# IN4331 - Group 2

*A reactive solution using Celery/RabbitMQ*

CELERY
RabbitMQ
redis
mongoDB

Enrique Barba Roque, Philippe de Bekker, Arjan Hasami, Jokūbas de Kort, Simcha Vos

*...to serve and protect data.*

TU Delft

# Our System

# Databases & Logging

- For the main services: MongoDB
  - Easy to use, highly mature and popular.
  - Horizontal scaling through sharding.
  - Atomic, by checking on the database server.
  -
- Logging and recovery: Redis
  - Flexible data structures, highly mature and popular.
  - In-memory, so extremely fast.
  - Logs the steps in the saga.
  - Eventually scrapped

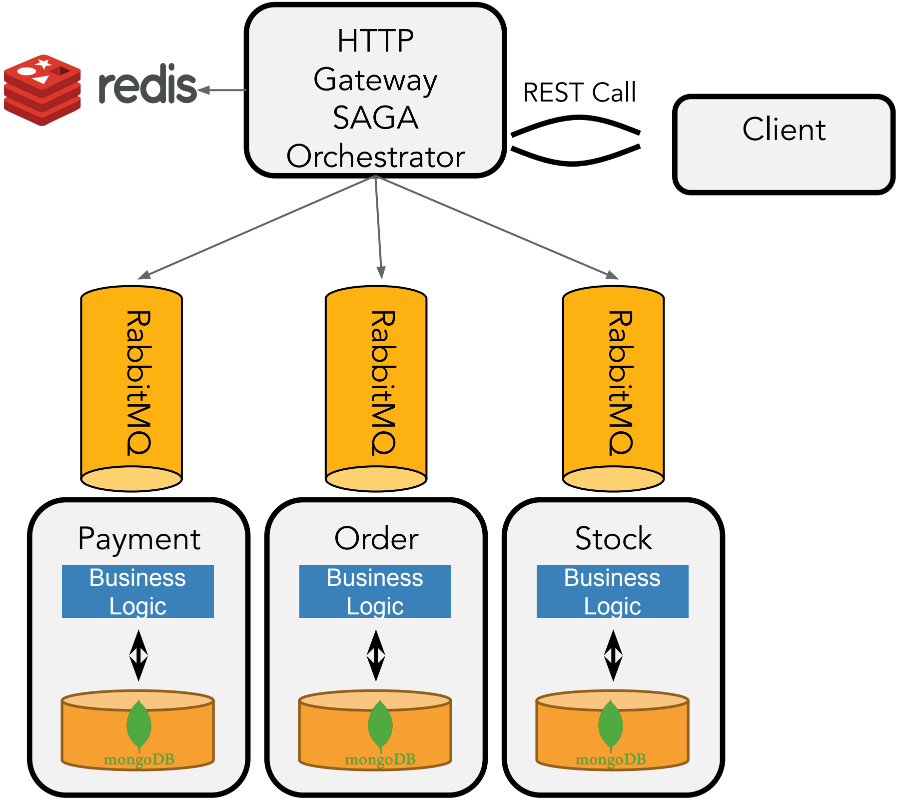
# Task Management & Messaging

- Distributed task queue: Celery
    - Allows distributed task processing, ideal for large amounts of concurrent tasks.
    - Highly mature and popular; robust and flexible.

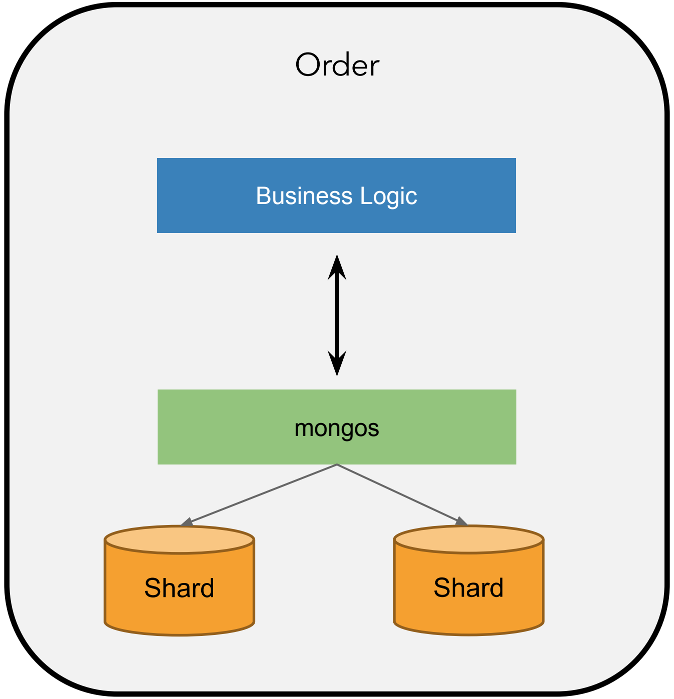
- Message broker: RabbitMQ
    - Implements Advanced Message Queuing Protocol; reliable message delivery. Highly mature and popular.
    - Celery workers grab from the queues.

# Our Solution

# Global overview

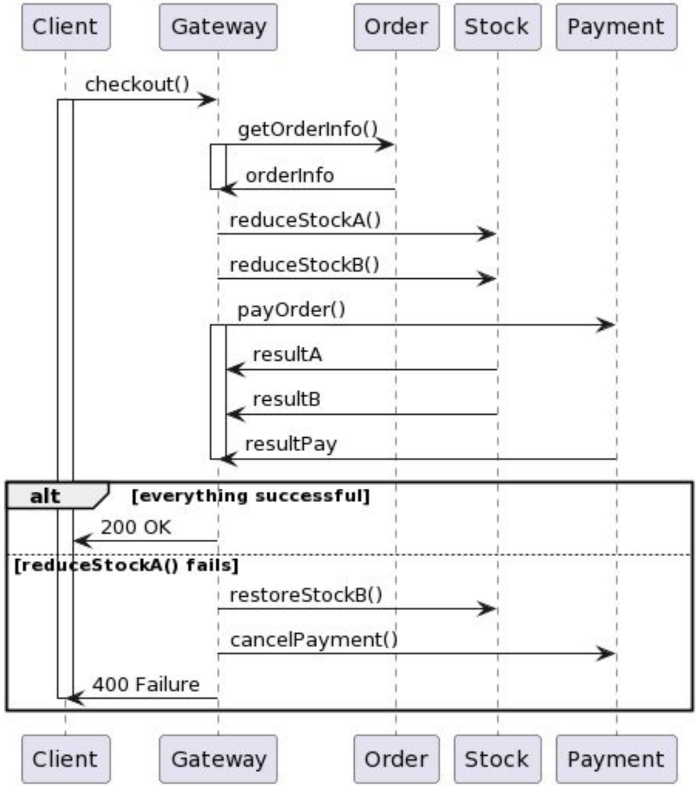


# MongoDB Sharding



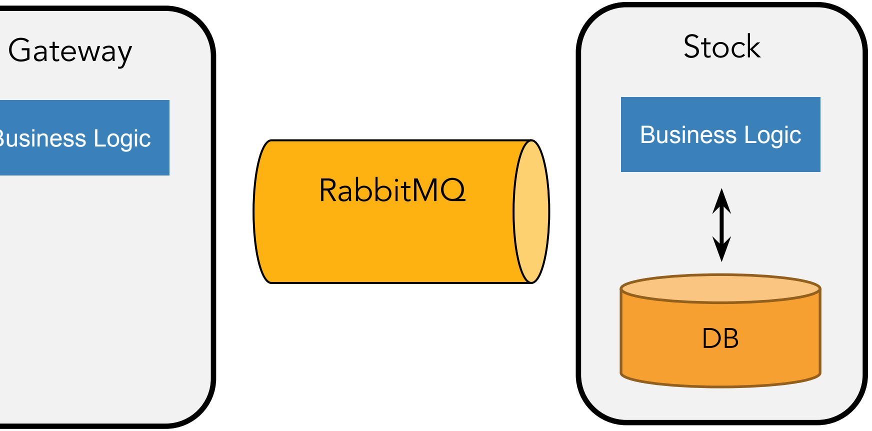

- MongoDB Sharding for DBscaling capabilities
- Operations by entity ID
- Partitioned by ID hashing
  - Equal distribution across shards

# SAGA



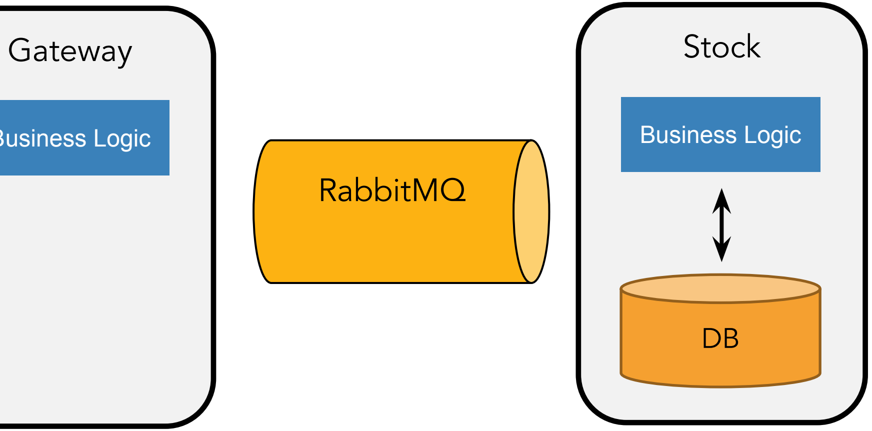

- Gateway acts as SAGA orchestrator.
  - Launch tasks to workers
  - Wait and track status
  - Compensate in case of failure

# RabbitMQ for internal communication




- RabbitMQ for communication between services
  1. Order publishes a message on RabbitMQ's message queue;
  2. Stock consumes messages on the queue and processes;
  3. Stock publishes response on queue;
  4. Order service consumes messages on the queue and continues processing the request.

# But what about REST?




- Classic microservices:
  - REST calls in the place of the queues makes all services synchronous
  - Result: horizontal scaling becomes less trivial due to synchronous exchanges
  - Services need to wait for a response and cannot handle other requests in the meantime

→ Asynchronous communication makes scaling specific services easy

# Celery

- If the API is different this would improve performance
- Celery used in code through `.delay()` and `.get()` functions

```
@router.get('/stock/find/{item_id}', status_code=status.HTTP_200_OK)
async def find_item(item_id: str):
    task = stock.find_item.delay(item_id)
    item = task.get()
    if item and not task.failed():
        return item
    else:
        raise HTTPException(status_code=404, detail="Item not found")
```

`.get()` blocks until it gets the actual result of the task

`.delay()` returns temporary object while queueing the execution of the task

# Load balancing @PHILIPPEEEEEEE

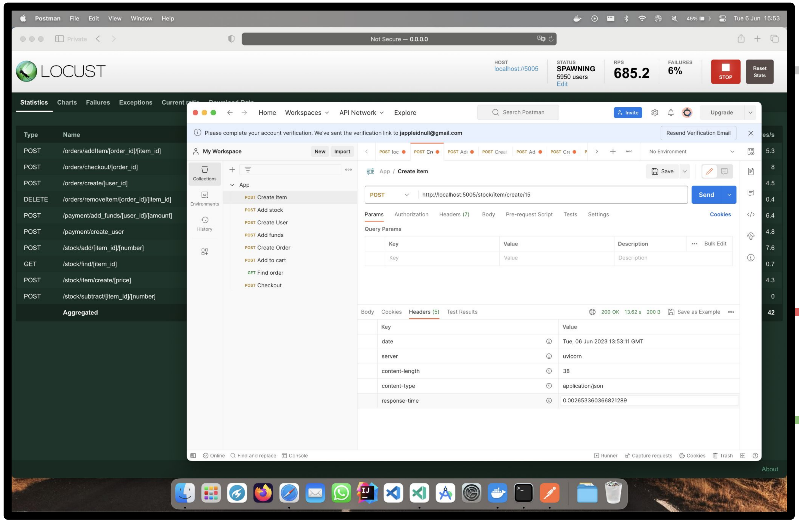
- Load balancing is used to

# Results

# Throughput


700 avg

# Latency



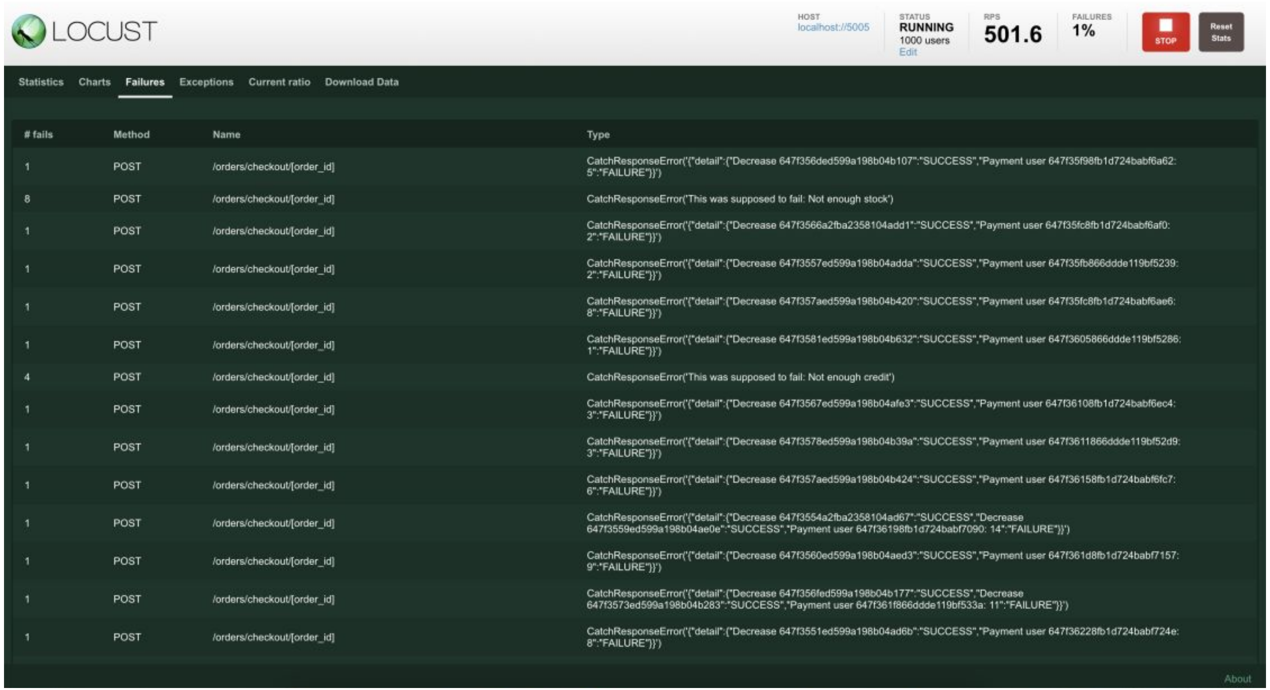

**POST Request to create stock item** ***(using Postman)***


**Number of users and requests per second**


**Bottleneck occurs in processing requests**


**Task completion is fast**

# Consistency




| # fails | Method | Name | Type |
|---|---|---|---|
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f356ded599a198b04b107":"SUCCESS","Payment user 647f35f98fb1d724babf6a62: 5":"FAILURE"}}') |
| 8 | POST | /orders/checkout/[order_id] | CatchResponseError('This was supposed to fail: Not enough stock') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f3566a2fba2358104add1":"SUCCESS","Payment user 647f35fc8fb1d724babf6af0: 2":"FAILURE"}}') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f3557ed599a198b04adda":"SUCCESS","Payment user 647f35fb866ddde119bf5239: 2":"FAILURE"}}') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f357aed599a198b04b420":"SUCCESS","Payment user 647f35fc8fb1d724babf6ae6: 8":"FAILURE"}}') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f3581ed599a198b04b632":"SUCCESS","Payment user 647f3605866ddde119bf5286: 1":"FAILURE"}}') |
| 4 | POST | /orders/checkout/[order_id] | CatchResponseError('This was supposed to fail: Not enough credit') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f3567ed599a198b04afe3":"SUCCESS","Payment user 647f36108fb1d724babf6ec4: 3":"FAILURE"}}') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f3578ed599a198b04b39a":"SUCCESS","Payment user 647f3611866ddde119bf52d9: 3":"FAILURE"}}') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f357aed599a198b04b424":"SUCCESS","Payment user 647f36158fb1d724babf6fc7: 6":"FAILURE"}}') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f3554a2fba2358104ad67":"SUCCESS","Decrease 647f3559ed599a198b04ae0e":"SUCCESS","Payment user 647f36198fb1d724babf7090: 14":"FAILURE"}}') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f3560ed599a198b04aed3":"SUCCESS","Payment user 647f361d8fb1d724babf7157: 9":"FAILURE"}}') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f356fed599a198b04b177":"SUCCESS","Decrease 647f3573ed599a198b04b283":"SUCCESS","Payment user 647f361f866ddde119bf533a: 11":"FAILURE"}}') |
| 1 | POST | /orders/checkout/[order_id] | CatchResponseError('{"detail":{"Decrease 647f3551ed599a198b04ad6b":"SUCCESS","Payment user 647f36228fb1d724babf724e: 8":"FAILURE"}}') |

**Chain of failures due to performance bottleneck, but still consistent**

# What would you do better?

*What would you have done better if you had two more months of time.*

# What would you do better?

- Try different frameworks (e.g. Spring WebFlux)
- Make the API calls asynchronous
- Amazon Web Services
- Crash Recovery

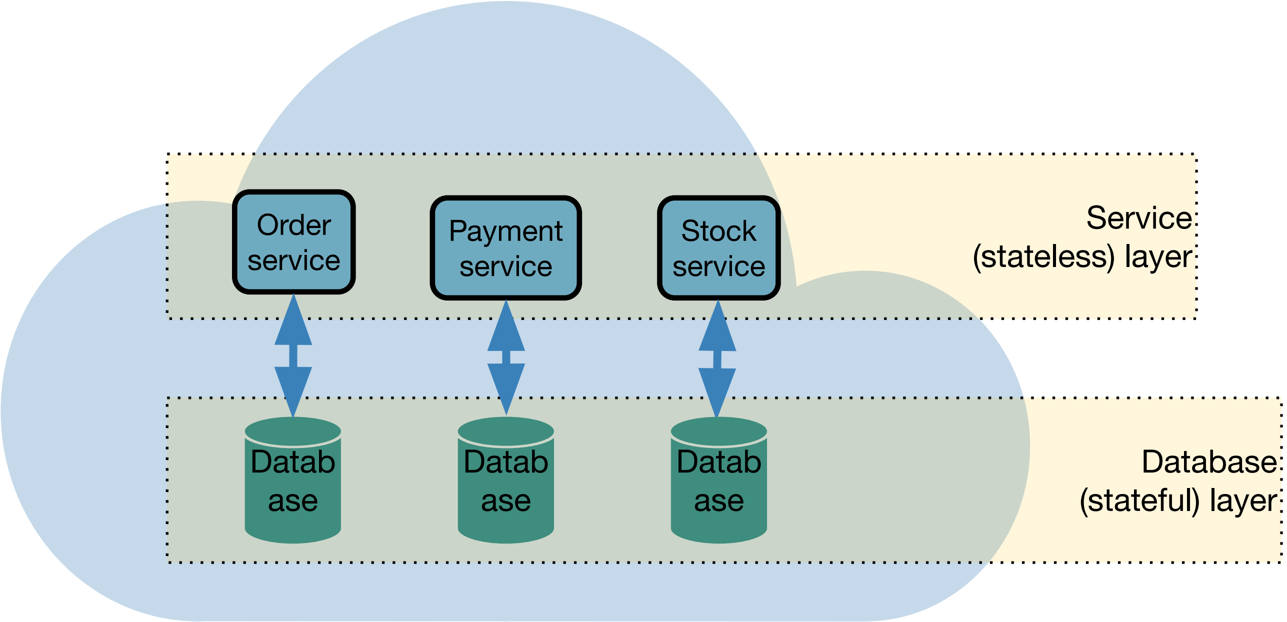
# Questions?

# Asynchronous API maximizes the potential of our architecture

- **Some slides that may be useful to you, in order to draw stuff.**

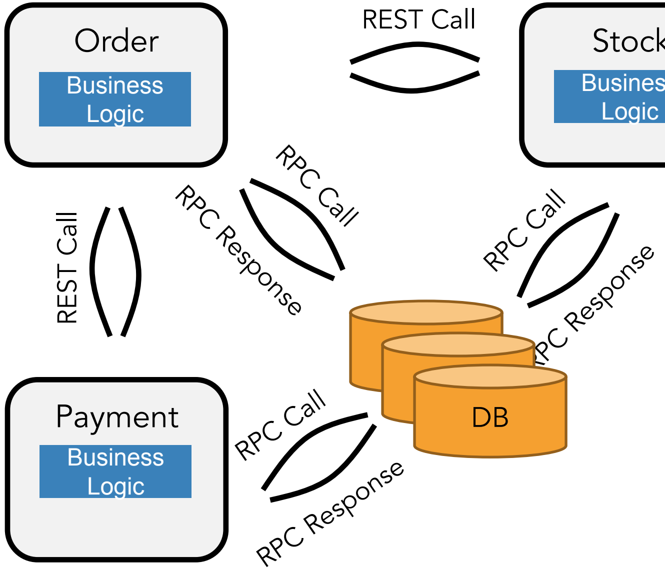
# LECTURE SLIDES NOT IN ACTUAL PRESENTATION

# A tale of three Cloud services




To checkout: stock & update stock, verify payment, checkout the cart. Atomically!

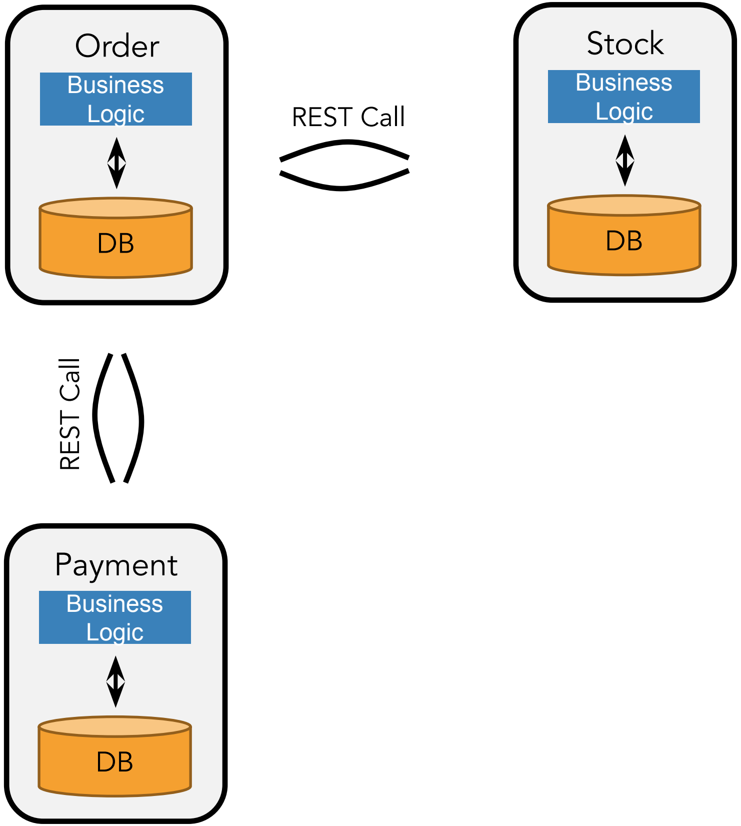
# Services Architecture (1): Easiest Implem.




- Perform an order iff there is stock available and the payment is cleared.

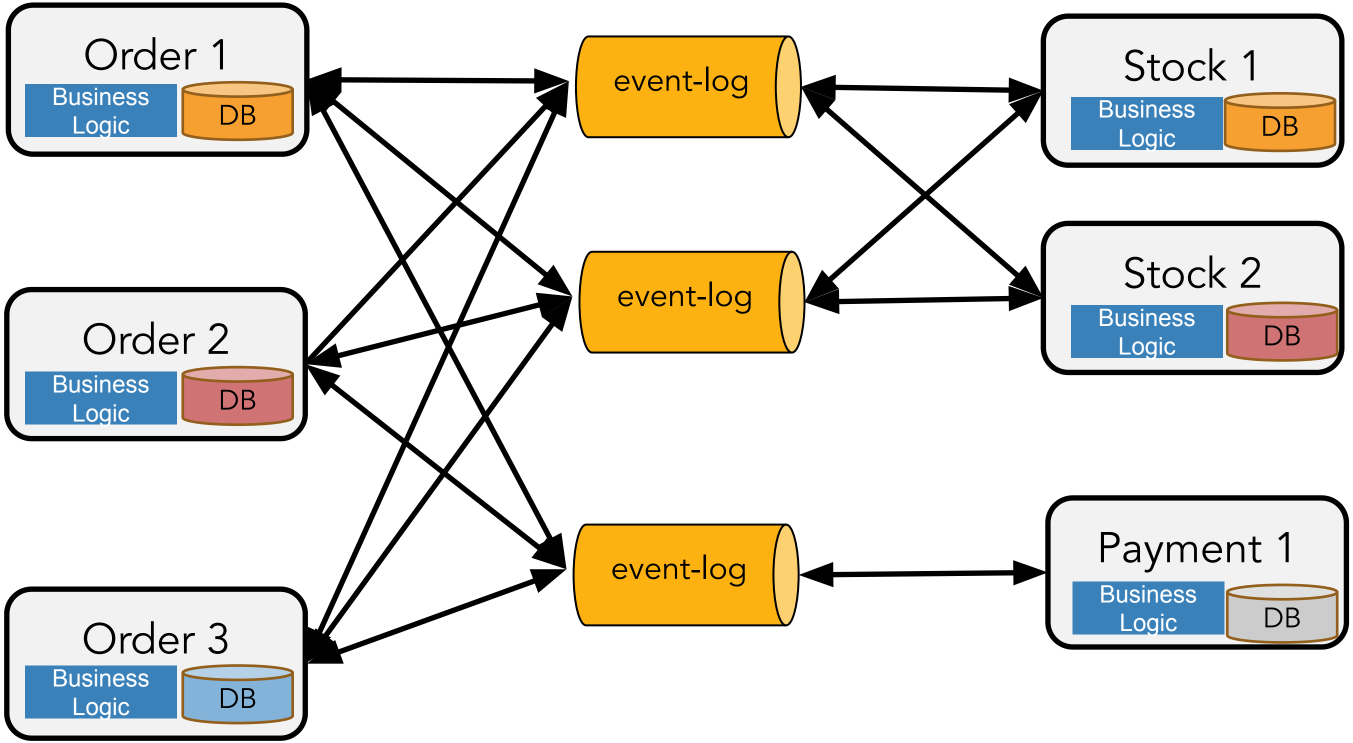
- Services are *stateless*
- Database does the heavy-lifting
- High latency, costly state access
- No guaranteed messaging

# Services Architecture (2): Embedded State/DB

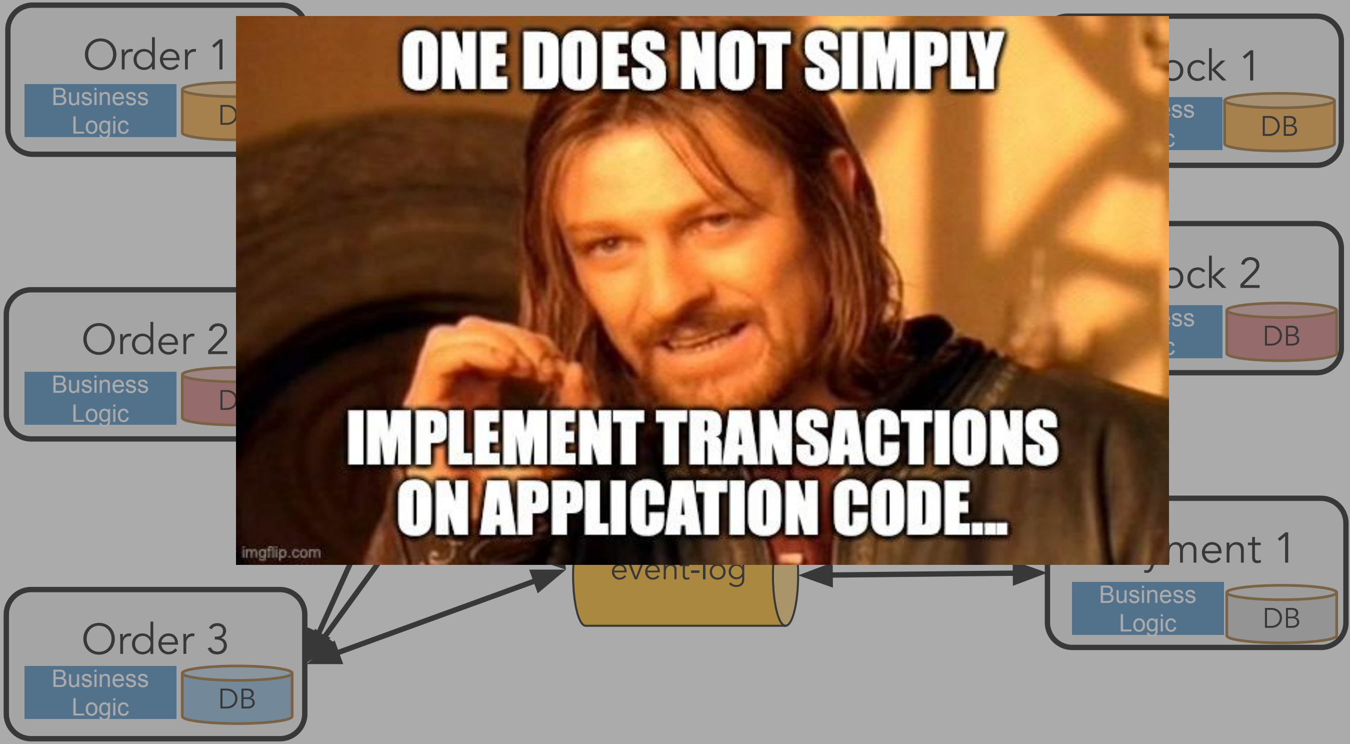



- Low-latency access to local state
- Service calls still expensive
- Messaging still not guaranteed
- Not obvious how to scale this out
- **Fault tolerance is hard!**

# Services Architecture (4): Scalable Deployment



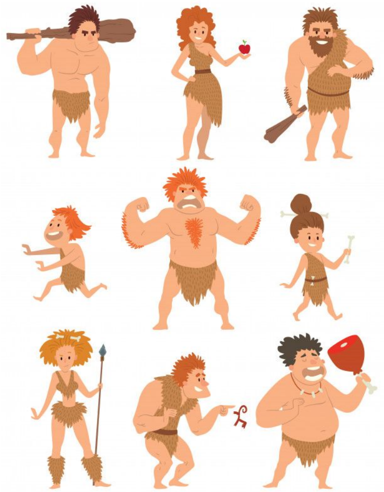
# Services Architecture (4): Scalable Deployment



# TL;DR

*We live in the stone ages.*

*Building scalable Cloud applications is like programming assembly before compilers were around.*



“Two-pizza” dev team in the year 2021.

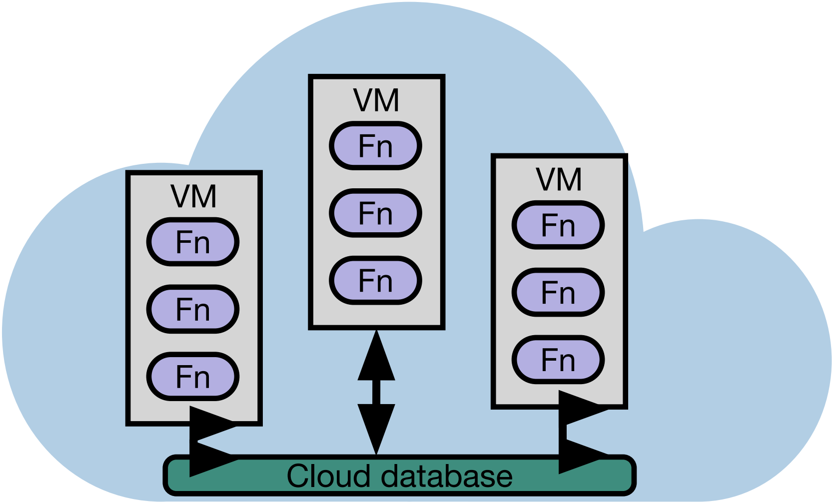
# Wait, what about serverless? That should work!

AWS Lambda

Google Cloud Functions

Managed Infrastructure (autoscaling, no ops) ✓

Function-based programming model ✓




✗ No State

✗ Fn-to-fn calls

✗ Transactions