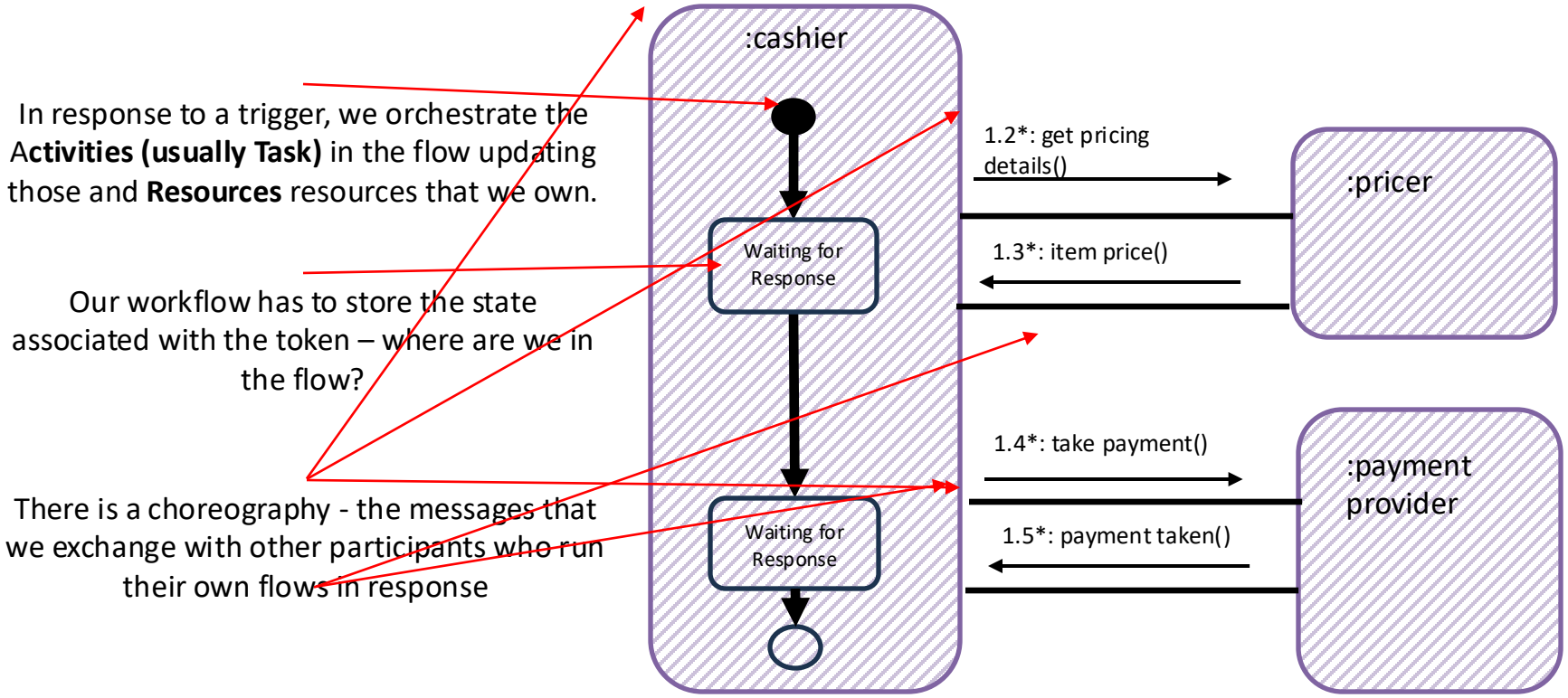
Orchestration vs. Choreography Summary

Activities and Resources

Broadly, a software component manages *activities* and *resources* [Pat Helland].

- **Resources**: domain concepts we manage, like restaurants, couriers, and customers.
- **Activities**: one or more sequences for our interaction with our resources.
- In response to the *token* moving through the sequence we update resources, and the activity to indicate our progress through the sequence.

The implementation question is: where does our activity state live, and who updates it?



Activity. Usually the task we need to perform. Most frequently a service task – our workflow engine just contains glue code that calls the domain logic internally, or messages another application. Might be a framework, might be bespoke state machine or pipes and filters.

Resources. Code and data used to manage shared items coordinated across multiple activities such as widgets in inventory or space on a truck needed for a shipment. Our domain model.

Durable Execution

Long-running processes must:

- Survive restarts
- Handle retries and failures
- Resume after waiting (e.g., waiting for payment or a courier)

Durable execution = we **persist activity state** to indicate which steps in a sequence have been completed, when we are awaiting an event etc.

Handlers + Activity State Updates

Handlers process events (change resources) and update activity state.

- Activity state is often implicit, such as within an aggregate.
- Durable because the state is stored (e.g., database), but **the control flow is implicit**
- Great for simple workflows, but logic becomes scattered across handlers
- Relies on guaranteed delivery to store work for retry unless ack'd
- Relies on Transactional Messaging (Outbox)
- Becomes complex once we introduce retry, circuit breakers & compensation
- Becomes complex once we have split, join, choice, merge (not a sequence)

```
public async Task<Order> HandleAsync(Order order, CancellationToken cancellationToken)
{
  var posts = new List<Guid>();

  var tx = await _uow.BeginOrGetTransactionAsync(cancellationToken);
  try
  {

    //should save the order
    orderRepository.Received(_uow, order);

    //then raise accept/reject
    bool canMake = await stock.Check(order);
    if (canMake)
        posts.Add(await _postBox.DepositPostAsync(new OrderAccepted(order.Id), cancellationToken:
cancellationToken));
    else
        posts.Add(await _postBox.DepositPostAsync(new OrderAccepted(order.Id), cancellationToken:
cancellationToken));

    await tx.CommitAsync(cancellationToken);

    //kick off cook and assign courier tasks
    await cookRequests.Writer.WriteAsync(new CookRequest(order), cancellationToken);
    await deliveryRequests.Writer.WriteAsync(new DeliveryRequest(AssignCourier(), order), cancellationToken);


  }
  catch (Exception e)
  {
    _logger.LogError(e, "Exception thrown handling Add Greeting request");
    //it went wrong, rollback the entity change and the downstream message
    await tx.RollbackAsync(cancellationToken);
    return await base.HandleAsync(addGreeting, cancellationToken);
  }

  await _postBox.ClearOutboxAsync(posts, cancellationToken:cancellationToken);

    return await base.HandleAsync(addGreeting, cancellationToken);
}

```

(Fault) Handlers + Activity State Updates

How do we handle compensation flows?

- May retry the handler, and only if it fails X times, run the fallback.
- On fault, choose to run a “fallback handler” – initiates compensating action.
- Typically, needs to reverse actions that have occurred so far; write an “undo” for each “do”.
- Typically, a straightforward approach is to send a fault message back upstream and undo it upon receipt.

- Fault Message
  - For reliable In-Only, a fault channel is required, which is only used for compensation.
  - For In-Out, the message sent to the response channel should indicate fault, not success.

State Machine + Activity State Updates

When the interaction of Handlers becomes complex, we can make the activity explicit.

- A State Machine represents the activity and triggers actions in response to transitions.
- Transitions are caused by receiving a message.
- The handler receives a message and loads the corresponding state machine associated with the conversation ID.
- This triggers the transition denoted by the receipt of the message; the state machine runs any code associated with that transition.
- Save the new state of the state machine and ack the message.

```
1        public OrderStateMachine(ILogger<OrderStateMachine> logger)
2         {
3             InstanceState(x => x.CurrentState);
4             ConfigureCorrelationIds();
5
6             Initially(
7               When(OnOrderSubmitted) // Receive the request
8               .Then(x => logger.LogInformation($"Order submitted"))
9               .SendAsync(new Uri("queue:stock-check"), context => context.Init<DebitAccount>(new
10              {
11                  context.Message.OrderId,
12                  //... details
13
14              })) // Calls debit
15              .TransitionTo(PendingStock)
16              );
17
18            During(PendingStock, // While PendingStock, if...
19                When(OnHasStock) // Check has been succeeded?
20                  .Then(x => logger.LogInformation($"Stock available"))
21                  .SendAsync(new Uri("queue:order-accepted"), context => context.Init<CallPartner>(new
22                  {
23                      context.Message.CorrelationId,
24                      context.Message.OrderId,
25                      //..details
26
27                  }))  // Calls the kitchen
28                  .TransitionTo(AwaitingKitchen),
29                When(OnLackStock) // Stock check has been failed?
30                  .Then(x => logger.LogInformation($"Stock failed"))
31                  .TransitionTo(RejectOrder)
32                );
33
34            //...
35         }
```

(Fault) State Machine + Activity State Updates

How do we handle compensation flows?

- May retry the transition, and only if it fails X times, run the compensation.
- On fault, transition to a faulted state and initiate compensating flow to the “safe” state.
- Typically, this involves reversing actions that have occurred so far; write an “undo” for each “do”.
- Requires *durable execution* – if we fault partway through, on restart, we can resume.

- The “**saga**” pattern, when used by messaging frameworks, is typically a state machine.
  - In essence, guarantees that on a fault, the state machine will transition back to “safe”
  - Implementations run all fault transitions to safe before ack fault message.

Routing Slip + Activity State Updates

If we want to distribute steps in the sequence, we need to transport the activity state in the message.

- A routing slip is an envelope that indicates the activity state (next step, context, etc.).
- Think of it as “Message As the Engine of Application State”
- The handler updates the state when work is done and forwards according to the routing slip. Then acks.
- Complex if “next” is dynamic routing, as the code must understand how to execute.
- Complex because you are distributing the steps, hard to observe [Otel]

(Fault) Routing Slip + Activity State Updates

How do we handle compensation flows?

- Next is now the fault next, from this step; what do we need to reverse.
- Context is “faulted”
- The handler updates the state when work is done and forwards according to the routing slip. Then acks.

Workflow Engines + Activity State Updates

When we want more than just state transitions (split, join, merge, choice, etc.), use a workflow engine.

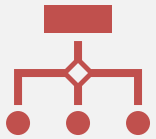
- A workflow engine explicitly models a sequence of activities and gateways.
- May directly support BPMN (or proprietary language); allows visual modelling.
- A step receives the token. The step is glue code: it invokes the action in the domain, calls another service, and sends a message.
- A step may wait for an event or a timer to signal that execution should resume. It may be the handler equivalent, or it may be triggered by the handler calling the engine.
- A job scheduler provides durable execution. Typically, any instance can resume a paused job – distributed.

(Fault) Workflow Engines + Activity State Updates

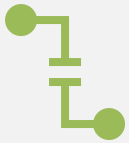
How do we handle compensation flows?

- Is the error a business error or a technical error?
- A known business error (i.e. card declined) is usually modelled explicitly as a branch via a gateway.
- A retry frequently handles technical faults. If the retry fails and a catch is present, then a fallback runs. If there is no catch, then either the workflow rolls back to the last wait state or terminates.

Embedded vs. External Workflow Engines



Embedded: The workflow runs in process and calls your code directly.



External: The workflow runs in its own process; it can execute code directly, but should favour calling services to get work done.

Workflow Engines: Lessons from SOA

SOA era favoured external workflow engines, Enterprise Service Buses (ESBs)

Business logic moved into the bus — tightly coupled, hard to test

Led to **Guerilla SOA**: reclaim domain logic into services, embrace simplicity

Smart Endpoints, Dumb Pipes

ESB products often include sophisticated facilities for message routing, choreography, transformation, and applying business rules. The microservice community favours an alternative approach: **smart endpoints and dumb pipes**.

https://martinfowler.com/articles/microservices.html#SmartEndpointsAndDumbPipes

Embedded vs. External Workflow Engines

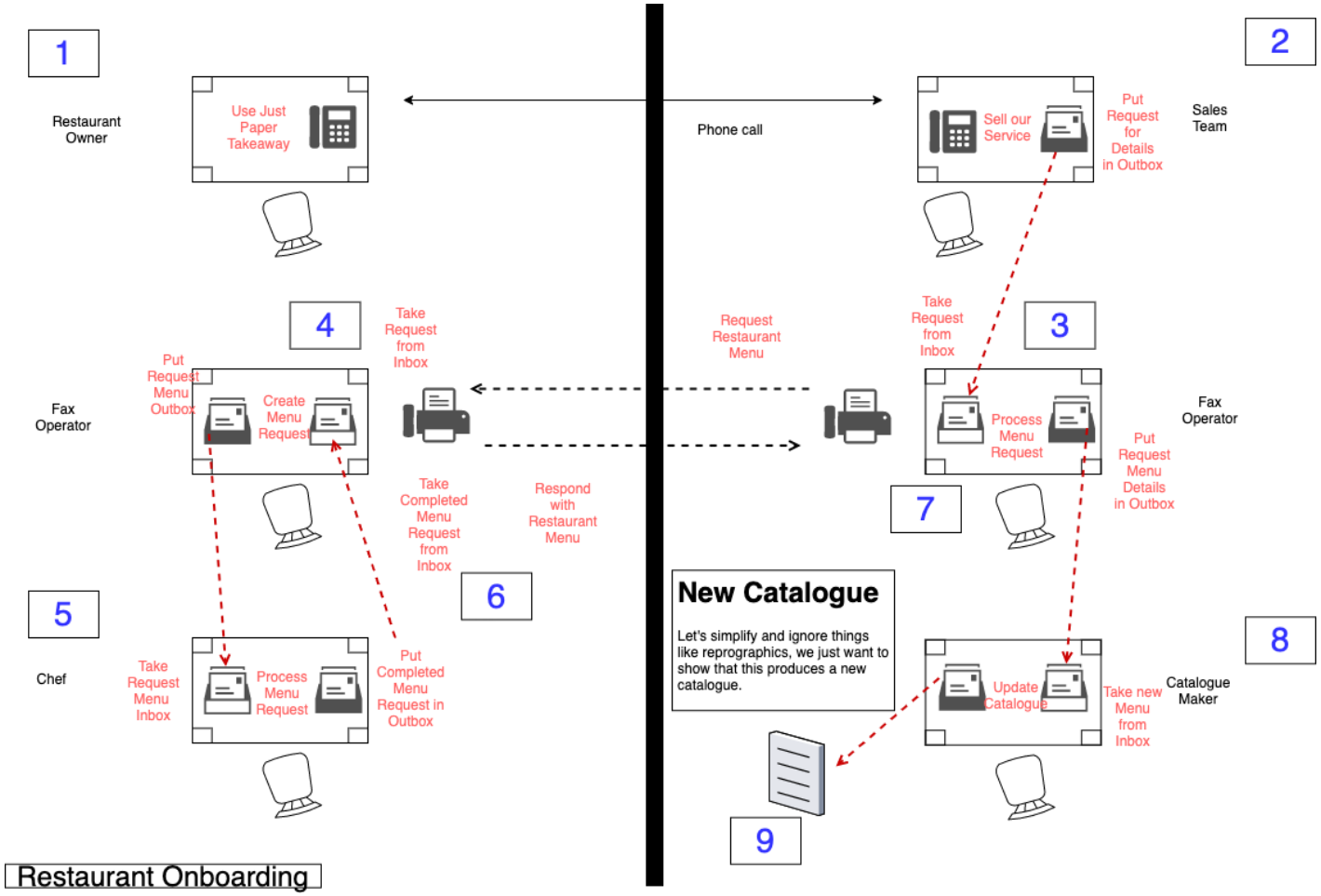
Favor embedded Workflow engines when the workflow is within a bounded context; everything between bounded contexts is choreography

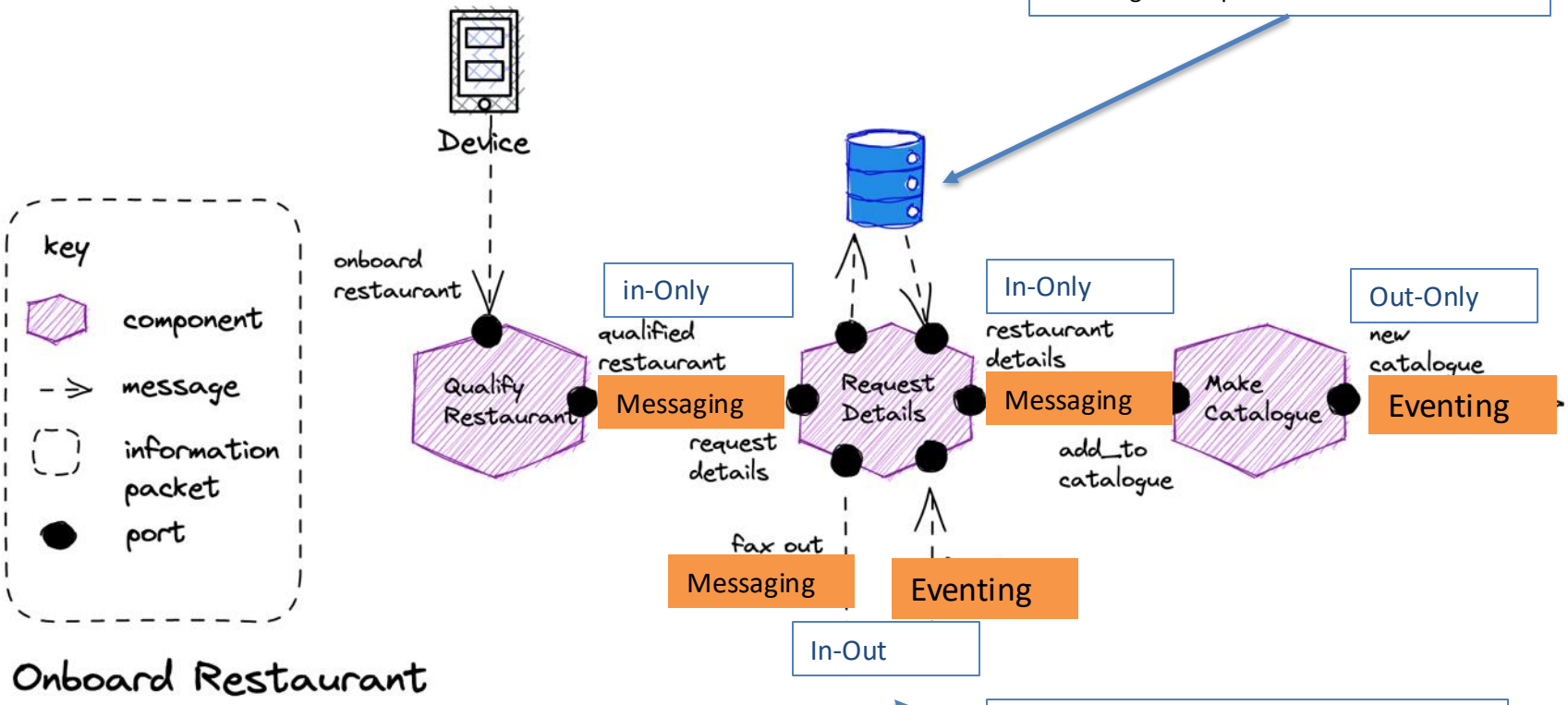
Reserve external Workflow engines for orchestrating the flow between bounded contexts, when relying on choreography is too complex.

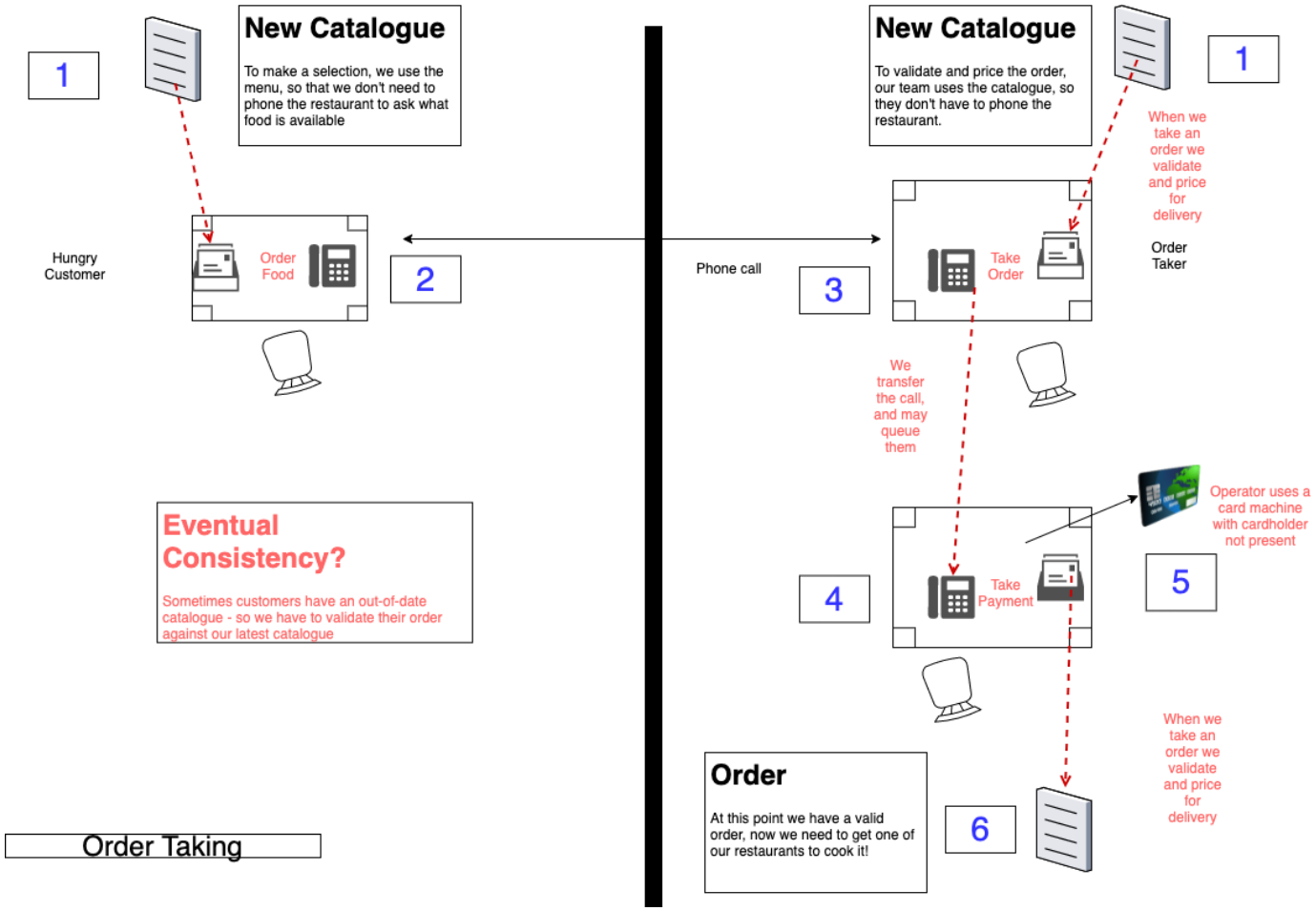
Implementing Workflow Patterns

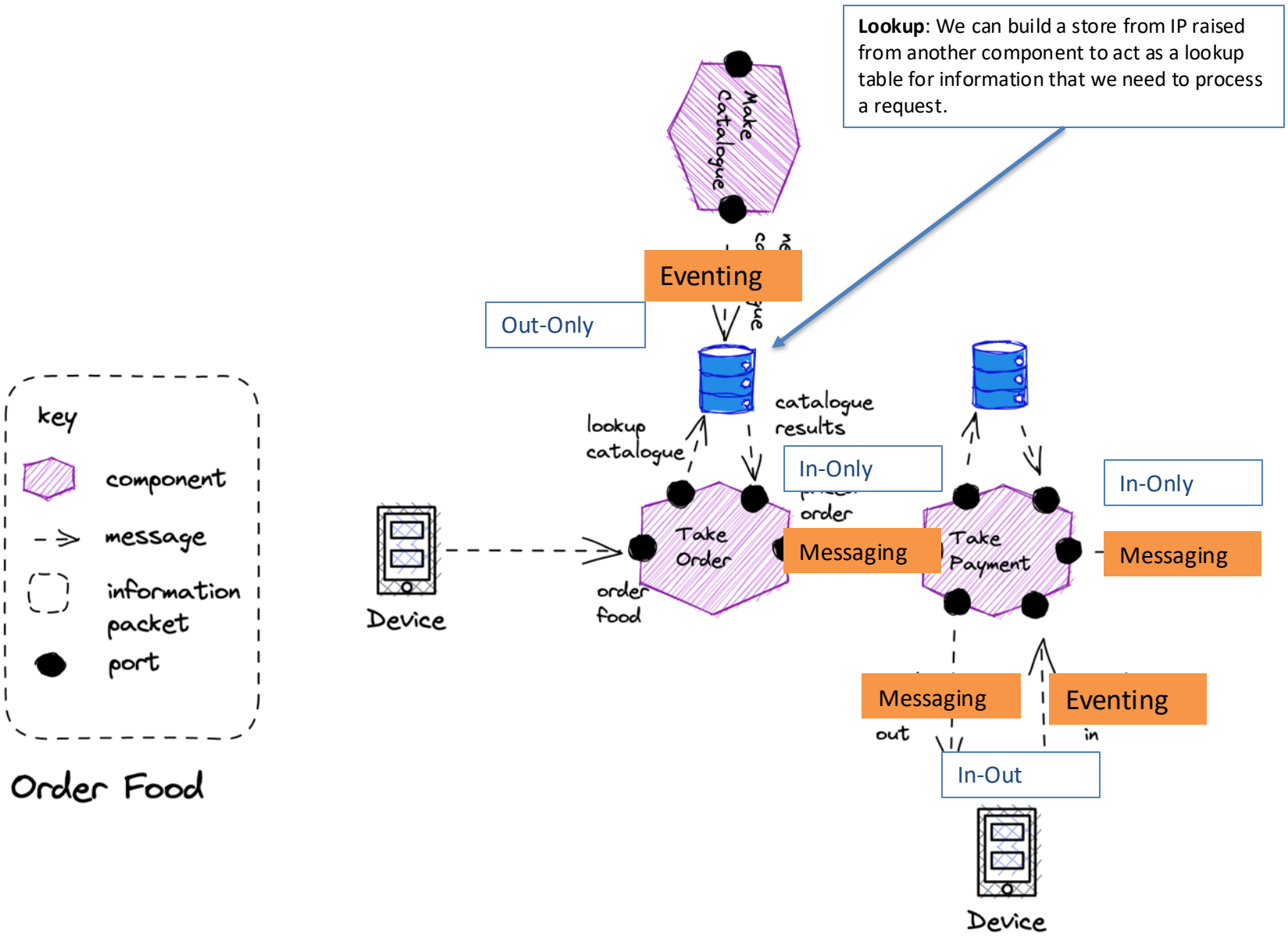
Pattern	Handlers Only	State Machine	Workflow Engine
Sequence	Chain message handlers	State transitions (A→B→C)	Sequential steps
Parallel Split	Trigger multiple handlers	⚠ Complex coordination manually	Parallel branches with fork/join
Synchronization	Wait for multiple states	⚠ Hard to do without orchestration	Join after parallel branches
Compensation	Fallback Handler	Saga	Implicit
Exclusive Choice	Conditional in handler	State transition based on input	Conditional branching
Simple Merge	Trigger single follow-up	Re-entrant transition or rehydration	Multiple incoming flows, OR gateway

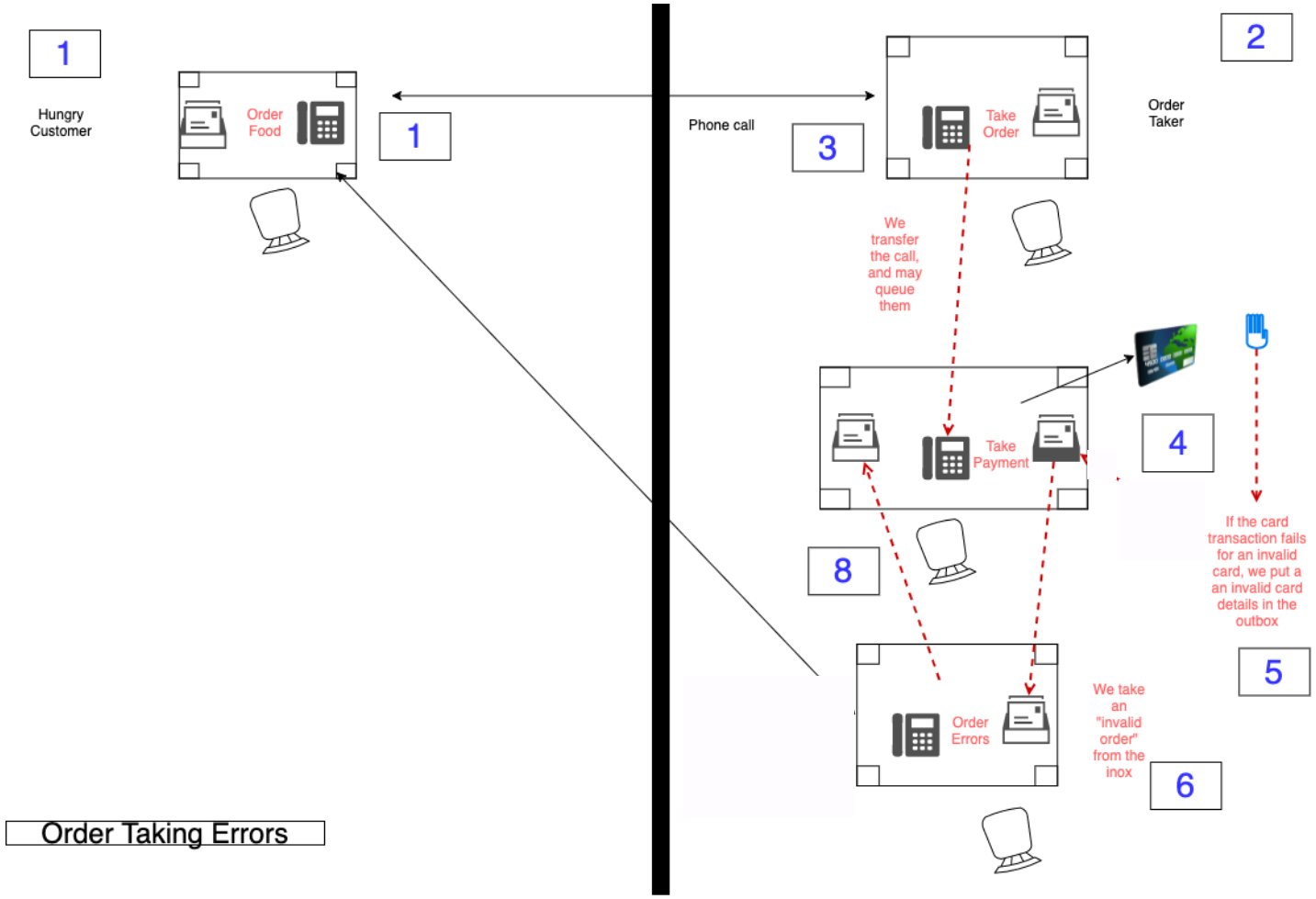
PUTTING IT TOGETHER

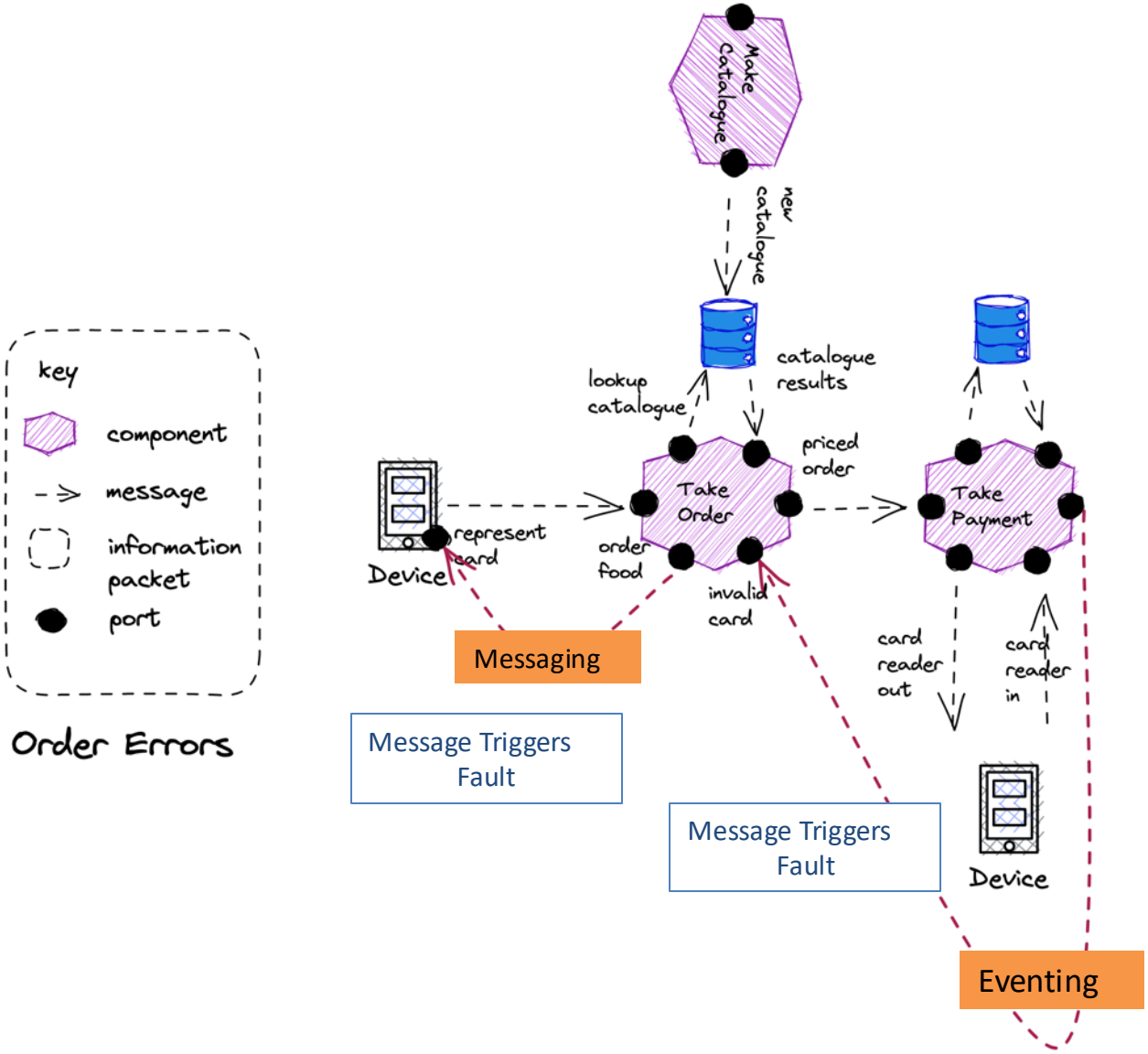

1
Restaurant Owner
Use Just Paper Takeaway
Phone call
2
Sell our Service
Put Request for Details in Outbox
Sales Team
4
Take Request from Inbox
Put Request Menu Outbox
Create Menu Request
Fax Operator
Request Restaurant Menu
Take Request from Inbox
3
Process Menu Request
Put Request Menu Details in Outbox
Fax Operator
7
Take Completed Menu Request from Inbox
Respond with Restaurant Menu
6
5
Chef
Take Request Menu Inbox
Process Menu Request
Put Completed Menu Request in Outbox
New Catalogue
Let's simplify and ignore things like reprographics, we just want to show that this produces a new catalogue.
8
Update Catalogue
Take new Menu from Inbox
Catalogue Maker
9
Restaurant Onboarding

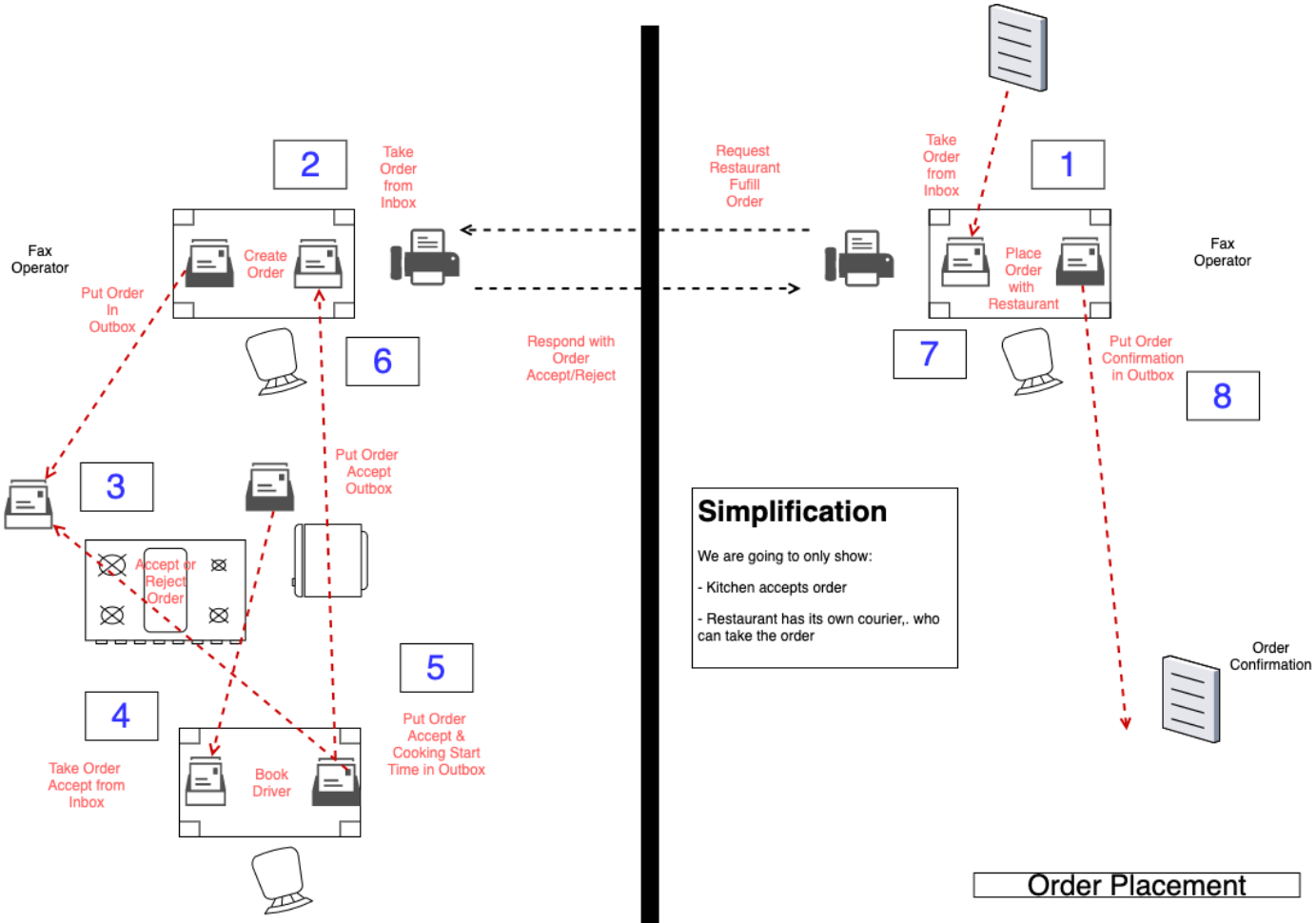

Storage: Request details may need storage to be reliable, it takes work from the request_details port, but cannot forward it to the restaurant_details port without waiting for an answer from fax_in. We store the state of the workflow in case we crash before receiving the response.
Device
key
component
message
information packet
port
onboard restaurant
in-Only
qualified restaurant
Qualify Restaurant
Messaging
request details
Request Details
In-Only
restaurant details
Messaging
add_to catalogue
Make Catalogue
Out-Only
new catalogue
Eventing
fax out
Messaging
Eventing
In-Out
Correlation: To allow us to match the response to the saved state we use a correlation id. We send it on the ip to fax_out, and the restaurant includes it on the ip from fax_in. We use the ip to look up the workflow we stored.
Onboard Restaurant

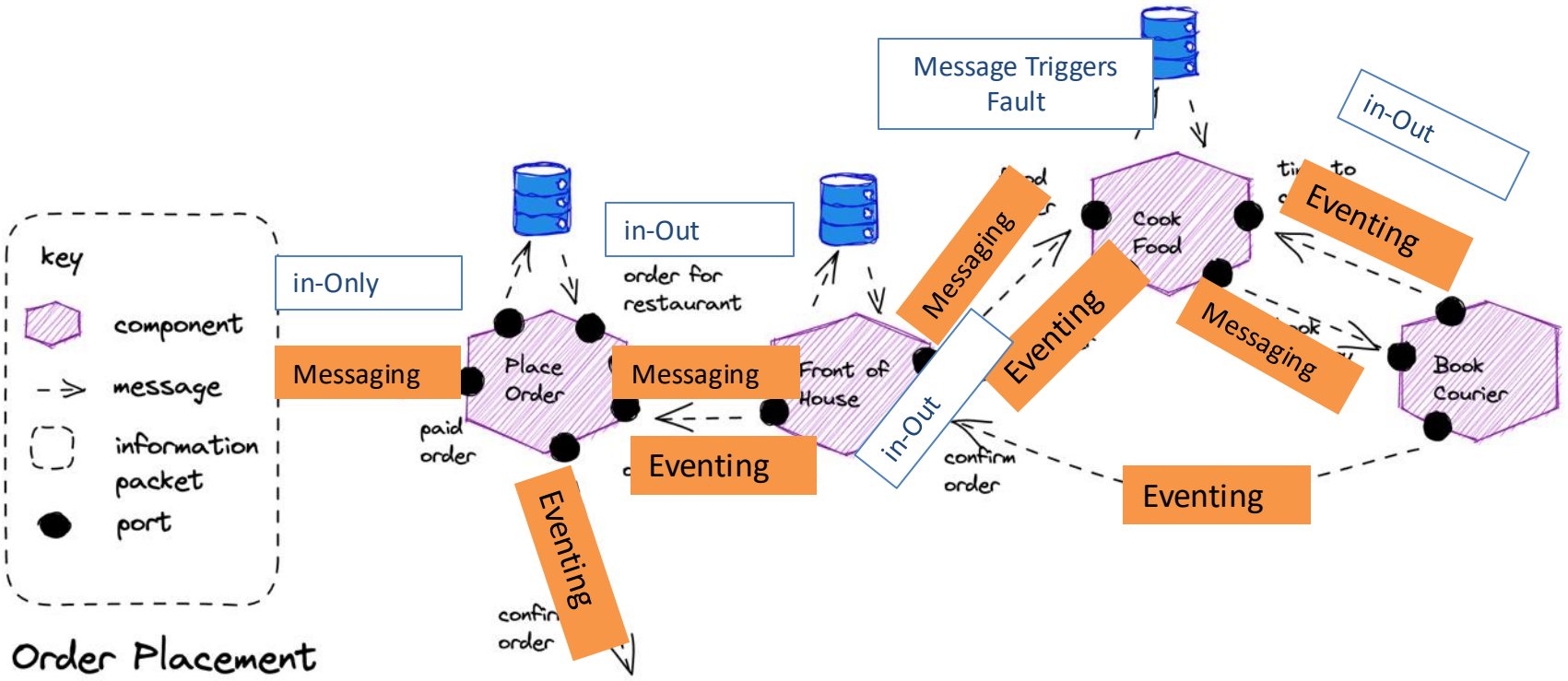

1
New Catalogue
To make a selection, we use the menu, so that we don't need to phone the restaurant to ask what food is available
Hungry Customer
Order Food
2
Phone call
3
New Catalogue
To validate and price the order, our team uses the catalogue, so they don't have to phone the restaurant.
1
When we take an order we validate and price for delivery
Order Taker
Take Order
We transfer the call, and may queue them
Eventual Consistency?
Sometimes customers have an out-of-date catalogue - so we have to validate their order against our latest catalogue
4
Take Payment
5
Operator uses a card machine with cardholder not present
When we take an order we validate and price for delivery
Order
At this point we have a valid order, now we need to get one of our restaurants to cook it!
6
Order Taking

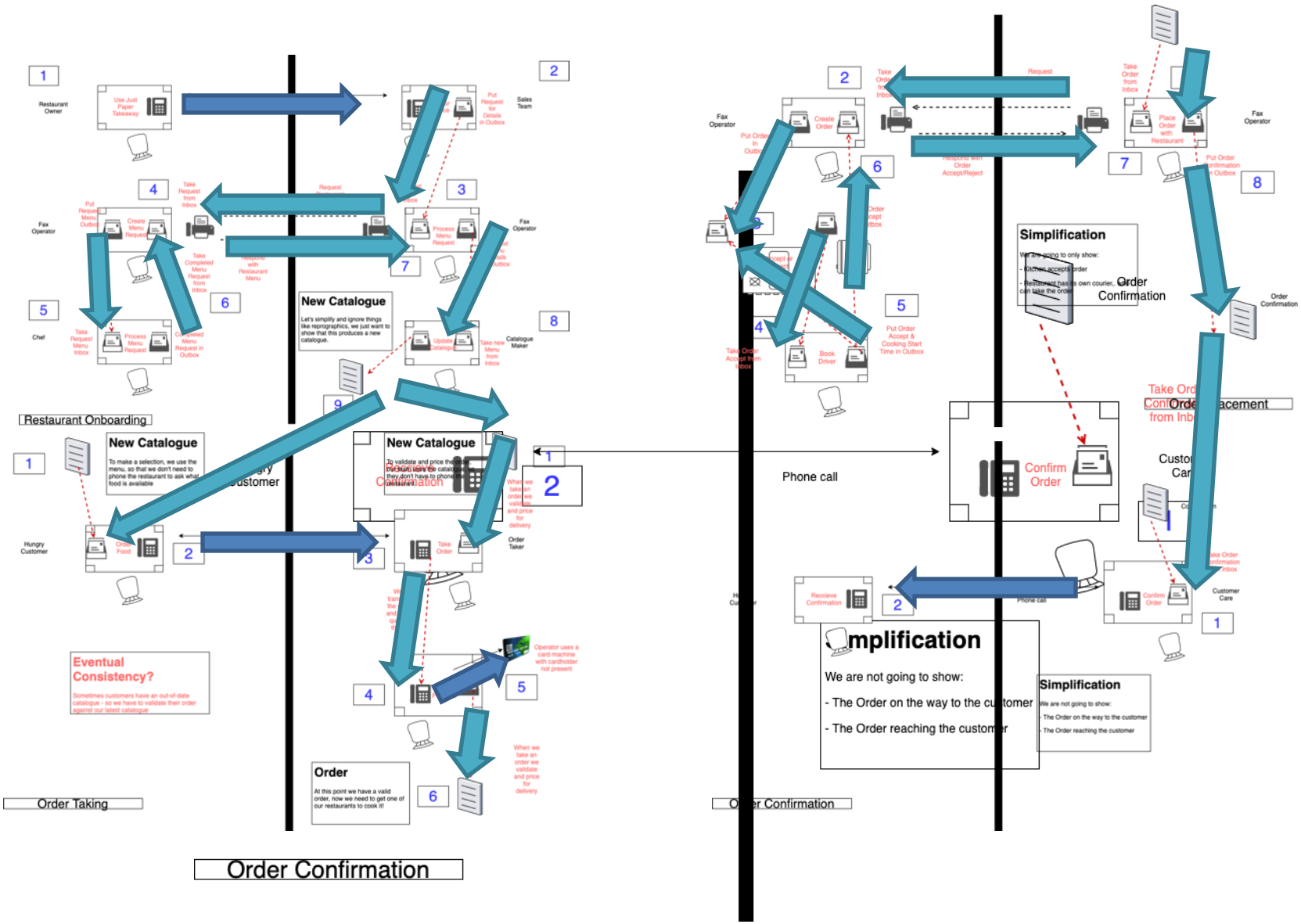

Lookup: We can build a store from IP raised from another component to act as a lookup table for information that we need to process a request.
Make Catalogue
Eventing
Out-Only
lookup catalogue
catalogue results
In-Only
order
Take Order
Messaging
order food
Device
Take Payment
In-Only
Messaging
Messaging
Eventing
out
in
In-Out
Device
key
component
message
information packet
port
Order Food

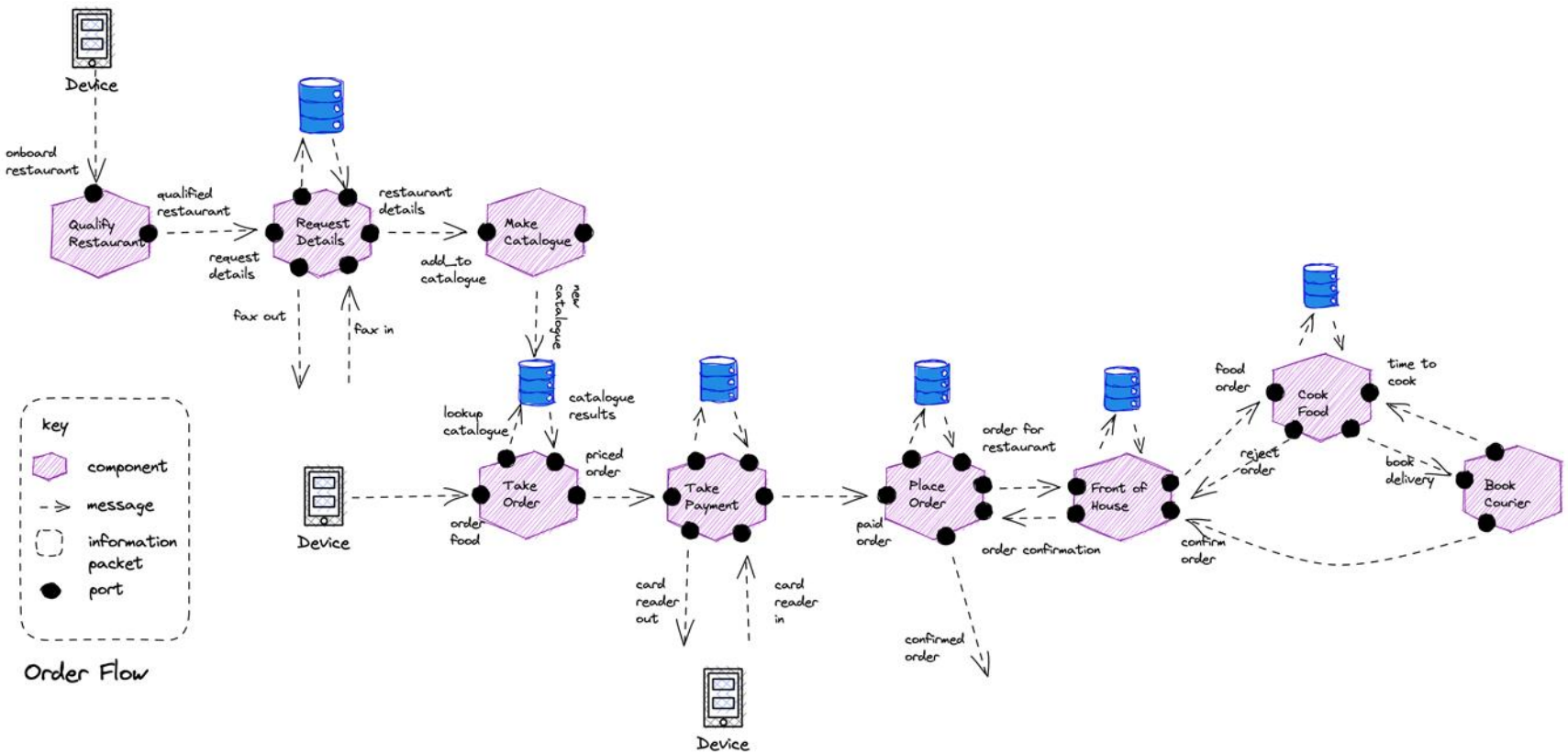

1
Hungry Customer
Order Food
1
Phone call
3
2
Take Order
Order Taker
We transfer the call, and may queue them
Take Payment
4
If the card transaction fails for an invalid card, we put a an invalid card details in the outbox
5
8
Order Errors
We take an "invalid order" from the inox
6
Order Taking Errors


key
component
message
information packet
port
Order Errors
Make Catalogue
new catalogue
lookup catalogue
catalogue results
priced order
Take Order
Take Payment
represent card
Device
order food
invalid card
card reader out
card reader in
Device
Messaging
Message Triggers Fault
Message Triggers Fault
Eventing


2
Take Order from Inbox
Request Restaurant Fufill Order
Take Order from Inbox
1
Fax Operator
Create Order
Place Order with Restaurant
Fax Operator
Put Order In Outbox
6
Respond with Order Accept/Reject
7
Put Order Confirmation in Outbox
8
Put Order Accept Outbox
3
Accept or Reject Order
Simplification
We are going to only show:
- Kitchen accepts order
- Restaurant has its own courier,. who can take the order
Order Confirmation
5
4
Put Order Accept & Cooking Start Time in Outbox
Take Order Accept from Inbox
Book Driver
Order Placement


key
component
message
information packet
port
Order Placement
in-Only
Messaging
Place Order
paid order
Eventing
confirm order
in-Out
order for restaurant
Messaging
Front of House
Eventing
Message Triggers Fault
Messaging
Eventing
in-Out
confirm order
Cook Food
Eventing
in-Out
Messaging
Book Courier
Eventing

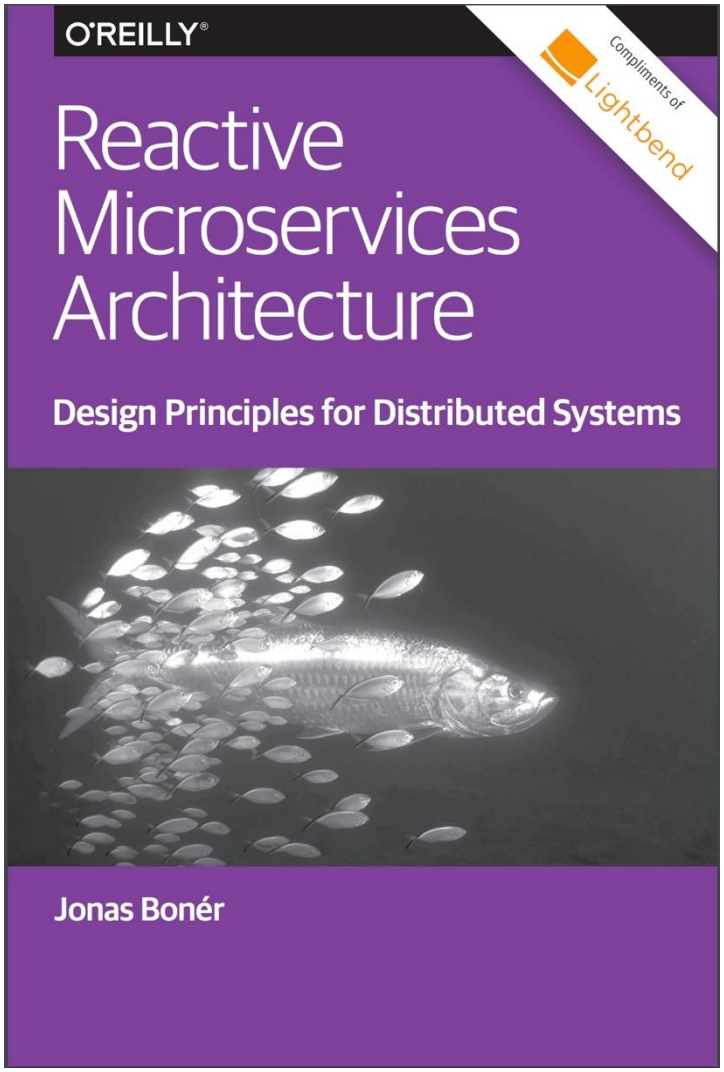

Restaurant Onboarding
New Catalogue
Let's simplify and ignore things like reprographics, we just want to show that this produces a new catalogue.
New Catalogue
To make a selection, we use the menu, so that we don't need to phone the restaurant to ask what food is available
New Catalogue
Eventual Consistency?
Sometimes customers have an out-of-date catalogue - so we have to validate their order against our latest catalogue
Order
At this point we have a valid order, now we need to get one of our restaurants to cook it!
Order Taking
Order Confirmation
Phone call
Confirm Order
Simplification
We are not going to show:
- The Order on the way to the customer
- The Order reaching the customer
Order Confirmation


Device
onboard restaurant
Qualify Restaurant
qualified restaurant
Request Details
request details
restaurant details
Make Catalogue
add_to catalogue
new catalogue
fax out
fax in
lookup catalogue
catalogue results
Take Order
order food
priced order
Device
Take Payment
card reader out
card reader in
Device
Place Order
paid order
confirmed order
order for restaurant
order confirmation
Front of House
food order
Cook Food
time to cook
reject order
book delivery
Book Courier
confirm order
key
component
message
information packet
port
Order Flow

NEXT STEPS

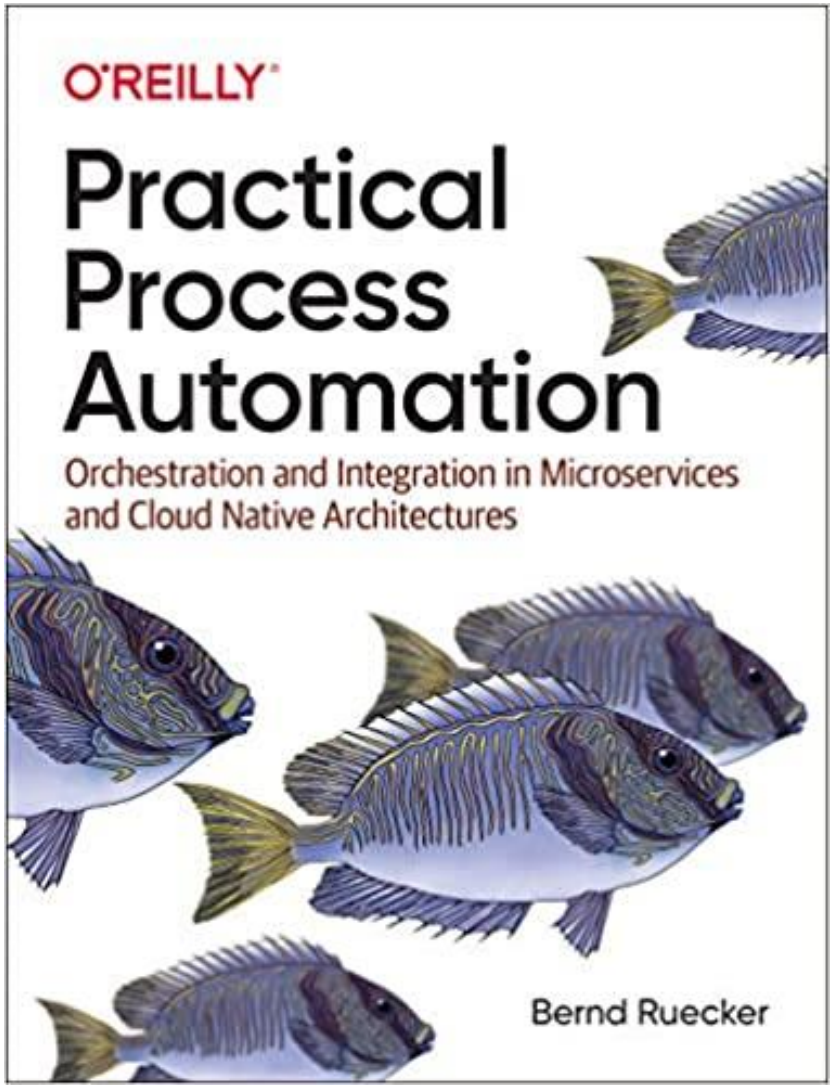
Further Reading

https://jonasboner.com/resources/Reactive_Microservices_Architecture.pdf

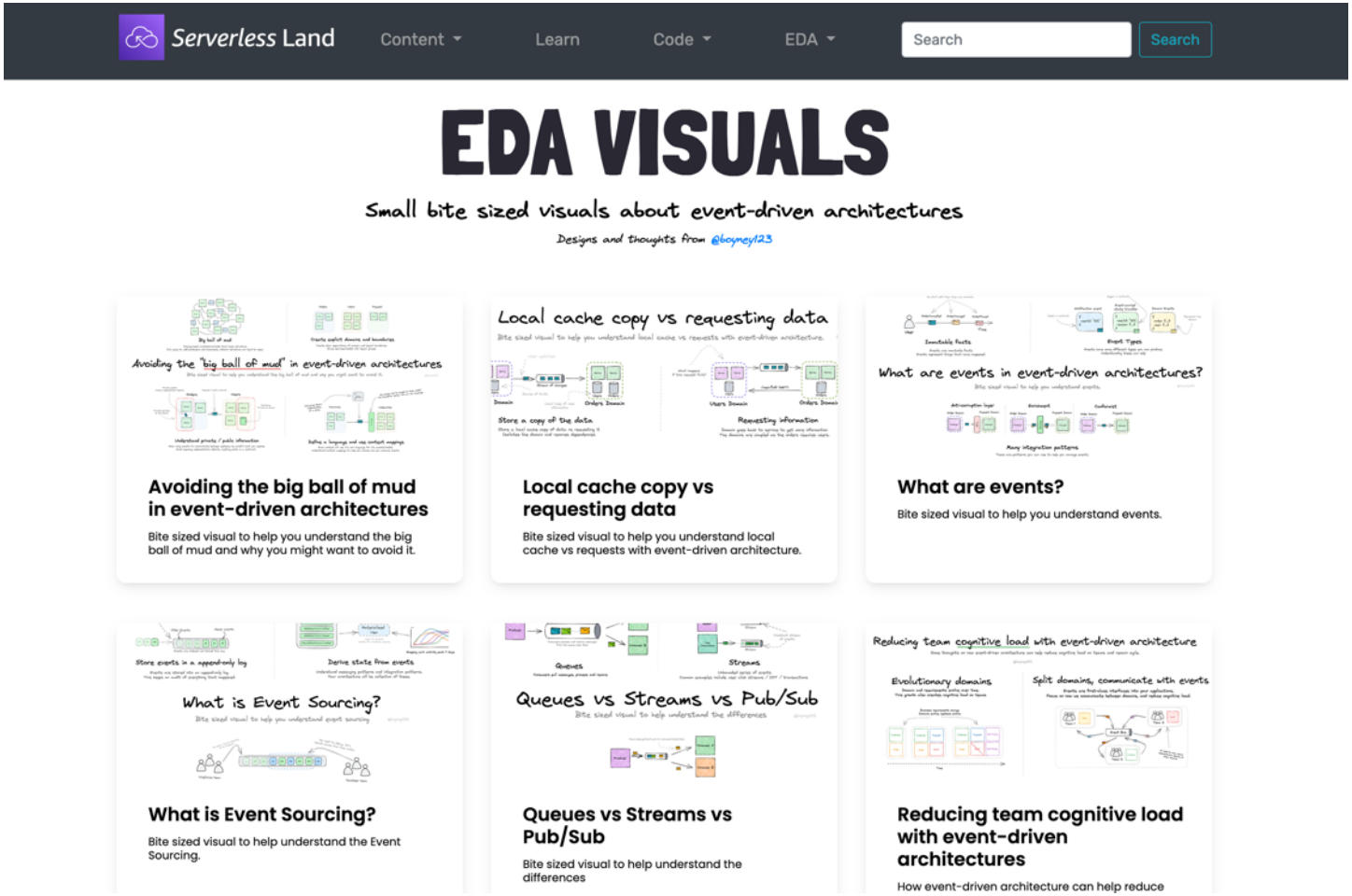


Further Reading

https://processautomationbook.com/



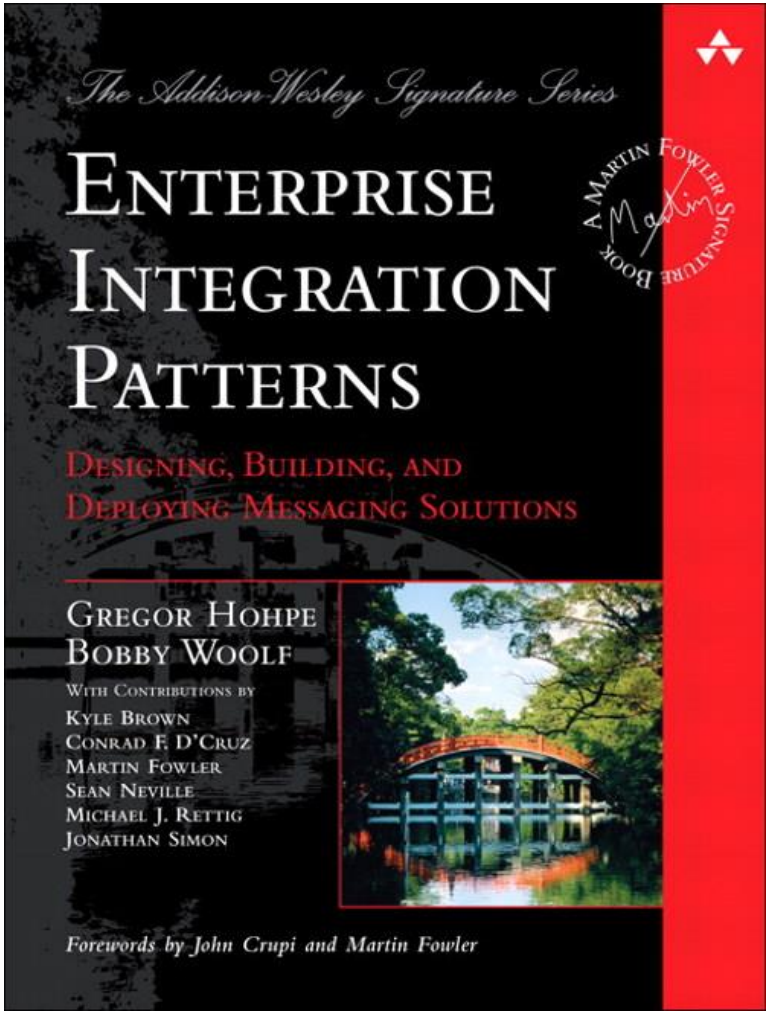
Further Reading



https://serverlessland.com/event-driven-architecture/visuals

Further Reading

https://www.enterpriseintegrationpatterns.com/gregor.html



Q&A