curr. service

```
//...
//...
val cred: List[String] = authService.getCredentials(user)
//...
//...
```

A

B

we have to wait

```
val x = 2 + 2
```

we don't need

DOESN'T DEPEND

```
if (cred.contains("admin")) ???
```

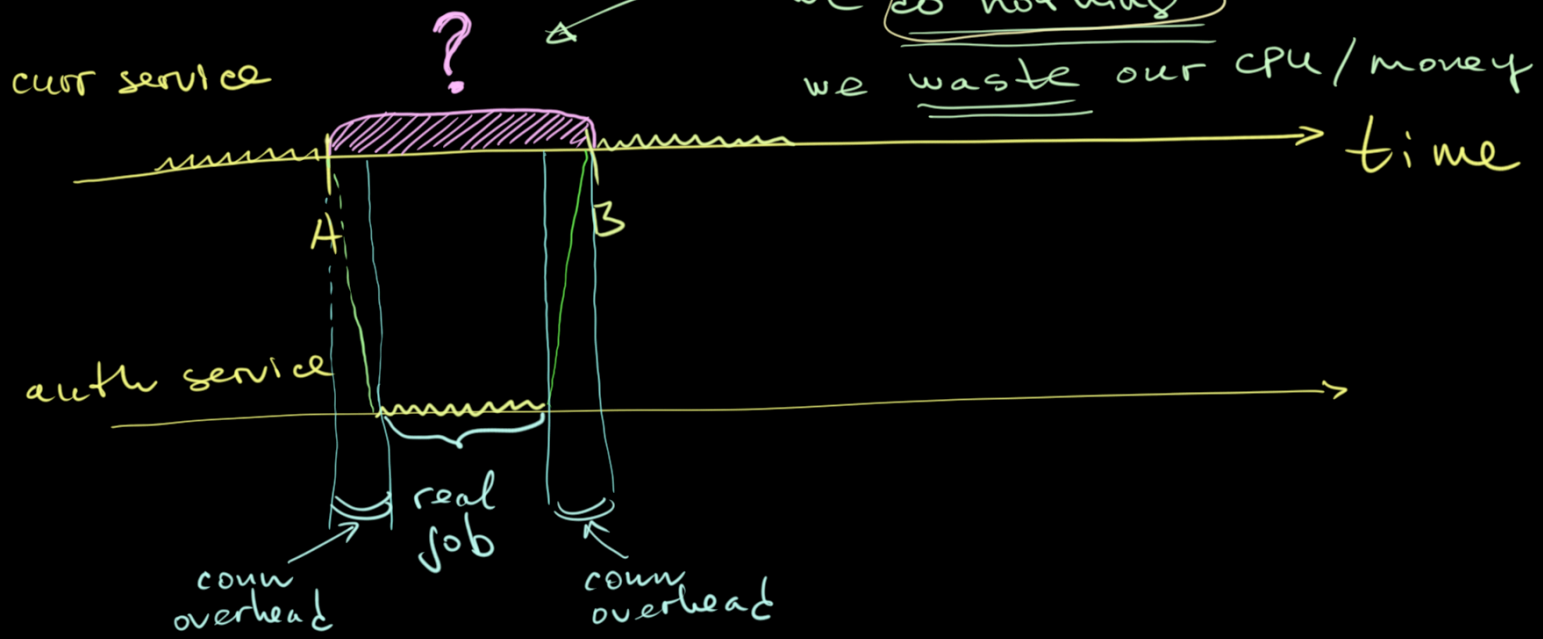
we need to wait

we can't optimize this block

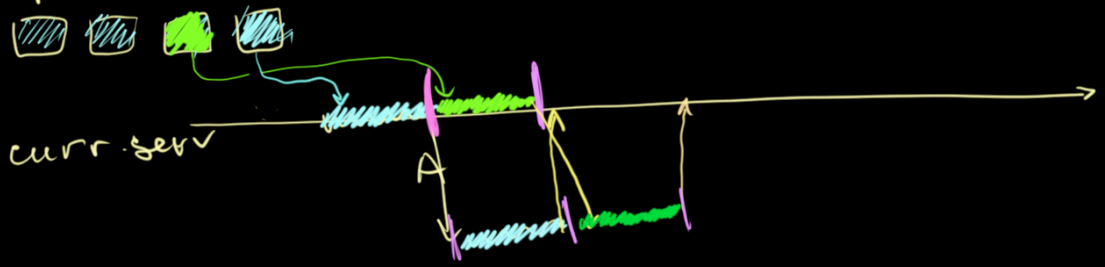
curr. serv. is waiting

we do nothing

we waste our CPU / money



queue of requests

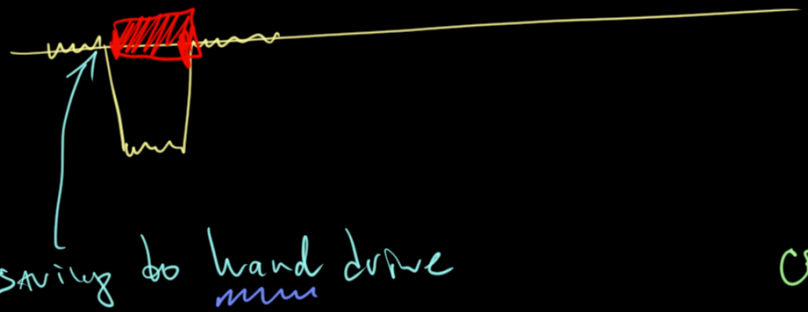


when we have many requests

even in situation we "can't do" anything

we eliminate moments/chunk of time we do nothing

It's hard to switch the context, threads, pick the next item from queue ...



we are into resource utilization.

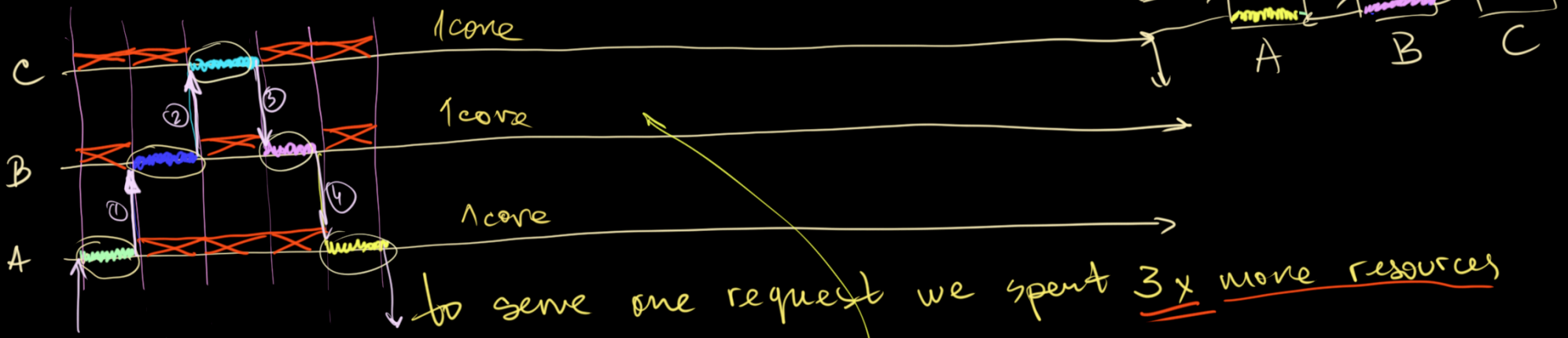
everything is out of the CPU is the waste of resources

10x 10x 10-100x

CPU < MEMORY < LOCAL HDD < REMOTE

1-10 ns 10..100 ns 1..10 ms 10..500 ms

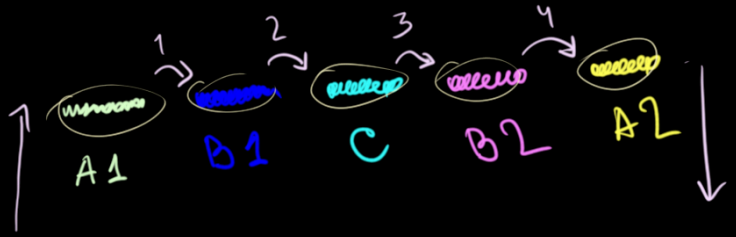
the main concept to fix that → Messaging

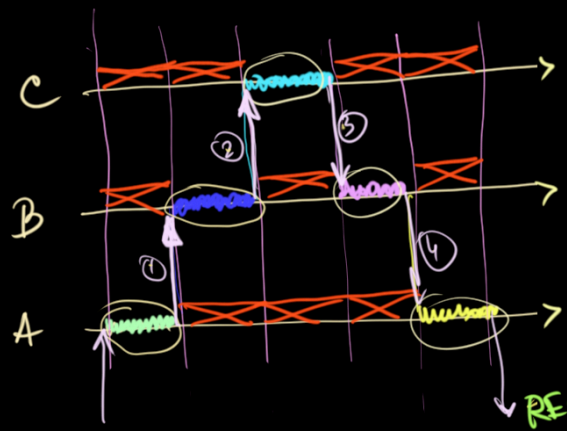


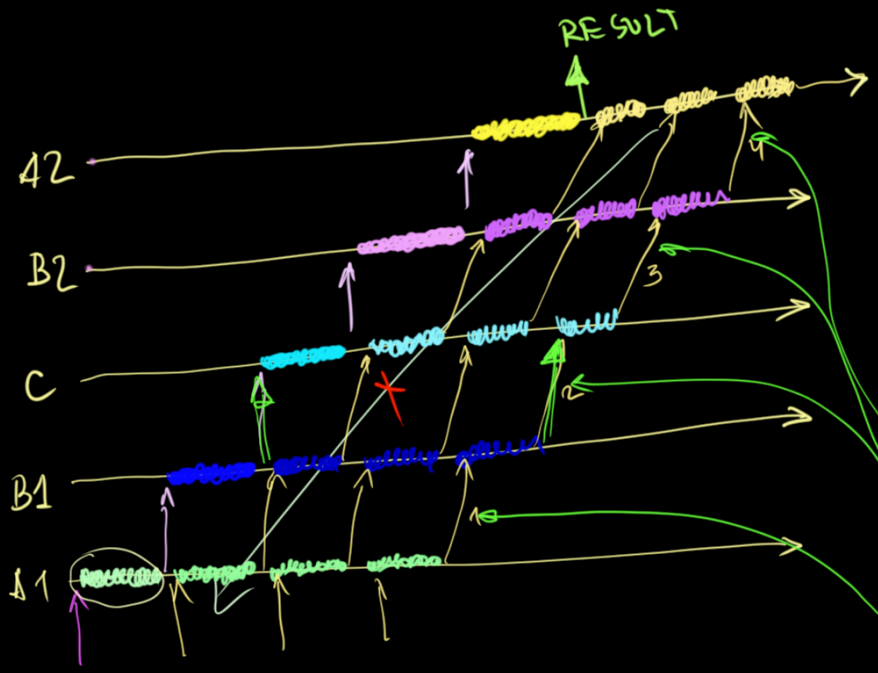
to serve one request we spent 3x more resources

1. we can develop them independently
2. we pay too much for that.

100% match







① do we need to go back? NO!

② do we care/need to know what's going on after A1 is done NO!

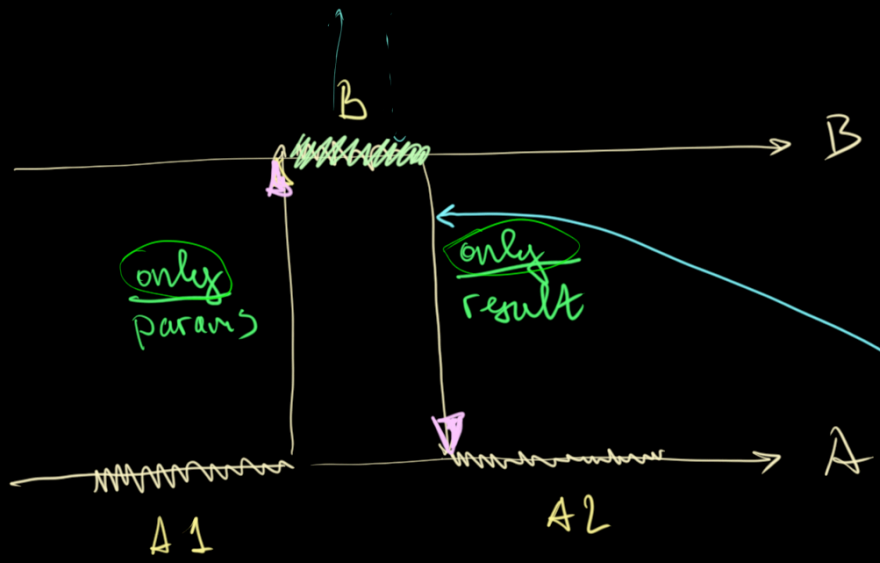
③ do we need to wait? NO!

④ do we waste the resources? NO!

messages, not the calls

3x Better (resources)!

Where is the complexity if everything is so cool ?

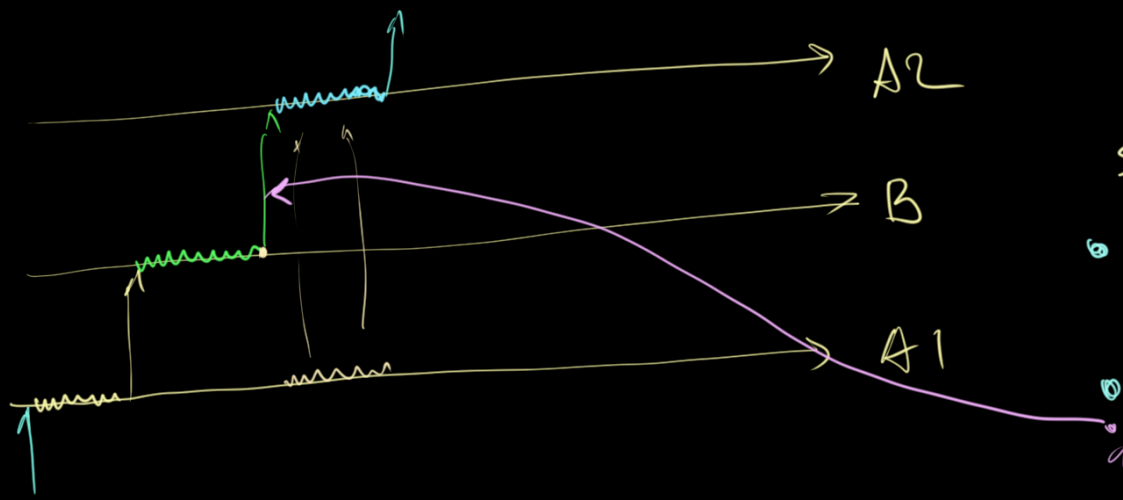


should ms B be aware of A1 → NO

should ms B be aware of A2 → NO

A2 depends on **result of B**

just a return to originating call

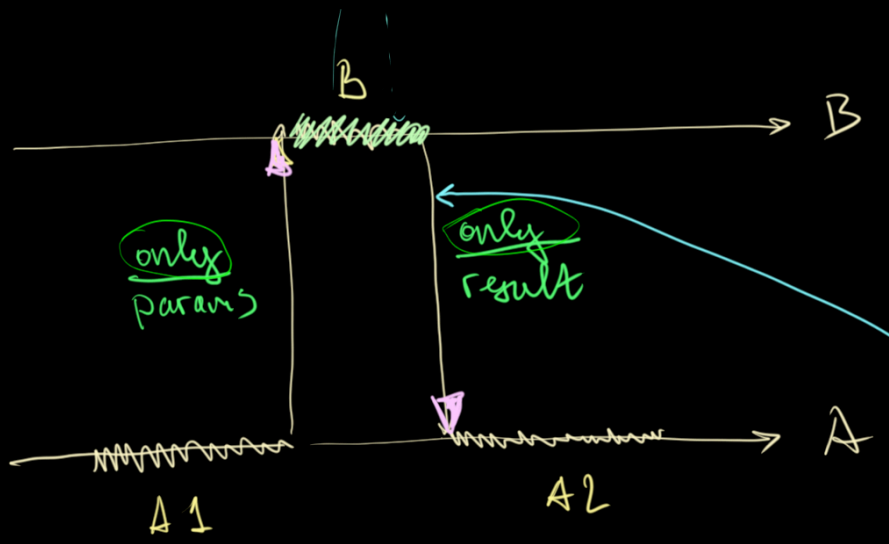


should ms B be aware of A1 → NO

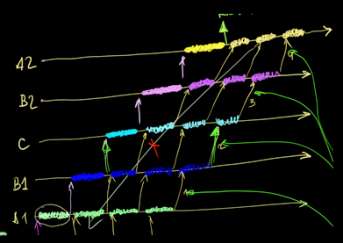
should ms B be aware of A2 → YES

- because A2 contains some business logic
- A2 goes AFTER B

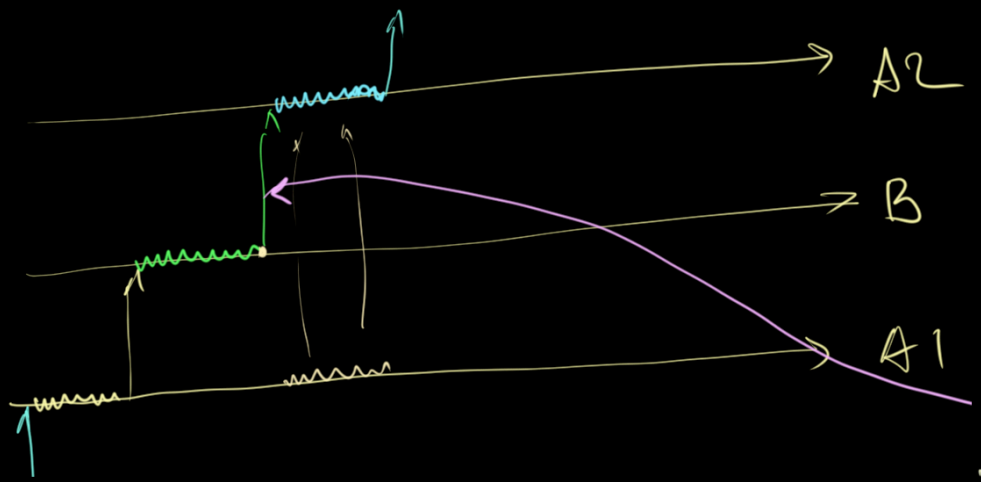
this is NEW CALL must contain ALL future business logic



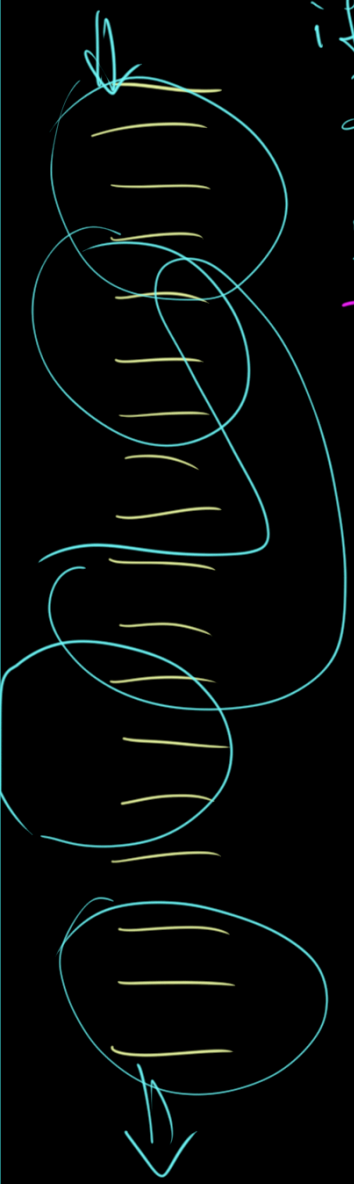
the SIMPLICITY of
is a complete redesign
of our Application
complete redesign - BIG PRICE



Just a result, I don't care
do what you want
here we return control to originator

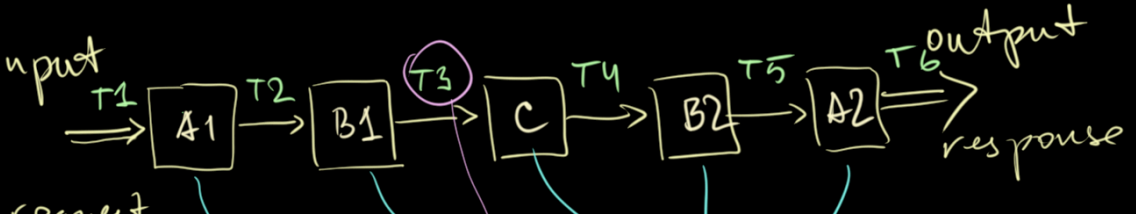


not only result,
but full awareness of further computation
here we don't return control to originator



if you try to redesign you will fail

=> we need to design our system in such way from the scratch



any piece of code

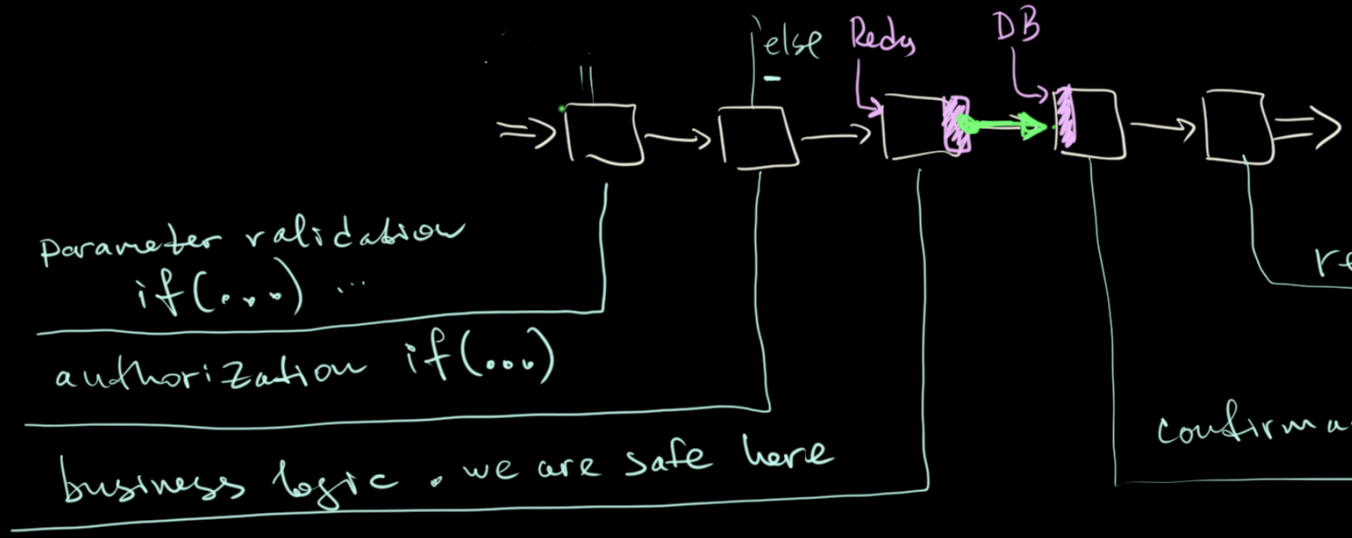
f: A => B

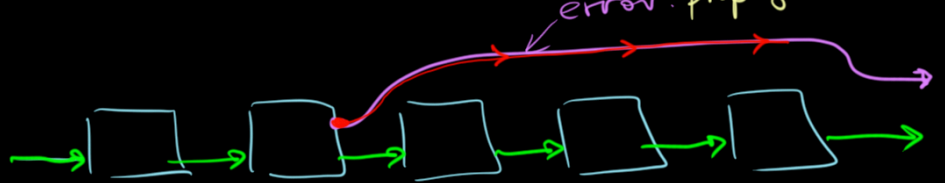
T1 ≅ input

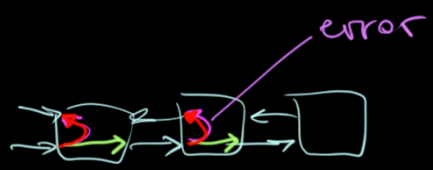
all of these transformations / steps are independent

A1: T1 => T2
B1: T2 => T3
C: T3 => T4
B2: T4 => T5
A2: T5 => T6

types always match







microservice

the difference in a structure . we need to have APP well structured

and represent all steps

as a function: A => B

(A, DB, Redis) => (Credit, Rq, User)

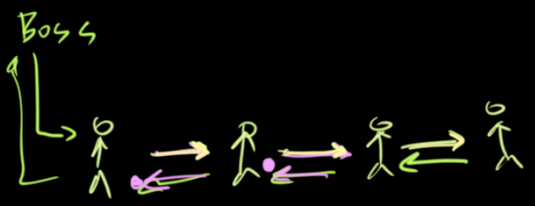
A1: T1 => T2
B1: T2 => T3
C: T3 => T4
B2: T4 => T5
A2: T5 => T6

Microservice design

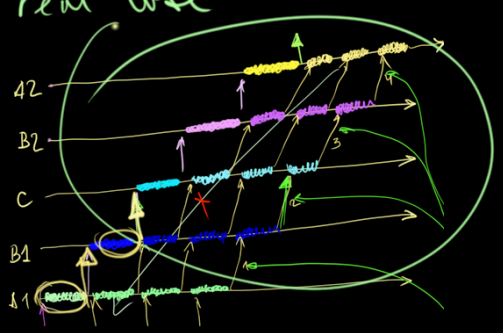
```
// ...
val cred: List[String] = authService.getCredentials(user)
// ...
```

we always have a result. we must wait

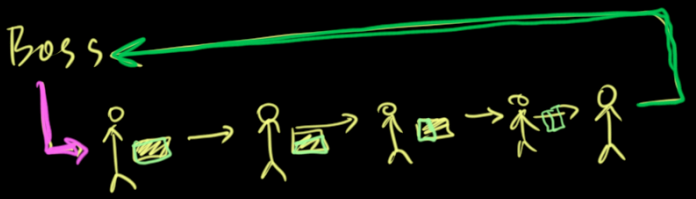
the code is running sequentially line-by-line



real life



- do we have result after the call NO!
  we just send the message to the next guy / service

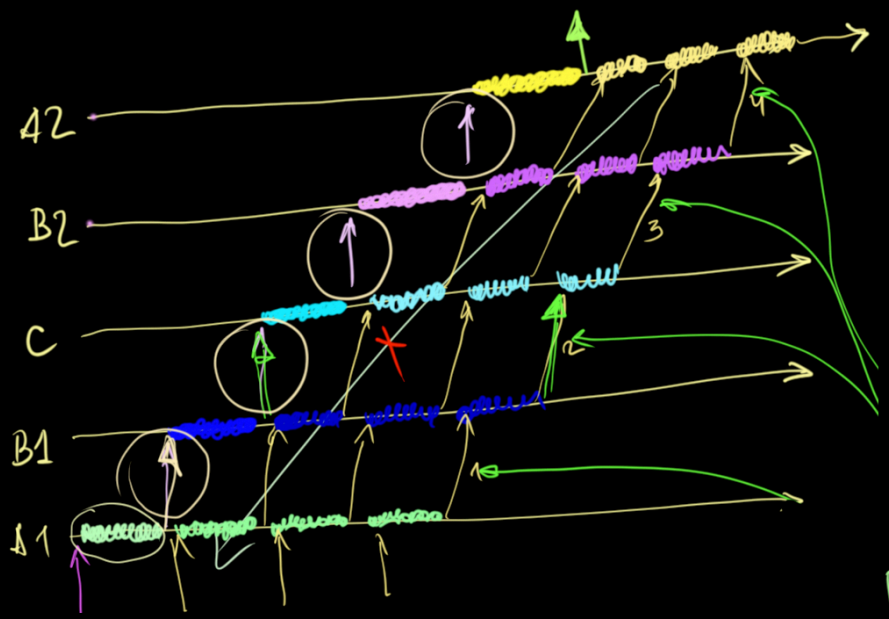


- I never have a result
- I don't know about all the further steps
- Do I know if something goes wrong? → NO! ↘ good? bad?

```
// ...
authService.doNext(user)
// ...
```

passing the message

We need to establish "protocol" to know about the probl



sending the message
is a stupid thing can be put to library

- we always need to know about the next step
- we finished
  we need to know whom to pass
- it creates a coupling

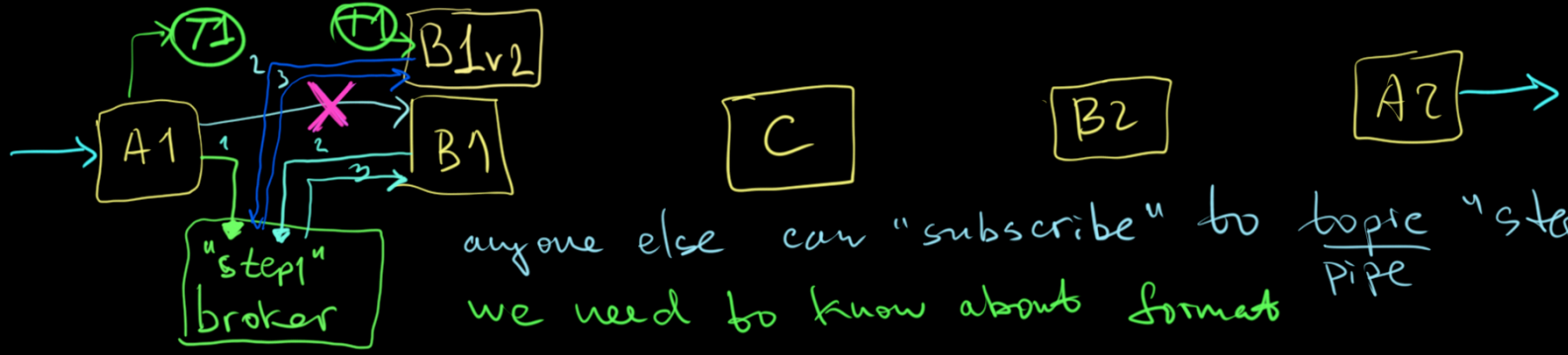
problem

no way to change a consumer.

Broker

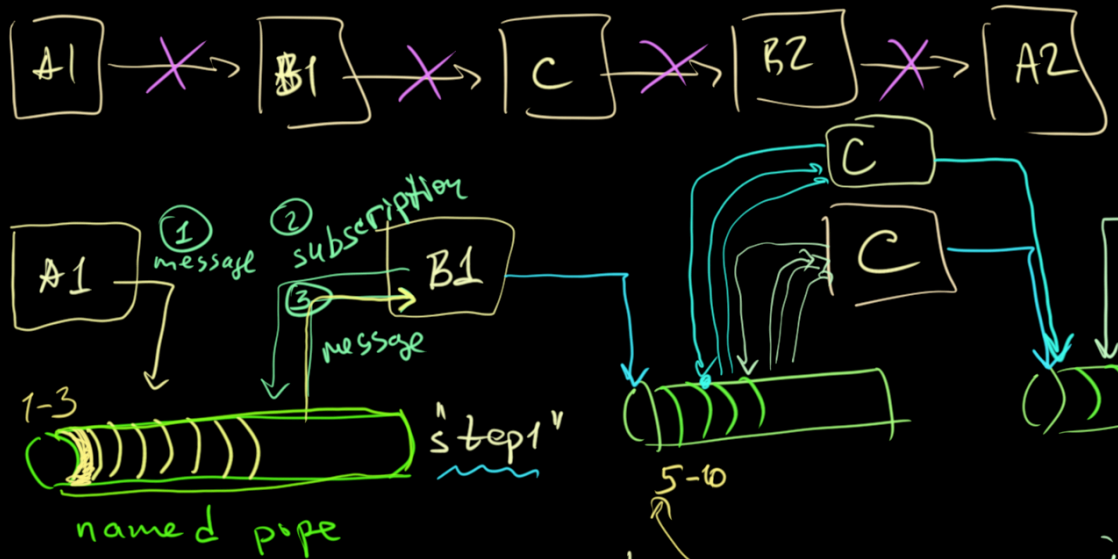
by introducing broker we can eliminate coupling

↳ we don't know whom to we send
we send to broker and give a name to queue



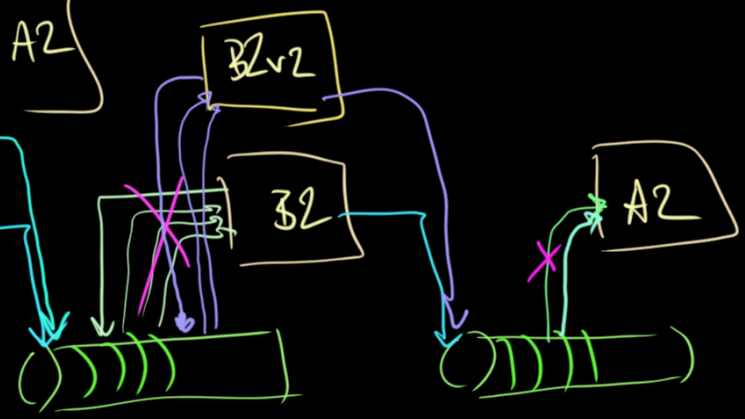
anyone else can "subscribe" to topic "step1"
pipe

we need to know about format



- buffering out of the box
- kept "events" / "message" will be sent once B1 is ready

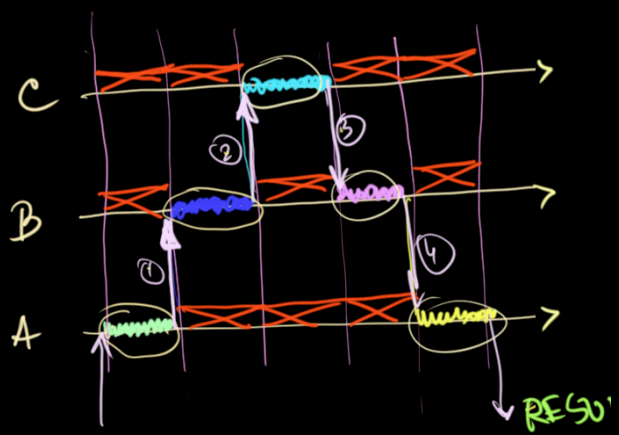
- we can analyze the queue size in the queue and always growing service C isn't able to process it we can spawn more instances and make process parallel

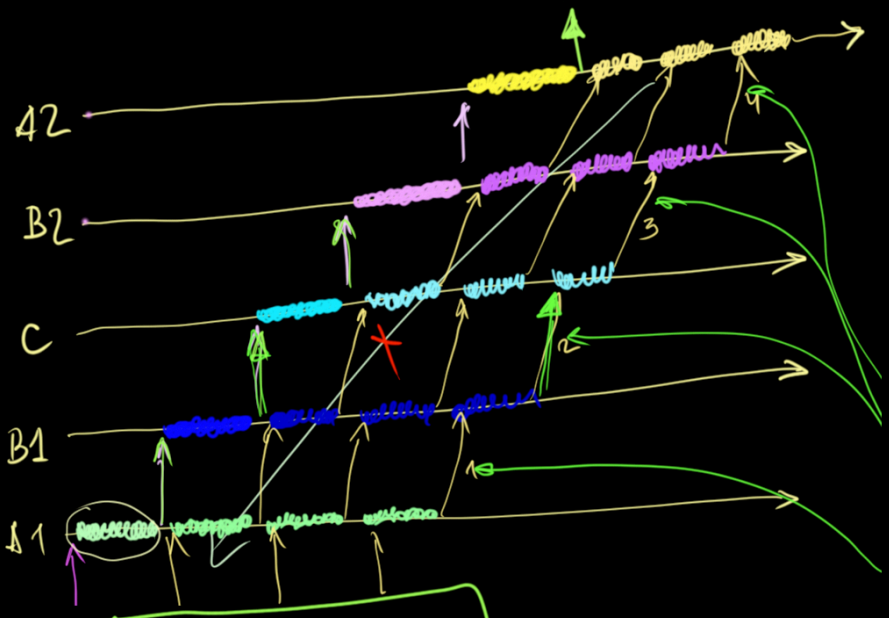


- instead of sending directly A1 → B1 we send message to a broker w/ topic name
- anybody can "subscribe" to a topic

Nowadays Messaging Systems are Every Where

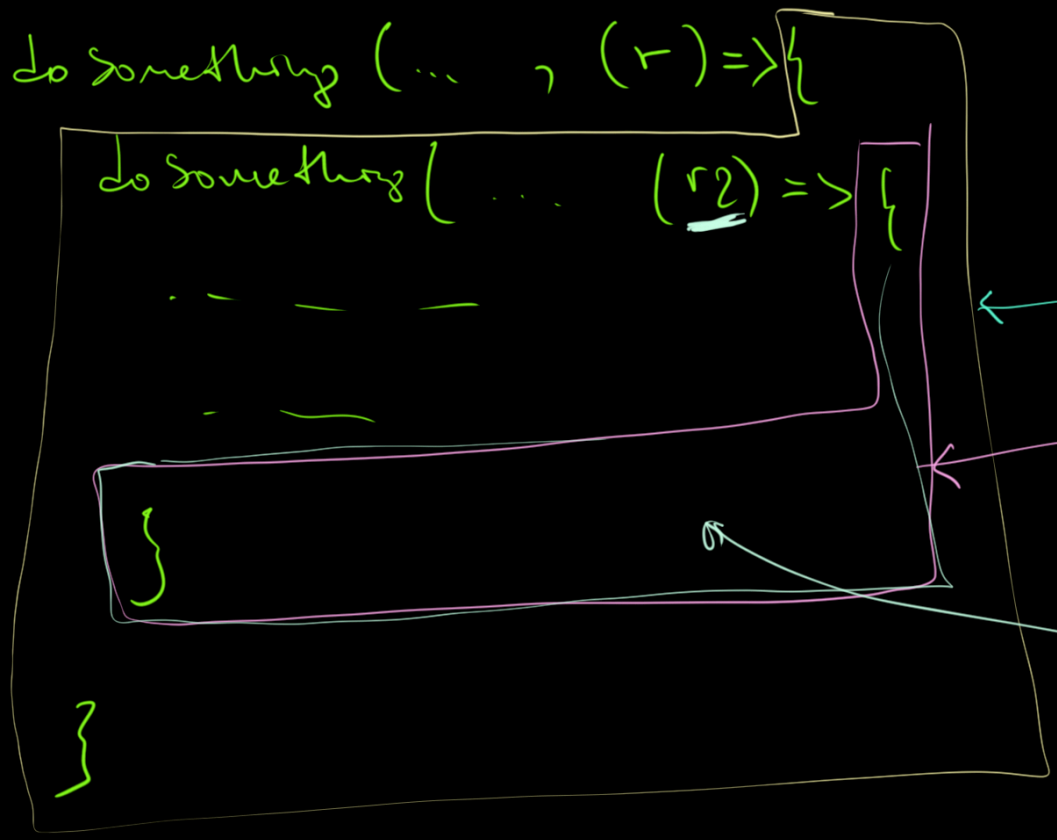
KAFKA
Rabbit MQ
IBM EventBus

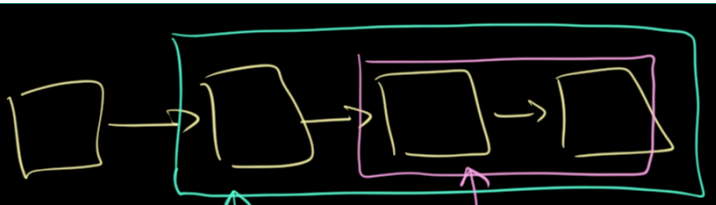




IS FIXED

the main problem of wasting resources
we need to design our system wisely, carefully
and need to use messaging systems
→ buffering
→ retries
→ multiple consuming / parallism } out of the BOX





it will be run, once
- r2 is ready
- V8 engine has a free window to run it

last 15-20 years we solve resource wasting problem

16/32/64 core

Kubernetes allows as to set fixed amount

of MEM / CPU for each POD (JVM process)

w/ proper setup on hardware with 16 cores

we can run 16 JVM process w/o problems

```
kafkaProducer.produceOneInto(topic)(kafkaKey, kafkaValue)
```

payload

```
KafkaConsumer
  .stream(consumerSettings)
  .evalTap(_.subscribeTo(aviLastUsedTopicName))
  .flatMap(_.stream)
  .groupWithin(250, 1.minutes)
  .metered(1.second)
  .mapAsync( maxConcurrent = 10)(processChunk)
```

```
val bulk = chunk.map { ccr =>
  val kValue        = ccr.record.value
  val shownAtString = OffsetDateTime
    .ofInstant(kValue.meta.createdAt, ZoneOffset.UTC)
    .truncatedTo(ChronoUnit.MILLIS)
    .format(ISO_OFFSET_DATE_TIME)

  UpdateUserRequest(
    kValue.data.email.value,
    IterablePayload(shownAtString),
  )
}.toList
```