Asynchronous Messaging
with ZMQ

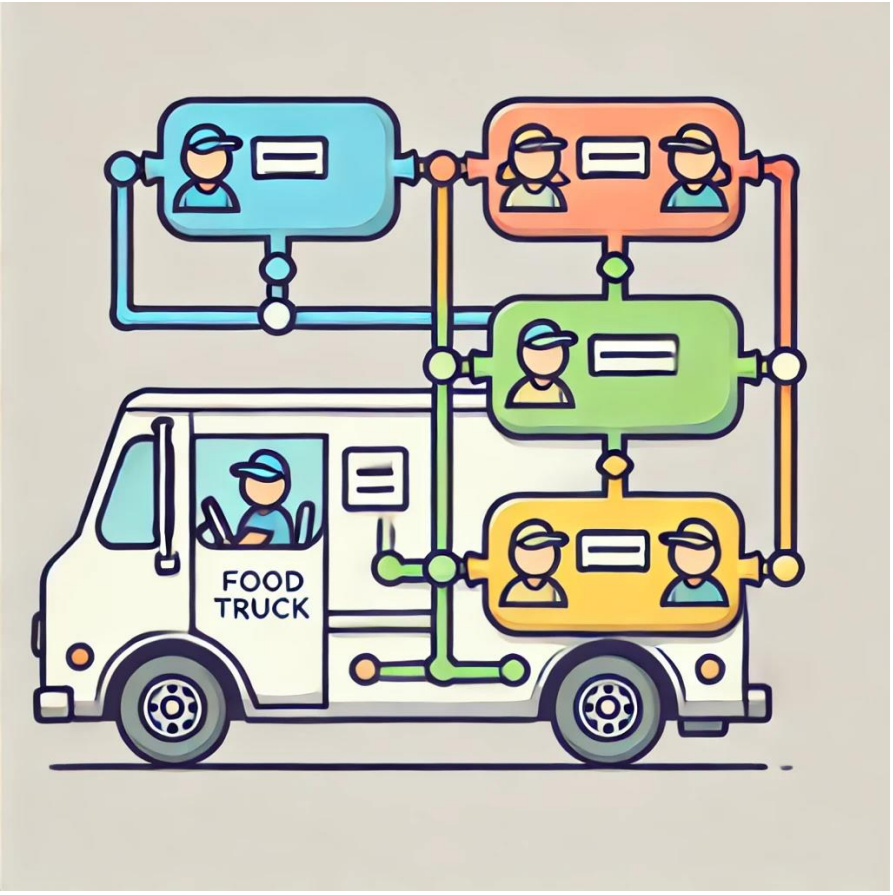
Julius Parulek
DSD SOFTWARE ENGINEERING 2
Scout (Ert, Everest, …, FMU)
Bergen

Asynchronous Messaging

with ZMQ (for running a foodtruck)

Julius Parulek

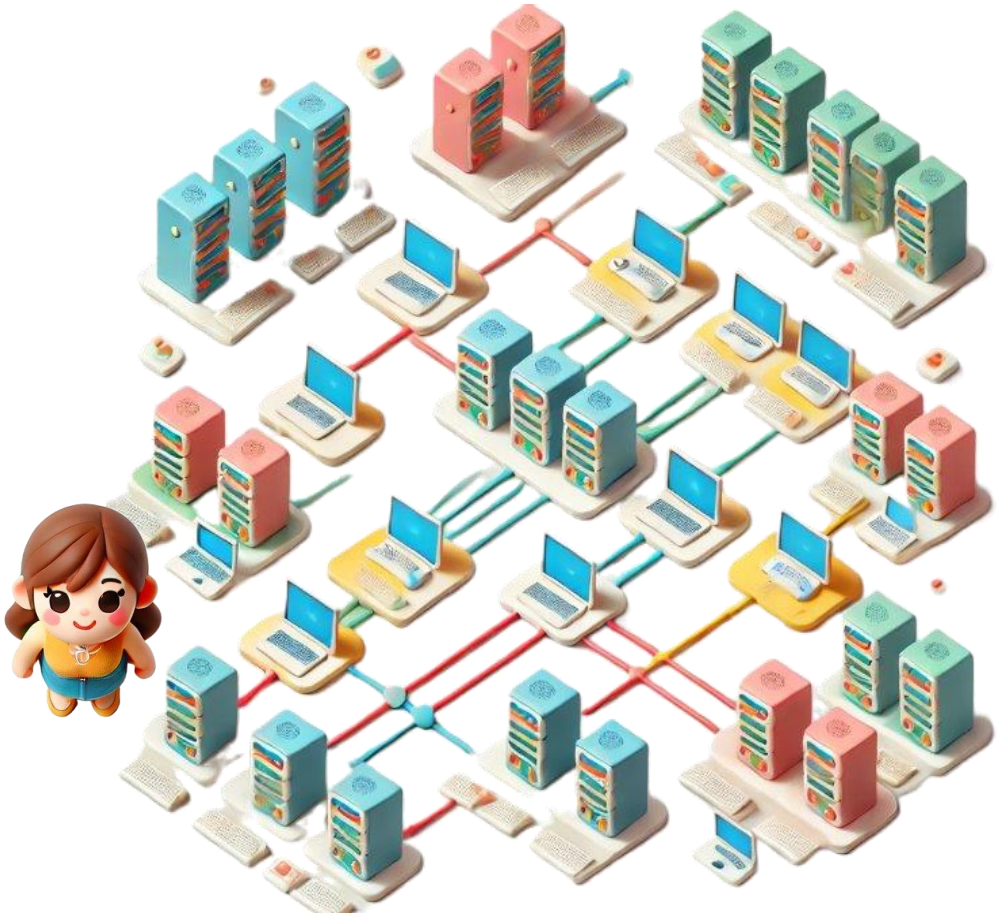
DSD SOFTWARE ENGINEERING 2
Scout (Ert, Everest, …, FMU)
Bergen



Background

Distributive systems

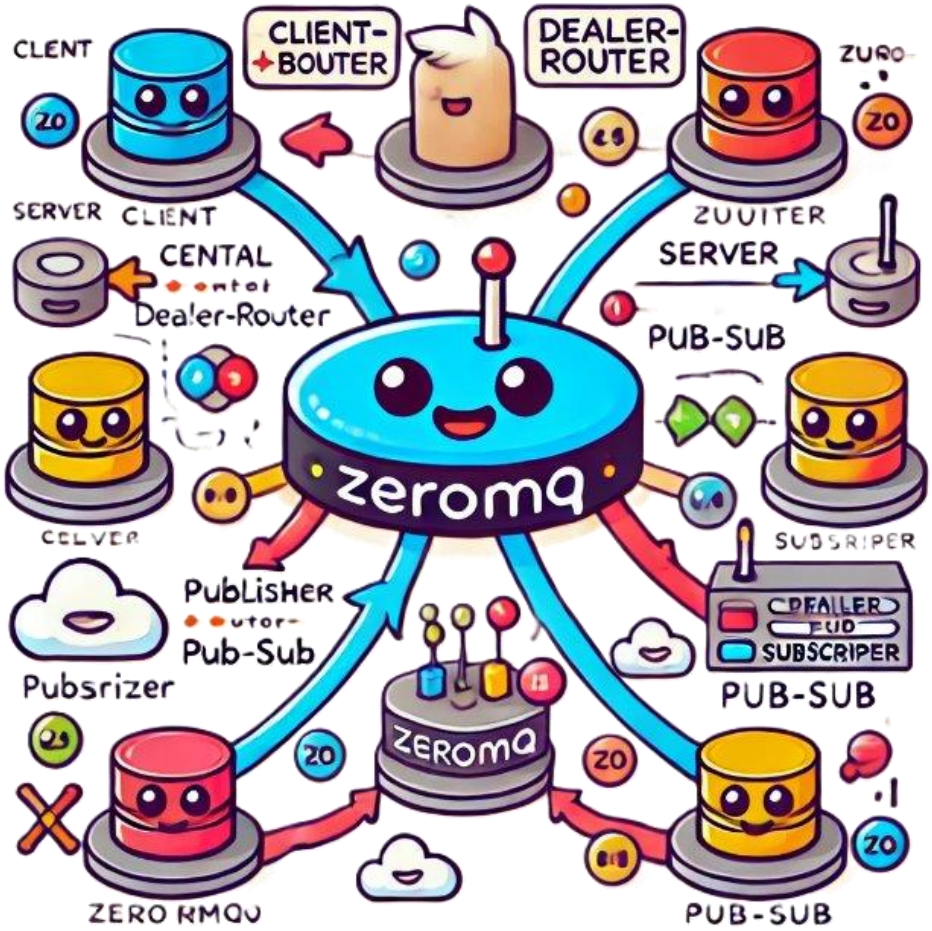
- Complex system of independent compute resources that appear as a single coherent system to the user
- **Features**
  - scalability
  - fault tolerance
  - distributive computation
  - distributive storage
  - resilient and efficient processing



Background

ØMQ

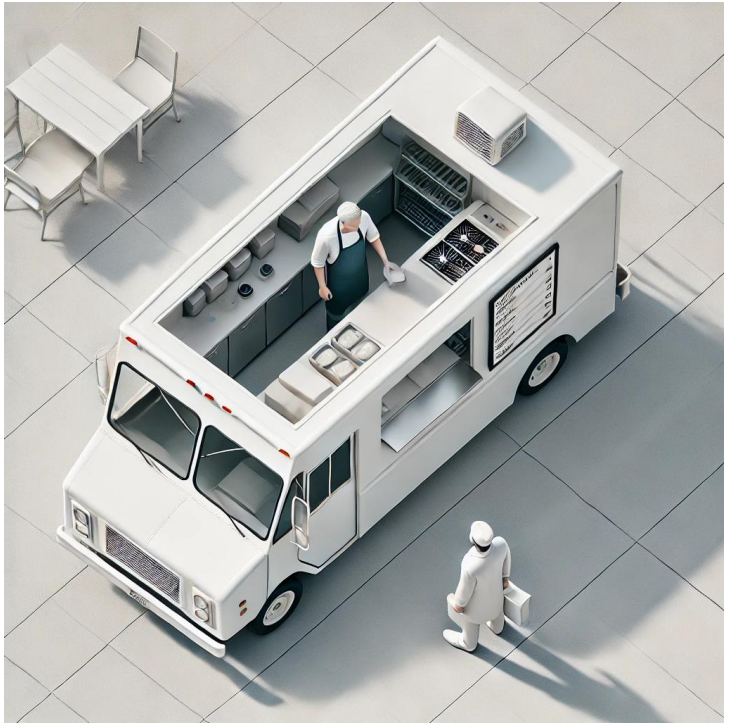
- High-Performance Messaging Library
- Socket Abstractions
  - Support different protocols
- **Asynchronous** design
- Built-in **messaging patterns**
- Peer-to-Peer Architecture
  - **Brokerless**
- Language Agnostic
  - C++, Java, Rust, Ruby, .Net, **Python**, ...



Background

Foodtruck

- (chatgpt) imagine a restaurant and a delivery van had a **baby that makes delicious food**
- (wiki) A food truck is a large motorized vehicle or trailer equipped to **store, transport, cook, prepare, serve**, and/or sell food



schematics

schematics

2. Ingredients

3. Cook

1. Order

4. Prepare

schematics

2. Ingredients

3. Cook

1. Order

4. Prepare

schematics

2. Ingredients

3. Cook

1. Order

4. Prepare

schematics

2. Ingredients

3. Cook

1. Order

4. Prepare

Request-Reply

Request-Reply

- Bidirectional communication
- Synchronous
- Simple

Request-Reply

Request-Reply

Request-Reply

Request-Reply

Request-Reply

2. Ingredients

3. Cook

1. Order

4. Prepare

Reply

Order {"item":"hotdog", "status":"done"}

Request-Reply

Request-Reply

```python
def customer():
    context = zmq.Context()
    socket = context.socket(zmq.REQ)
    socket.connect("tcp://localhost:5555")

    while True:
        order = {"item": "hotdog"}
        socket.send_json(order)
        order = socket.recv_json()
```


```python
def foodtruck():
    context = zmq.Context()
    socket = context.socket(zmq.REP)
    socket.bind("tcp://*:5555")

    while True:
        order = socket.recv_json()

        print(f"Received order: {order}")
        order = do_order(order)
        order = ingredients(order)
        order = cook(order)
        order = prepare(order)

        socket.send_json(order)
```

Request-Reply

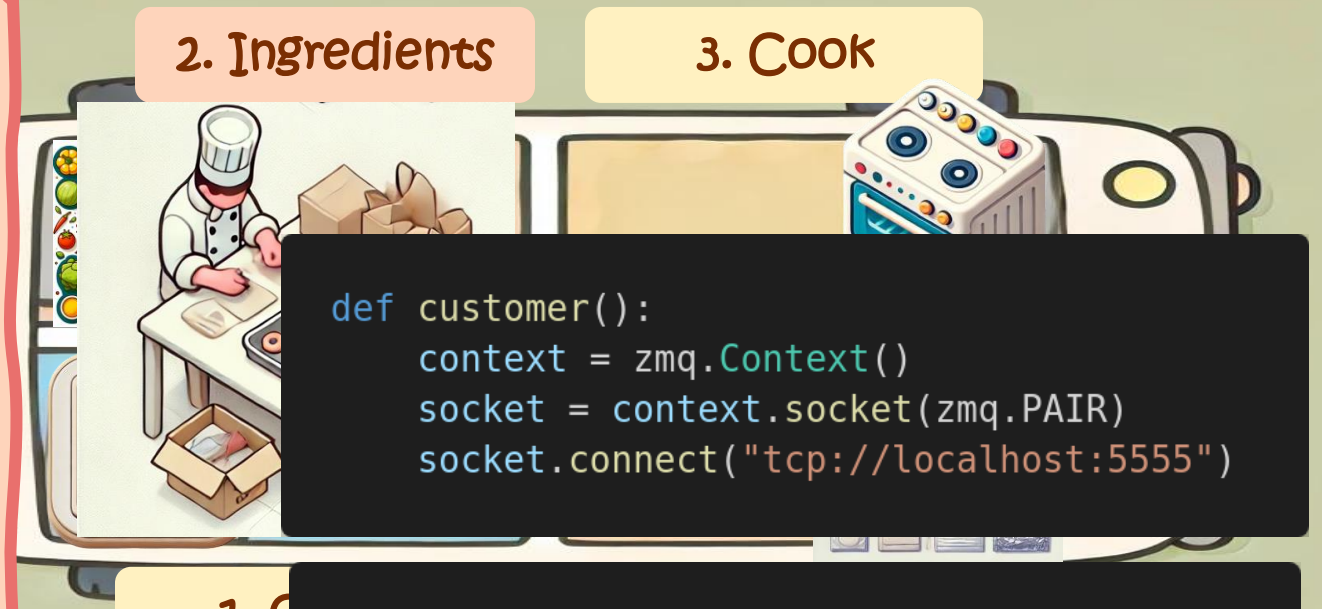
So ...

- **Pros**
  - Elegant & simple
  - Stateful
- **Cons**
  - Fully synced
  - Not resilient
  - Bad for multiple receivers
    - Socket blocked until replies comes

Exclusive Pair

Extension to Request-Reply

- Socket is **not** blocked




```
def customer():
    context = zmq.Context()
    socket = context.socket(zmq.PAIR)
    socket.connect("tcp://localhost:5555")
```

```
def foodtruck():
    context = zmq.Context()
    socket = context.socket(zmq.PAIR)
    socket.bind("tcp://*:5555")
```

Exclusive Pair

What about efficiency?

- Step-1 - 20 secs
- Step-2 - 20 secs
- Step-3 - 180 secs
- Step-4 - 20 secs
- 1 transaction
  - 240 secs = 4 mins

Exclusive Pair

What about efficiency?

- Step-1 - 20 secs
- Step-2 - 20 secs
- Step-3 - 180 secs
- Step-4 - 20 secs
- 1 transaction
  - 240 secs = 4 mins
  - 16 customers in 1 hour

More workers!

2. Ingredients

3. Cook

1. Order

4. Prepare

More workers with Req-Rep/Pair

More workers with Req-Rep/Pair

So ...

- **Order** wants to say something to **Ingredients**
  - **Ingredients** needs to stop & reply
  - **Ingredients** can be busy, **Order** needs to wait

More workers with Req-Rep/Pair

Req/Rep/Pair is not great for multiple connections and workers

More workers

What we need?

- **Asynchronous** messaging & execution
  - **Push / Pull**
  - **Publish / Subscribe**
  - **Router / Dealer**

Push-Pull

Push-Pull

- Distribute jobs across many workers
- **Scatter-Gather**
- Load balancing
- Work parallelization
- Messages **are not duplicated**

Push-Pull

{"**item**":"hotdog",
"**do_stage**": "ing"}

Master queue

Order {"item":"hotdog"}

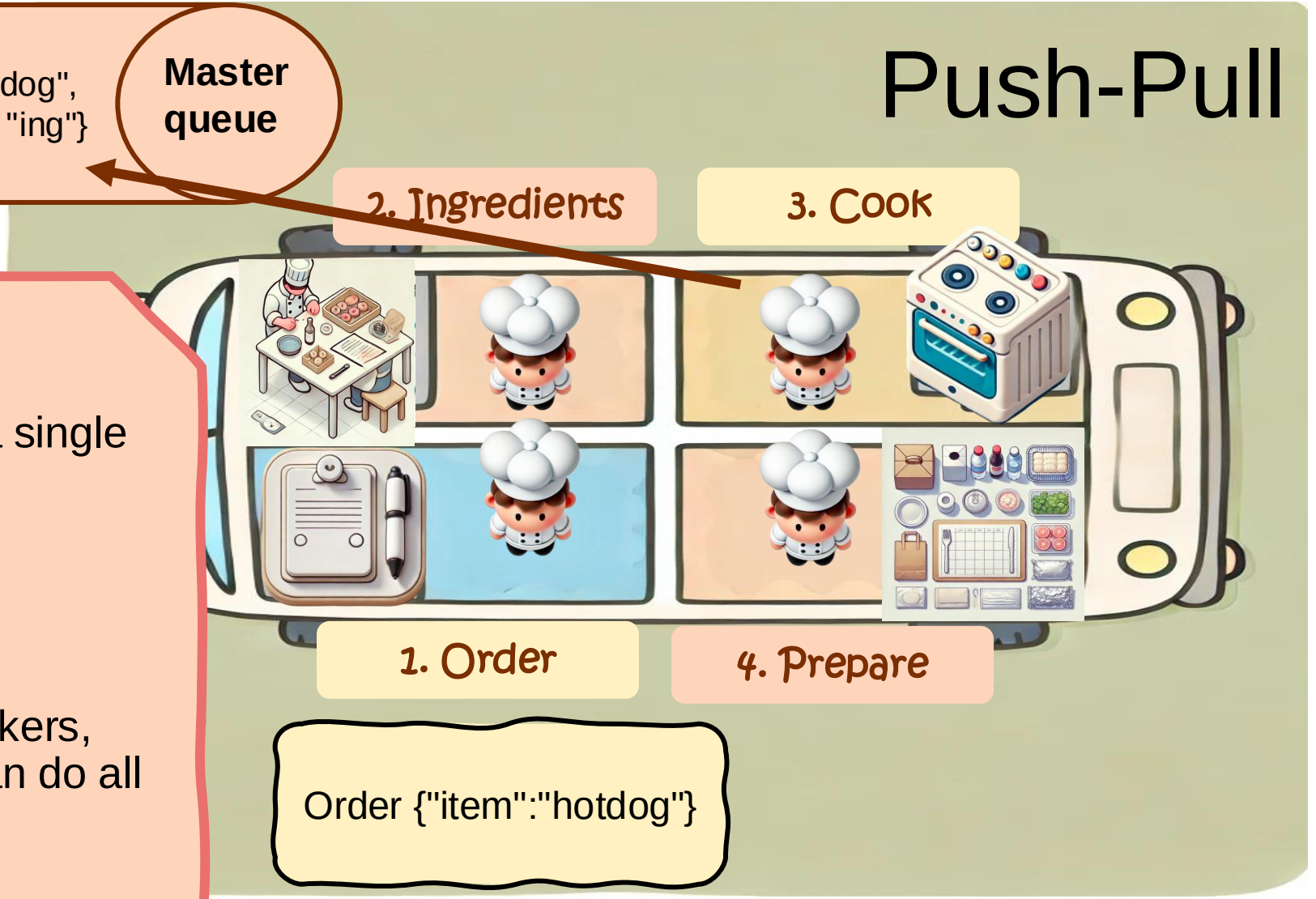
So what about a single **master queue** managed by **order stage ?**

- All can pull
- Flexible workers, each one can do all 4 jobs

Push-Pull

{"**item**":"hotdog",
"**do_stage**": "ing"}

Master queue



So what about a single **master queue** managed by **order stage ?**

- All can pull
- Flexible workers, each one can do all 4 jobs

Push-Pull

{"**item**":"ice-cream", "**do_stage**": "ing"}

{"**item**":"hotodg", "**do_stage**": "ing"}

Master queue

So what about a single **master queue** managed by **order stage ?**

- All can pull
- Flexible workers, each one can do all 4 jobs

Push-Pull

{"**item**":"ice-cream", "**do_stage**": "ing"} {"**item**":"hotodg", "**do_stage**": "ing"}

Master queue

So what about a single **master queue** managed by **order stage ?**

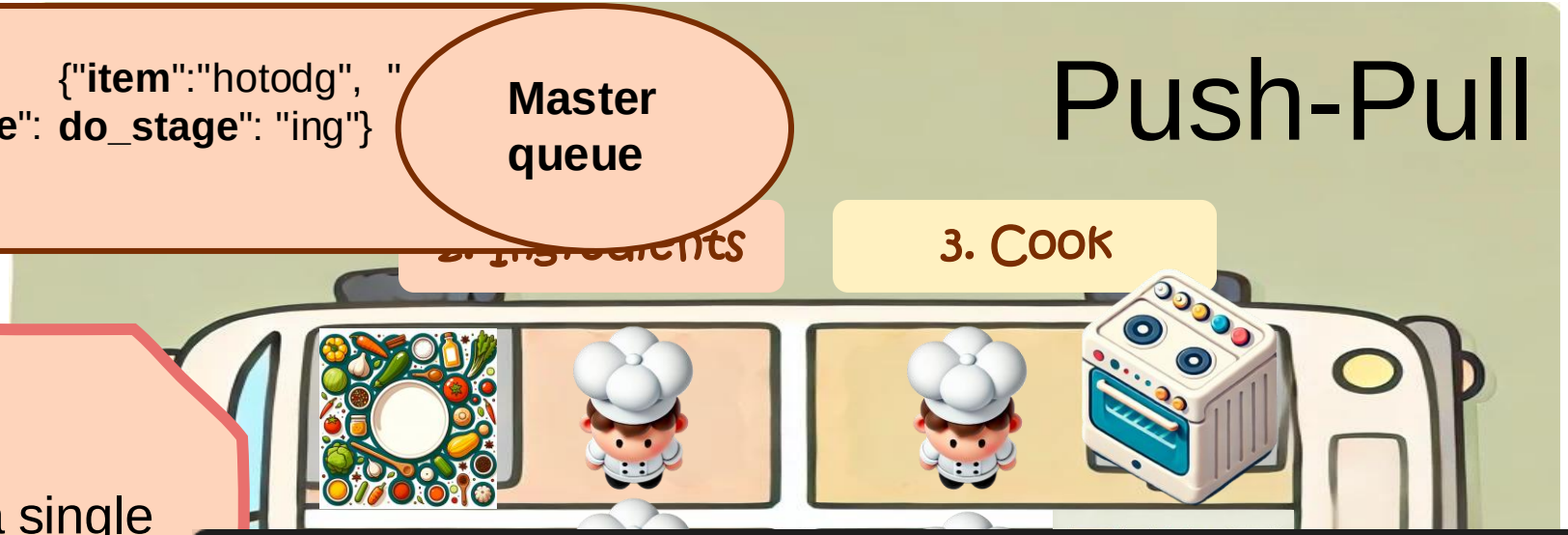
- All can pull
- Flexible workers, each one can do all 4 jobs

Push-Pull

{"**item**":"ice-cream", "**do_stage**": "ing"}

{"**item**":"hotodg", "**do_stage**": "ing"}

Master queue



So what about a single **master queue** managed by **order stage ?**

- All can pull
- **Flexible** workers, each one can do all 4 jobs

```
while True:
    order = await master_queue.recv_json()
    worker_name = order["do_stage"]
    while not free_stage(worker_name):
        await asyncio.sleep(0.1)
    order = await do_work(worker_name, order)
    master_queue.send_json(order)
```

h-Pull

Tight coupling between stages!

So what ab
m

Push-Pull for jobs of the same type

ers,
one can do all 4 jobs

```
while True:
    order = await master_queue.recv_json()
    worker_name = order["do_stage"]
    while not free_stage(worker_name):
        await asyncio.sleep(0.1)
    order = await do_work(worker_name, order)
    master_queue.send_json(order)
```

Push-Pull

So what about a message **queue** between every **stage ?**

- Ex. **Ingredients** queue
  - **Order stage** does push
  - **Ingredients stage** does pull

Push-Pull

Cook

2. Ingredients

3. Cook

Ing.

Prep

1. Order

4. Prepare

Order

Push-Pull

Cook

2. Ingredients

3. Cook

Ing.

Prep

1. Order

4. Prepare

Request

Order {"item":"hotdog"}

Order

Push-Pull

Cook

2. Ingredients

3. Cook

Ing.

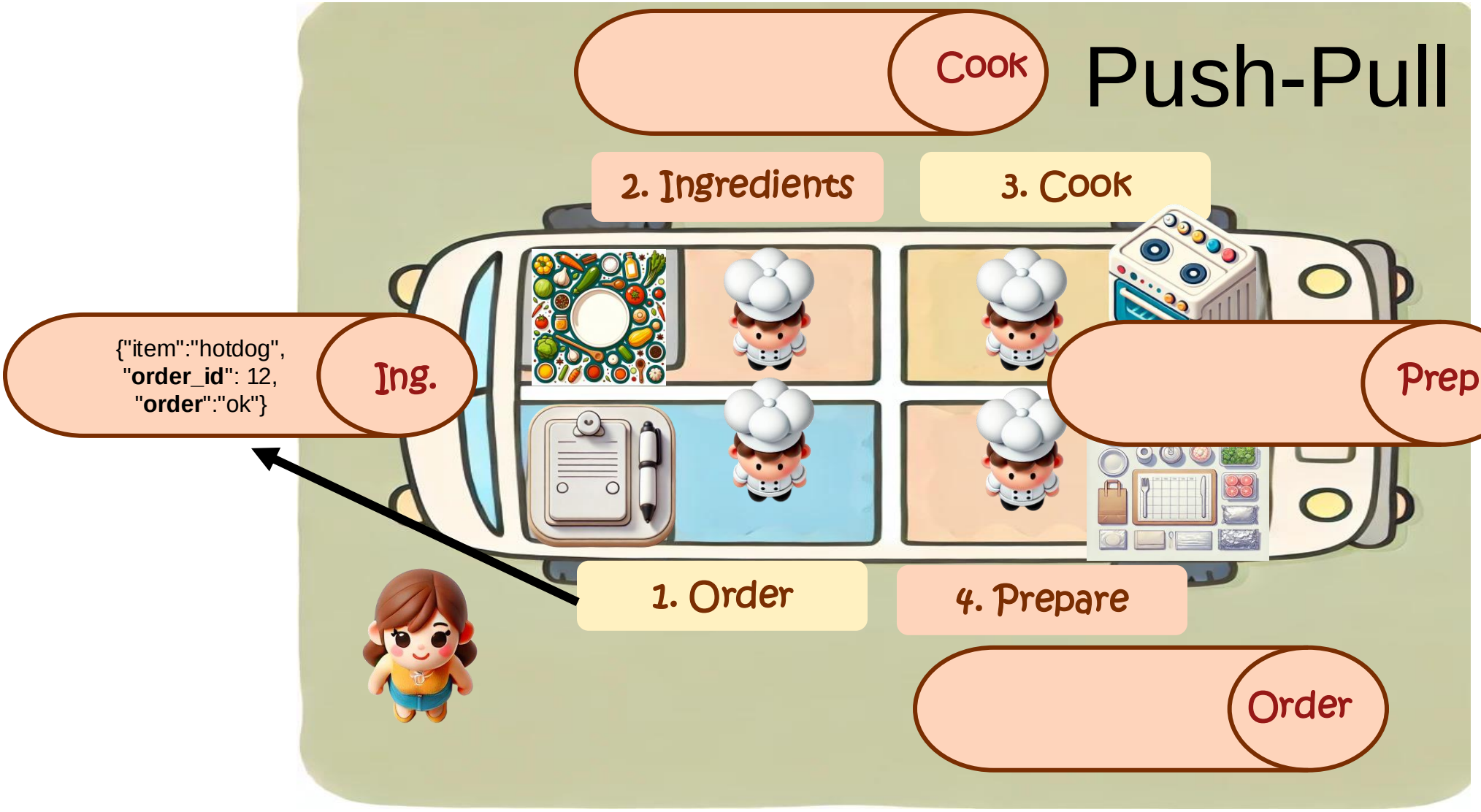
Prep

1. Order

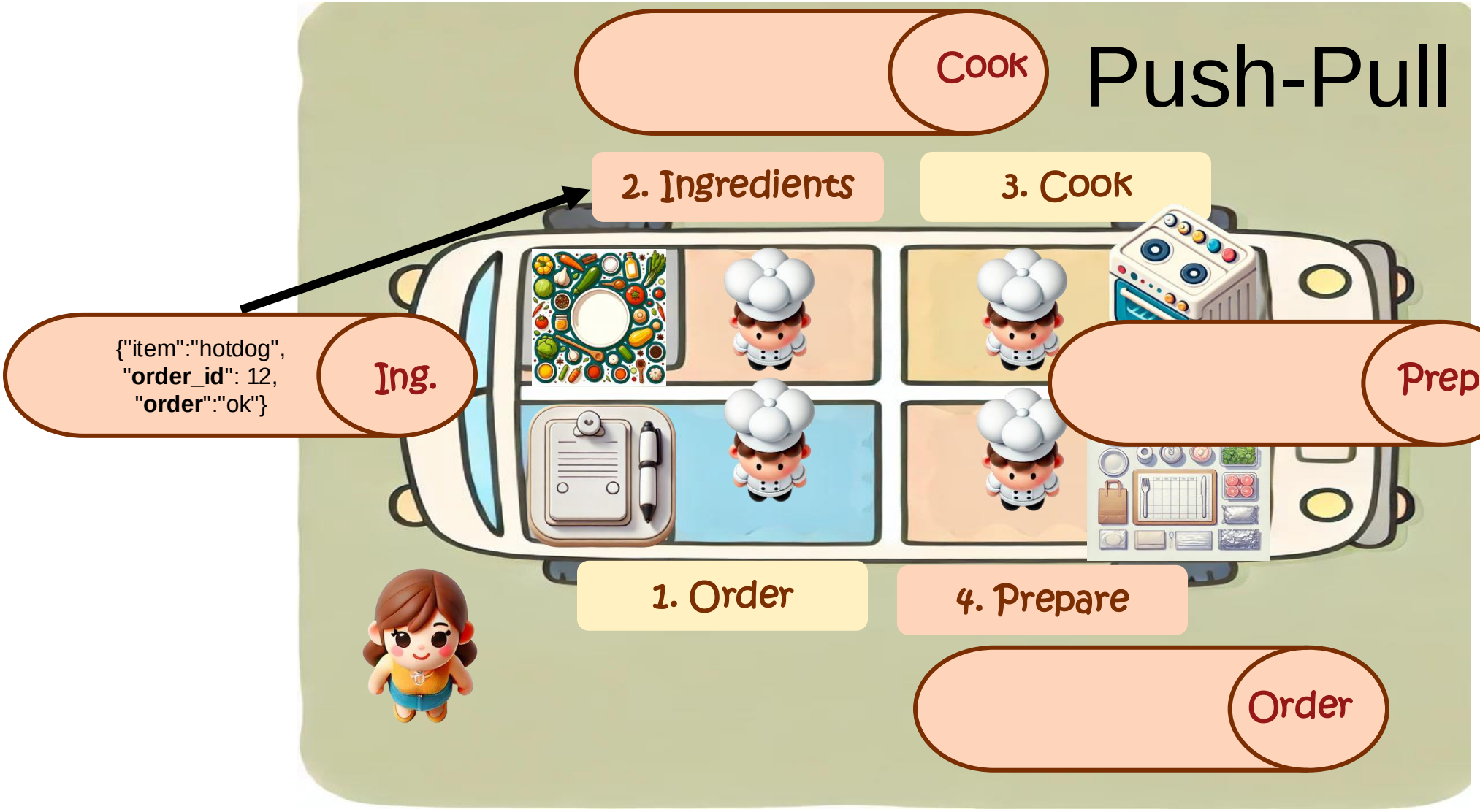
4. Prepare

Order

Push-Pull


Cook
2. Ingredients
3. Cook
{"item":"hotdog",
"order_id": 12,
"order":"ok"}
Ing.
Prep
1. Order
4. Prepare
Order

Push-Pull


Cook
2. Ingredients
3. Cook
{"item":"hotdog",
"order_id": 12,
"order":"ok"}
Ing.
Prep
1. Order
4. Prepare
Order

Push-Pull

Cook

2. Ingredients

3. Cook

Ing.

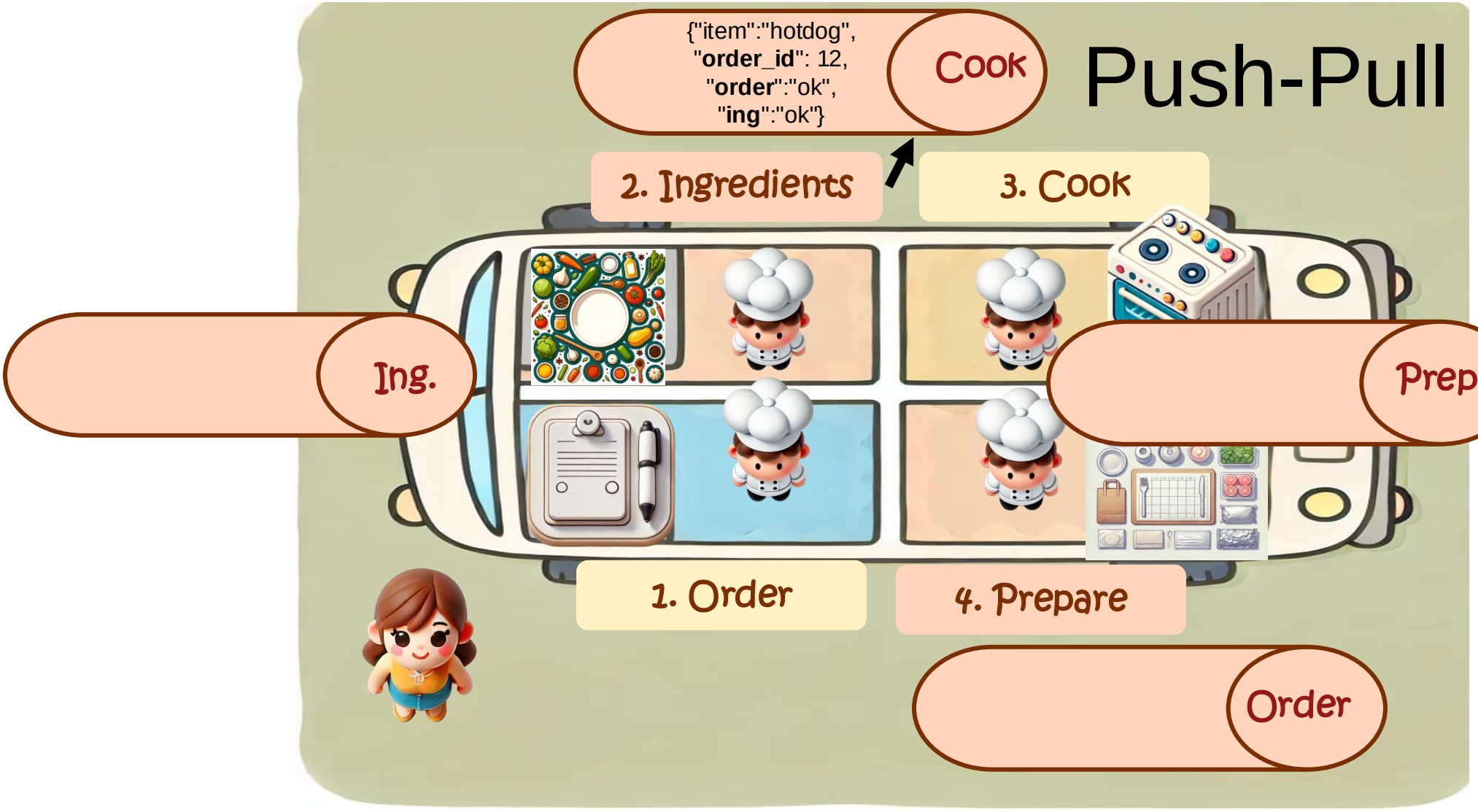
Prep

1. Order

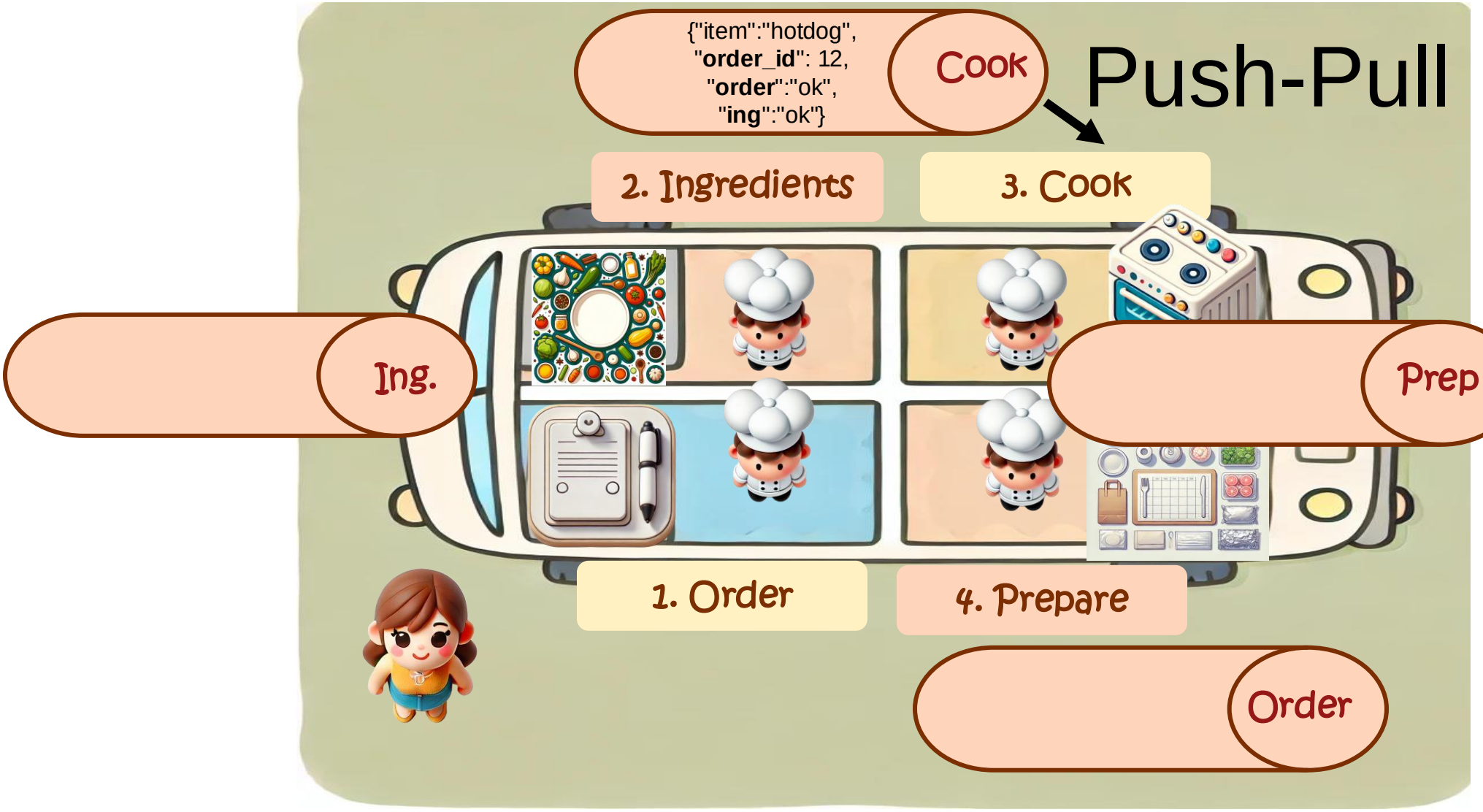
4. Prepare

Order

Push-Pull


{"item":"hotdog",
"order_id": 12,
"order":"ok",
"ing":"ok"}
Cook
2. Ingredients
3. Cook
Ing.
Prep
1. Order
4. Prepare
Order

Push-Pull


{"item":"hotdog",
"order_id": 12,
"order":"ok",
"ing":"ok"}
Cook
2. Ingredients
3. Cook
Ing.
Prep
1. Order
4. Prepare
Order

Push-Pull

Cook

2. Ingredients

3. Cook

Ing.

Prep

1. Order

4. Prepare

Order

Push-Pull

Cook

2. Ingredients

3. Cook

Ing.

{"item":"hotdog",
"**order_id**": 12,
"**order**":"ok",
"**ing**":"ok", "**cook**":"ok"}

Prep

1. Order

4. Prepare

Order

Push-Pull

Cook

2. Ingredients

3. Cook

Ing.

Prep

1. Order

4. Prepare

Order

Push-Pull

Cook

2. Ingredients

3. Cook

Ing.

Prep

1. Order

4. Prepare

{"item":"hotdog",
"**order_id**": 12,
"**order**":"ok", "**ing**":"ok",
"**cook**":"ok", "**prep**":"ok"}

Order

Push-Pull

Cook

2. Ingredients

3. Cook

Ing.

Prep

1. Order

4. Prepare

Reply

Order {"item":"hotdog", "status":"complete"}

Order

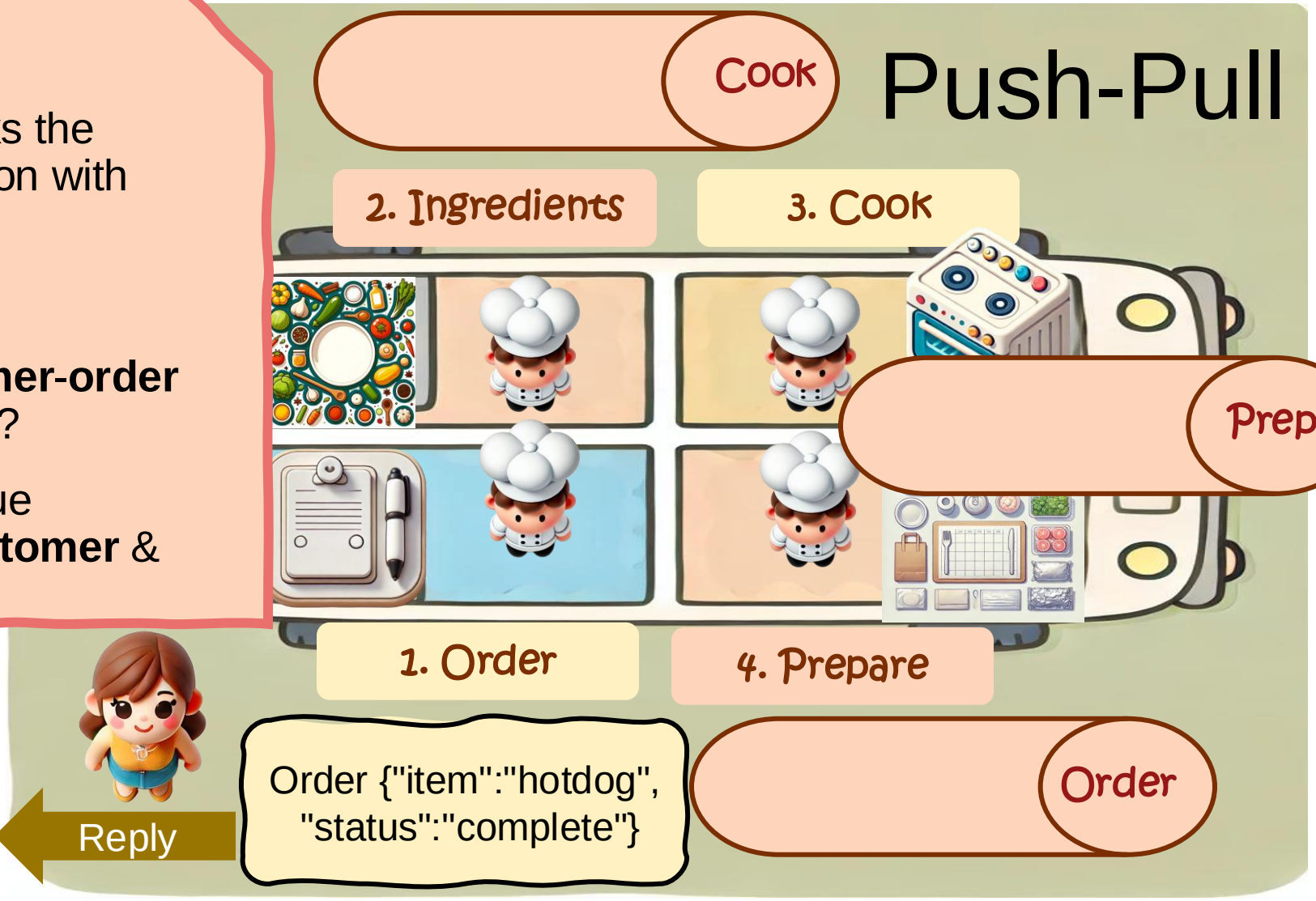
But ...

- req-rep blocks the communication with customers

Fix ...

- More **customer-order** req-rep ports?
- Another queue between **customer** & **order**?

Push-Pull



Push-Pull

Cook

Ing.

2. Ingredients

3. Cook

Prep

{"**item**":"hotdog"}

Order

1. Order

4. Prepare

Cust.

Push-Pull

Cook

{"**item**":"hotdog",
"order_id": 12,
"order":"ok"}

Ing.

2. Ingredients

3. Cook

Prep

Order

1. Order

4. Prepare

Cust.

Push-Pull

{"**item**":"hotodog",
"order_id": 12,
"order":"ok", "ing":"ok"}

Cook

Ing.

2. Ingredients

3. Cook

Prep

{"**item**":"ice-cream"}

Order

1. Order

4. Prepare

Cust.

Push-Pull

Cook

{"**item**":"ice-cream",
"order_id": 13,
"order":"ok"}

Ing.

2. Ingredients

3. Cook

Prep

Order

1. Order

4. Prepare

{"**item**":"hotdog",
"order_id": 12,
"order":"ok", "ing":"ok",
"cook":"ok", "prep":"ok"}

Cust.

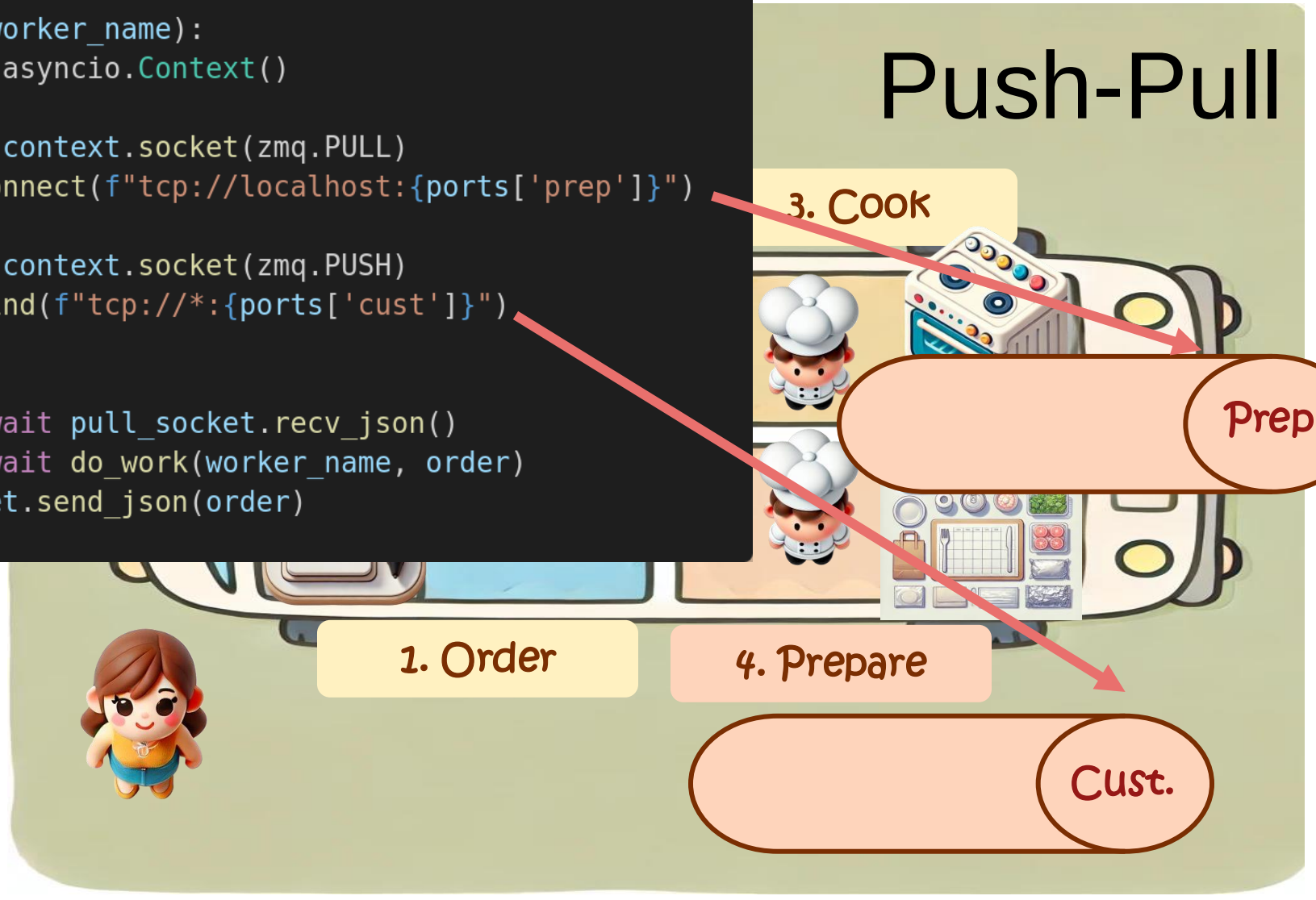
```python
async def worker(worker_name):
    context = zmq.asyncio.Context()

    pull_socket = context.socket(zmq.PULL)
    pull_socket.connect(f"tcp://localhost:{ports['prep']}")

    push_socket = context.socket(zmq.PUSH)
    push_socket.bind(f"tcp://*:{ports['cust']}")

    while True:
        order = await pull_socket.recv_json()
        order = await do_work(worker_name, order)
        push_socket.send_json(order)
```

Push-Pull



Push-Pull

Cook

{"**item**":"ice-cream",
"order_id": 13,
"order":"ok"}

Ing.

2. Ingredients

3. Cook

Prep

Order

1. Order

4. Prepare

Cust.

Push-Pull

{"**item**":"ice-cream",
"order_id": 13,
"order":"ok", "ing":"ok"}

Cook

{"**item**":"ice-cream",
"order_id": 13,
"order":"ok"}

Skip!

Order

2. Ingredients

3. Cook

Prep

1. Order

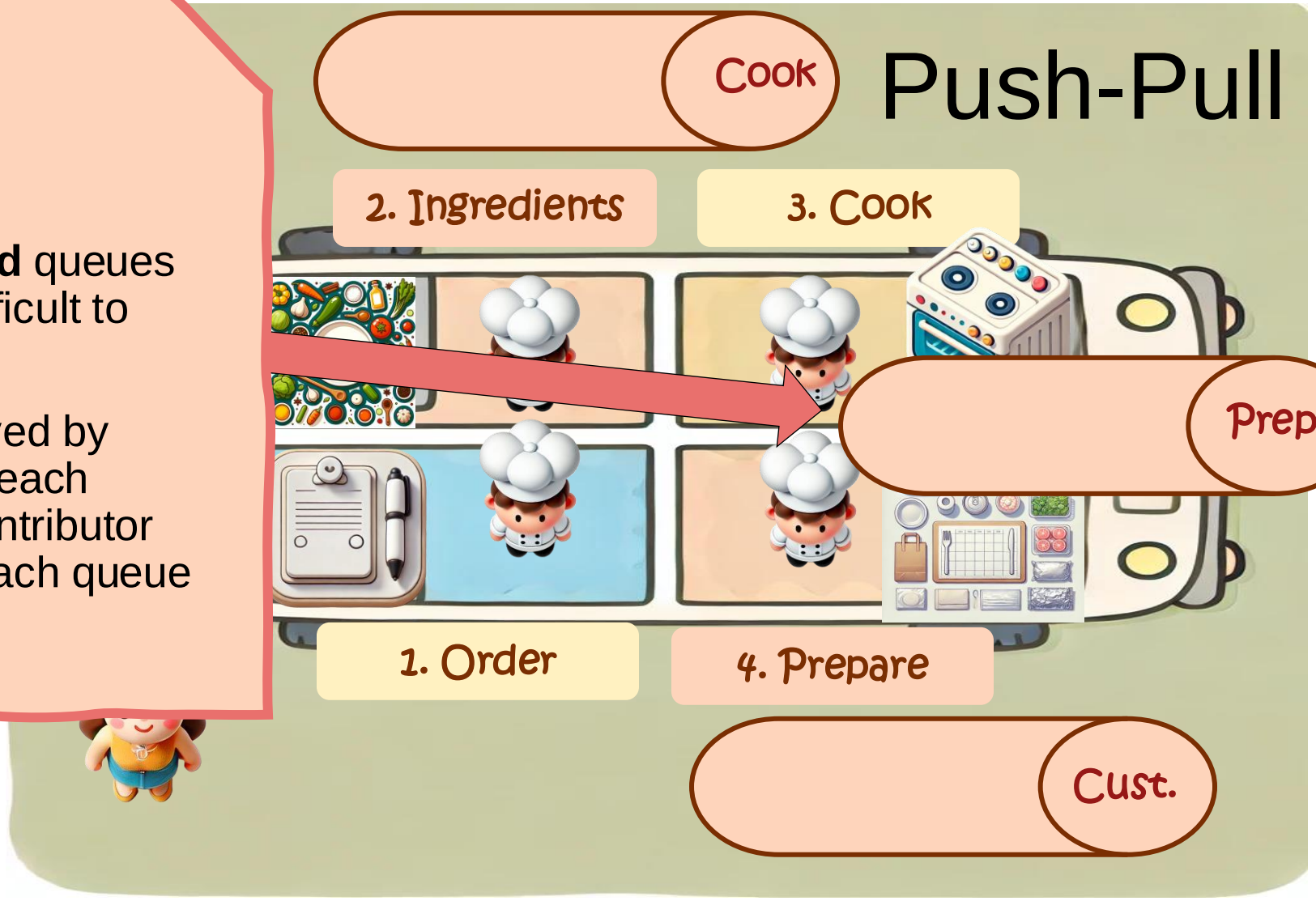
4. Prepare

Cust.

Push-Pull

So ...

- Having **fixed** queues makes it difficult to skip **stages**
- Can be solved by connecting each stage as contributor (**push**) to each queue



Push-Pull

So ...

- **Pros**
  - Elegant & simple
  - Stateless
  - Scalable
  - **Parallelization**
  - Fully async
- **Cons**
  - Foodtruck **fails** since a stage can fail and the **customer** is not informed
  - Integrating a new or skipping stages (ice-cream)
  - Customer's direct access to worker
  - **Jobs of the same type**

Publish-Subscribe

Features

- **One** sender, **multiple** receivers
- Subscribers can filter out messages – **topics**
- A new actor **- chef**

all:

chef:

Publish-Subscribe

Publish-Subscribe

all:

[cust, order, ing, cook, prep]

chef:

topic: order

{"**item**":"hotdog"}

[chef]

Publish-Subscribe

all:

topic: **order**
{"item":"hotdog"}

[cust, order, ing, cook, prep]

chef:

[chef]

Publish-Subscribe

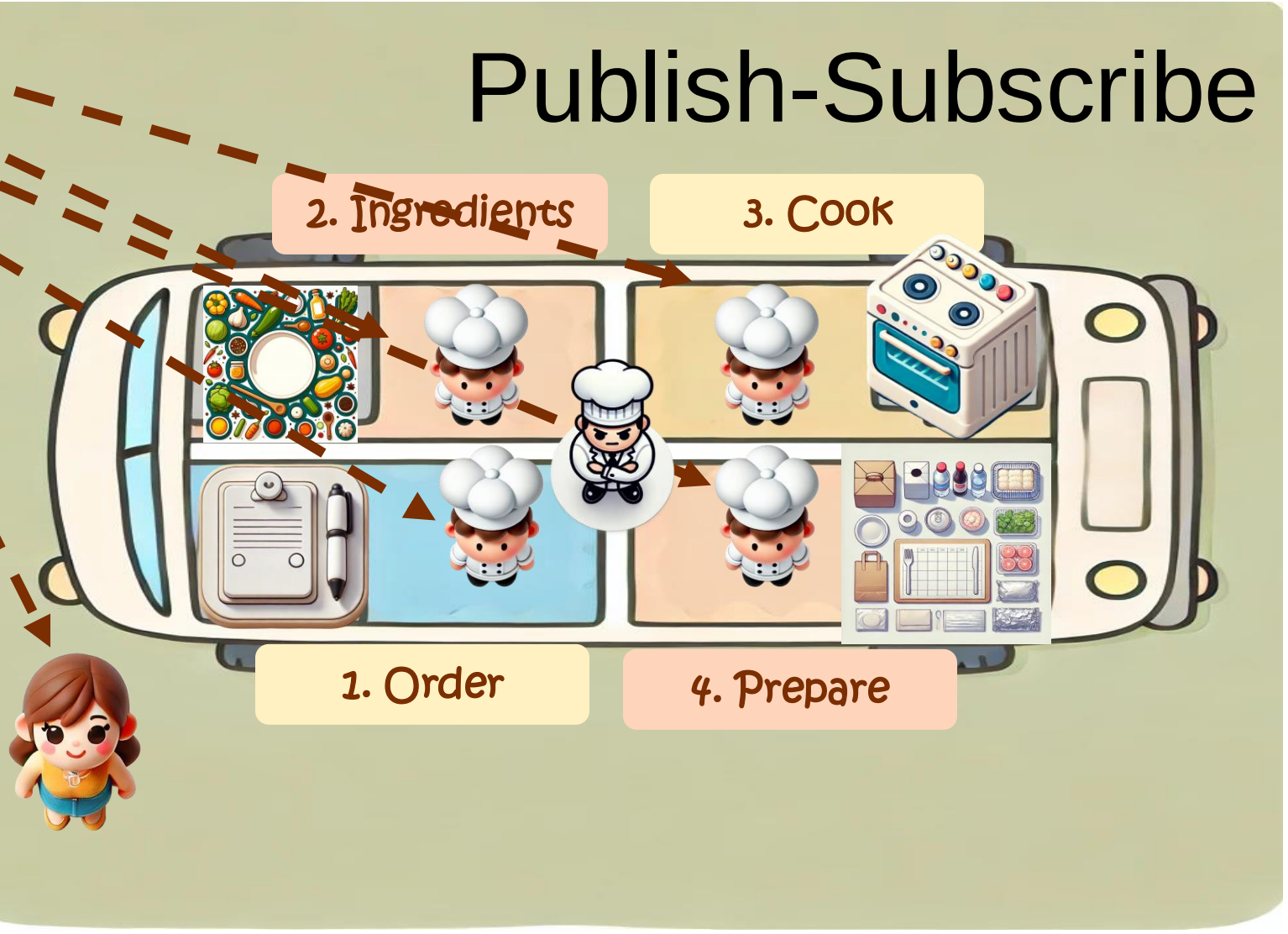
all:

topic: **order**
{"item":"hotdog"}

[cust, order, ing, cook, prep]

chef:

[chef]



Publish-Subscribe

all:

topic: **order**
{"item":"hotdog"}

[cust, order, ing, cook, prep]

chef:

[chef]

Publish-Subscribe

unit:

[cust, order, ing, cook, prep]

chef:

topic: ingredients
{"item":"hotdog",
"**order_id**":12,
"**order**":"ok"}

Publish-Subscribe

all:

topic:
ingredients
{"item":"hotdog",
"**order_id**":12,
"**order**":"ok"}

[cust, order, ing, cook, prep]

chef:

topic:
cook
{"item":"hotdog",
"order_id":12,
"**order**":"ok",
"**ing**":"ok"}

[chef]

Publish-Subscribe

all:

topic:
ingredients
{"item":"hotdog",
"**order_id**":12,
"**order**":"ok"}

[cust, order, ing, cook, prep]

chef:

topic:
cook
{"item":"hotdog",
"**order_id**":12,
"**order**":"ok",
"**ing**":"ok"}

```
while True:  # ingredients job
    msg = await all_socket.recv_string()
    _, json_data = msg.split(" ", 1)
    order = json.loads(json_data)
    order = await do_work("ingredients", order)
    new_topic = "cook"
    json_data = json.dumps(order)
    await chef_socket.send_string(new_topic + " " + json_data)
```

1. Order

4. Prepare

Publish-Subscribe

all:

topic: **cook**
{"item":"hotdog",
"**order_id**":12,
"**order**":"ok",
"**ing**":"ok"}

[cust, order, ing, cook, prep]

chef:

topic:
prepare
{"item":"hotdog",
"**order_id**":12,
"**order**":"ok",
"**ing**":"ok",
"**cook**":"ok"}

[chef]

Publish-Subscribe

all:

topic:
prepare
{"item":"hotdog",
"**order_id**":12,
"**order**":"ok",
"**ing**":"ok",
"**cook**":"ok"}

[cust, order, ing, cook, prep]

chef:

topic: customer
{"item":"hotdog",
"**order_id**":12,
"**order**":"ok",
"**ing**":"ok",
"**cook**":"ok",
"**prep**":"ok"}

[chef]

Publish-Subscribe

all:

topic: **customer**
{"item":"hotdog",
"**status**":"done"}

[cust, order, ing, cook, prep]

chef:

topic: customer
{"item":"hotdog",
"**order_id**":12,
"**order**":"ok",
"**ing**":"ok",
"**cook**":"ok",
"**prep**":"ok"}

Publish-Subscribe

all:

topic: **ingredients**
{"item":"hotdog",
"order_id":12}

What-if?

[cus, order, ing, cook, prep]

chef:

topic: ingredients
{"item":"hotdog",
"order_id":12,
"**issue**":"no sausages"}

[chef]

Publish-Subscribe

all:

topic: **prepare**
{"item":"sausage",
"order_id":20,
"**cmd**":"buy 200"}

What-if?

[cust, order, ing, cook, prep]

chef:

[chef]

Publish-Subscribe

all:

topic: **ingredients**
{"item":"ice-cream",
"order_id":13}

What-if?

[cus, order, ing, cook, prep]

chef:

topic: cook
{"item":"ice-cream",
"order_id":13}

[chef]

Publish-Subscribe

all:

topic: **prepare**
{"item":"ice-cream",
"order_id":13}

What-if?

[cust, order, ing, cook, prep]

chef:

topic: cook
{"item":"ice-cream",
"order_id":13}

[chef]

Publish-Subscribe

So ….

- **Pros**
  - Elegant & simple
  - Stateless
  - **Scalable**
  - Loose coupling
  - Fully async
- **Cons**
  - Potential message loss if subscribers are offline
  - Messages are **broadcasted**

Router-Dealer

Features

- **Router** can connect to multiple **dealers**
- Fits complex distributive systems
- Highly scalable!

Router-Dealer

identities:
[customer, order, ingredients, cook, prepare]

Router-Dealer

identities:
[customer, order, ingredients, cook, prepare]

customer
{"item":"hotdog"}

Router-Dealer

2. Ingredients

3. Cook

1. Order

4. Prepare

identities:
[customer, order, ingredients, cook, prepare]

order
{"item":"hotdog"}

Router-Dealer

2. Ingredients

3. Cook

1. Order

4. Prepare

identities:
[customer, order, ingredients, cook, prepare]

order
{"item":"hotdog", "order_id":12, "order":"ok"}

Router-Dealer

identities:
[customer, order, ingredients, cook, prepare]

Router-Dealer

ingredients
{"item":"hotdog",
"order_id":12,
"order":"ok"}

Router-Dealer

identities:
[customer, order, ingredients, cook, prepare]

ingredients
{"item":"hotdog",
"order_id":12,
"order":"ok",
"ing":"ok"}

2. Ingredients

3. Cook

1. Order

4. Prepare

identities:
[customer, order, ingredients, cook, prepare]

cook
{"item":"hotdog",
"order_id":12,
"order":"ok",
"ing":"ok"}

Router-Dealer

2. Ingredients

3. Cook

1. Order

4. Prepare

Router-Dealer

identities:
[customer, order, ingredients, cook, prepare]

cook
{"item":"hotdog",
"order_id":12,
"ing":"ok",
"order":"ok"}


```
while True:  # cook
    [_, msg] = await dealer_socket.recv_multipart()
    order = json.loads(msg)
    order = await do_work("cook", order)
    separator = b""
    await dealer_socket.send_multipart([separator, json.dumps(order).encode()])
```

Router-Dealer

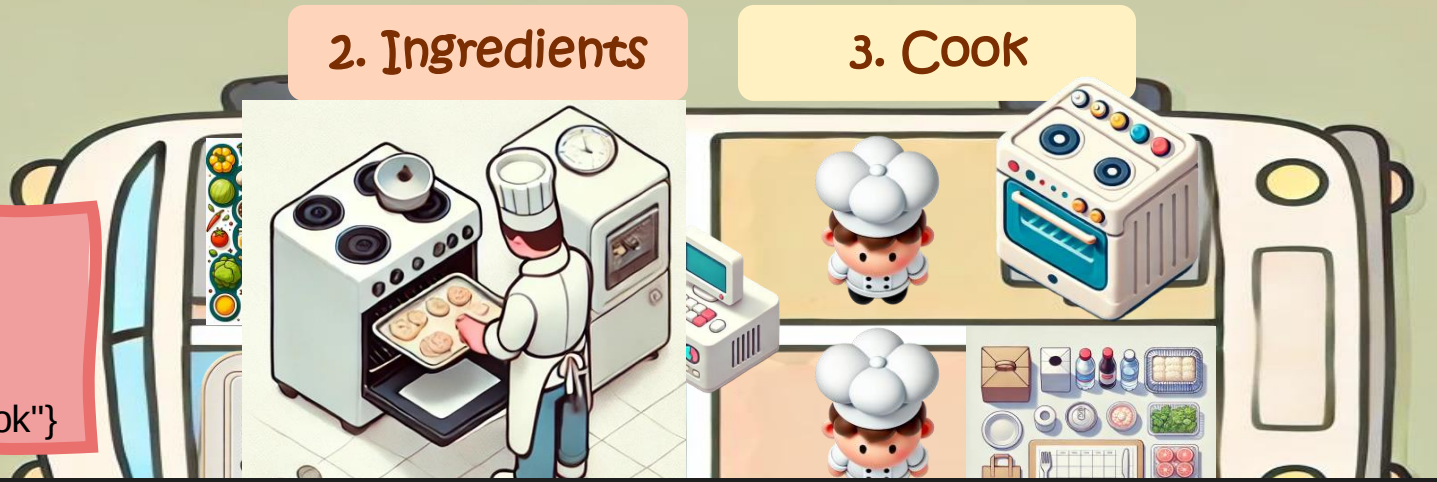
identities:
[customer, order, ingredients, cook, prepare]

cook
{"item":"hotdog",
"order_id":12,
"order":"ok",
"ing":"ok", "cook":"ok"}

Router-Dealer

identities:
[customer, order, ingredients, cook, prepare]

cook
{"item":"hotdog",
"order_id":12,
"order":"ok",
"ing":"ok", "cook":"ok"}




```
while True:  # router
    [sender_id, _, msg] = await router_socket.recv_multipart()
    sender_name = sender_id.decode() # cook
    receiver_name = next_step[sender_name] # prep
    await router_socket.send_multipart([receiver_name.encode(), b"", msg])
```

identities:
[customer, order, ingredients, cook, prepare]

prepare
{"item":"hotdog",
"order_id":12,
"order":"ok", "ing":"ok",
"cook":"ok"}

Router-Dealer

2. Ingredients

3. Cook

1. Order

4. Prepare

Router-Dealer

identities:
[customer, order, ingredients, cook, prepare]

prepare
{"item":"hotdog", "order_id":12, "order":"ok", "ing":"ok", "cook":"ok", "prep":"ok"}

2. Ingredients

3. Cook

1. Order

4. Prepare

identities:
[customer, order, ingredients, cook, prepare]

customer
{"item":"hotdog", "status":"done"}

Router-Dealer

identities:
[customer, order, ingredients, cook, prepare]

Router-Dealer

identities:
[customer, order, ingredients, cook, prepare]

Router-Dealer

Handles ...

- redundant & new stages
- missing ingredients & other issues

Can be solved by the router!

Router-Dealer

So …

- **Pros**
  - Scalable
  - Stateful
  - Universal
  - Fully async
- **Cons**
  - More complex to implement and manage
  - Weak coupling (identities)
  - Careful design

Protocols in ØMQ

- **tcp://**
  - Across network communications
- **udp://**
  - Across network communications, receiver side no guaranteed
- **ipc://**
  - Inter-process communications
- **inproc://**
  - Same process, multi-thread communication

More about ØMQ

- **Messages**
  - Binary objects
  - Any serializable object
- **Security**
  - Can be paired with CurveZMQ to provide encryption
- **Low level socket monitoring**
  - Validate various events occurring on the socket

Summary

- **Real world systems** often use combination of these pattern
  - Router-Dealer with Publisher-Subscriber for scalable and event driven systems
- Consider system **requirements**
  - Is scalability required?
  - Loose or tight coupling?
  - Message deliver guarantees?

Further Reading

Books

- *ZeroMQ: Messaging for Many Applications* (Pieter Hintjens)
- *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* (Hohpe Gregor and Woolf Bobby)
- *Building Microservices: Designing Fine-Grained Systems* (Sam Newman)

Web

- http://wiki.zeromq.org/

Examples from presentation

- https://github.com/xjules/edc2024-messaging

Further Reading

Books

- *ZeroMQ: Messaging for Many Ap*
- *Enterprise Integrati* *ying Messagi*
- *Building* *Grained Systems* (Sam Newman)

Web

- http://wiki.zeromq.org/

Examples from presentation

- https://github.com/xjules/edc2024-messaging

Thank you for your attention!