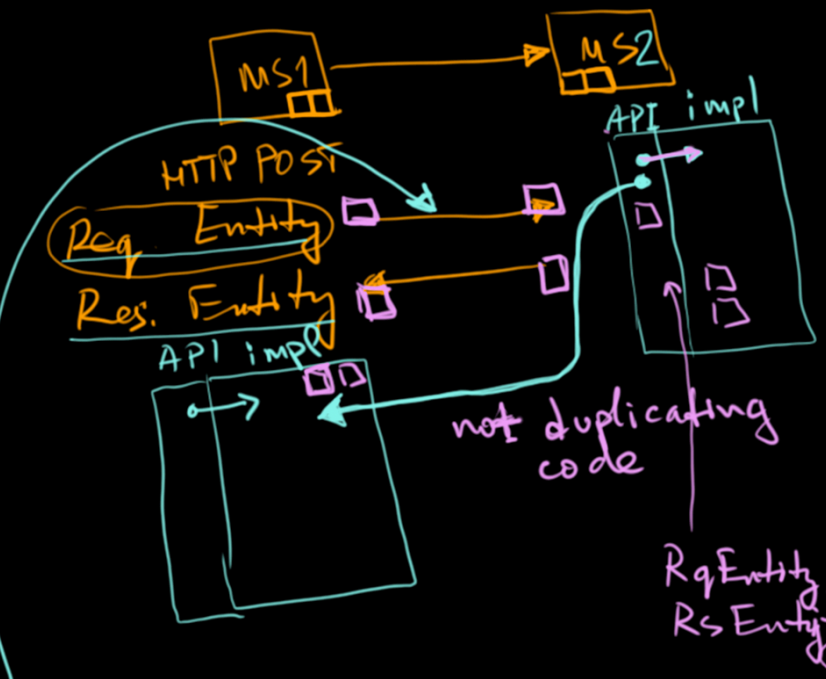
Microservices

Pros

1. better encapsulation
   separation of concerns
2. responsability
3. scalability
4. maintainability for big teams

cons

→ share some code between MSs



ms2 -api
ms2 -impl

ms1 -api
ms2 -impl

~~orders -common~~
~~payments -common~~

ms1
new RqEntity
entity.asJson
attach to http body

HTTP → decode body → RsEntity

1. sharing common code

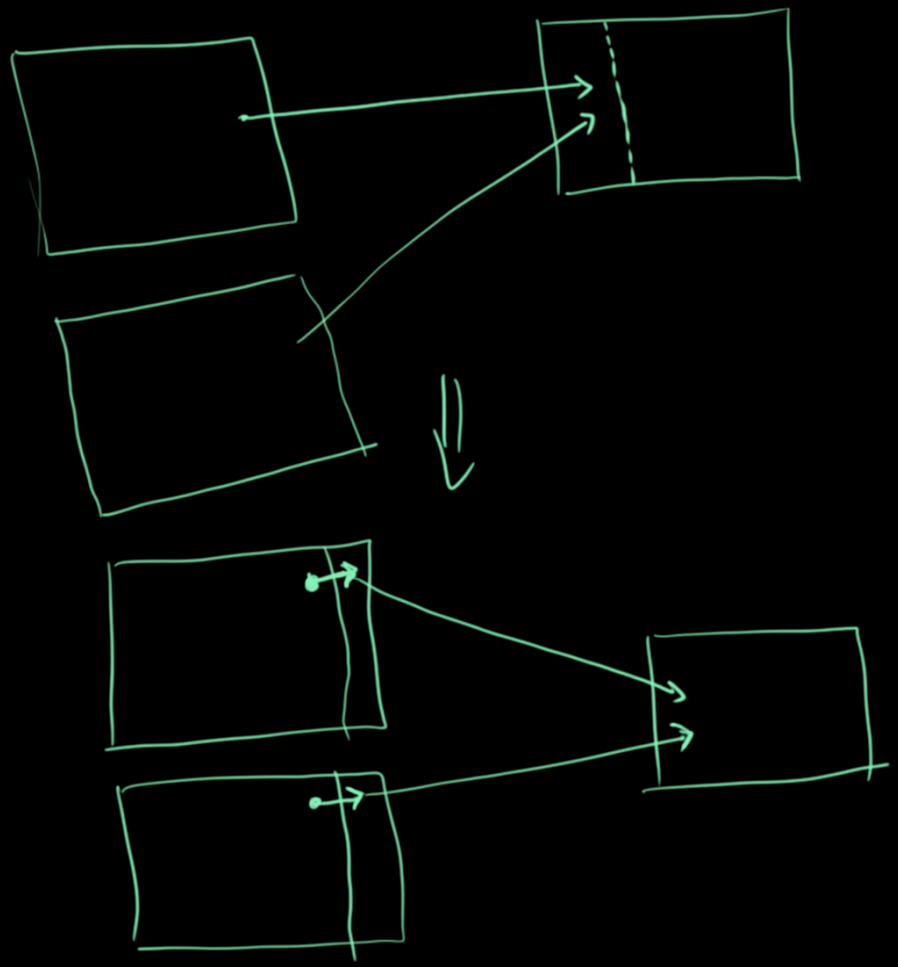
ms → ms-api
ms → ms-impl

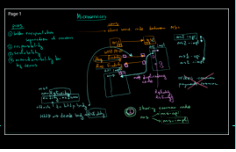
Page 2

Scope of changes

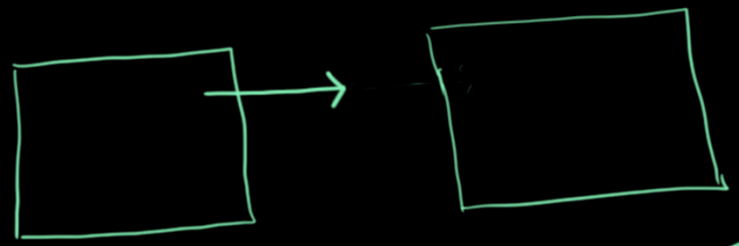
refactoring isn't simple easy

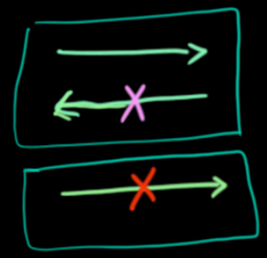
- more time
- more developers
- more mistakes
- more money spent





Failures





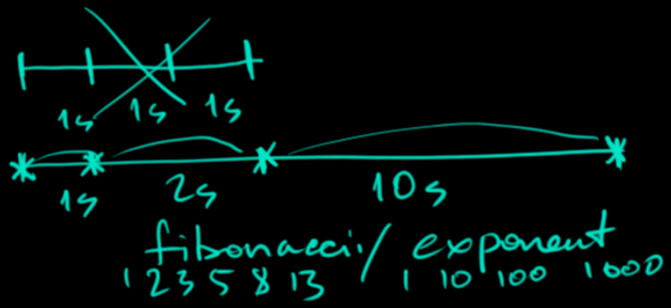
you never know

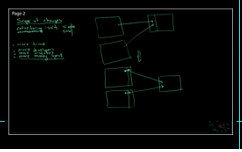
1. we didn't get response

MkPostRequest failed due to timeout

2. what is the proper timeout ? → you never know

1.1 how to handle absence of response
- whether to retry ? → timeout / retry policy
- response with the failure ?
  - limit retries
  - increment waiting time





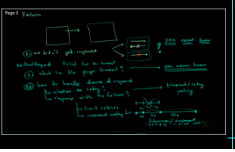
Special Data Design

Idempotent

any duplicate request can't change the state

	idempt	not
creating user	✓	✗
deleting user	✓	✓
updating user	✓	✗
finding/selecting	✓	✗

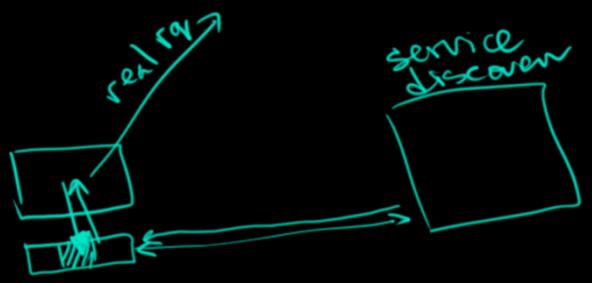
upd name → 1 → 1
another upd name → 1 → 2
2 → 1
2



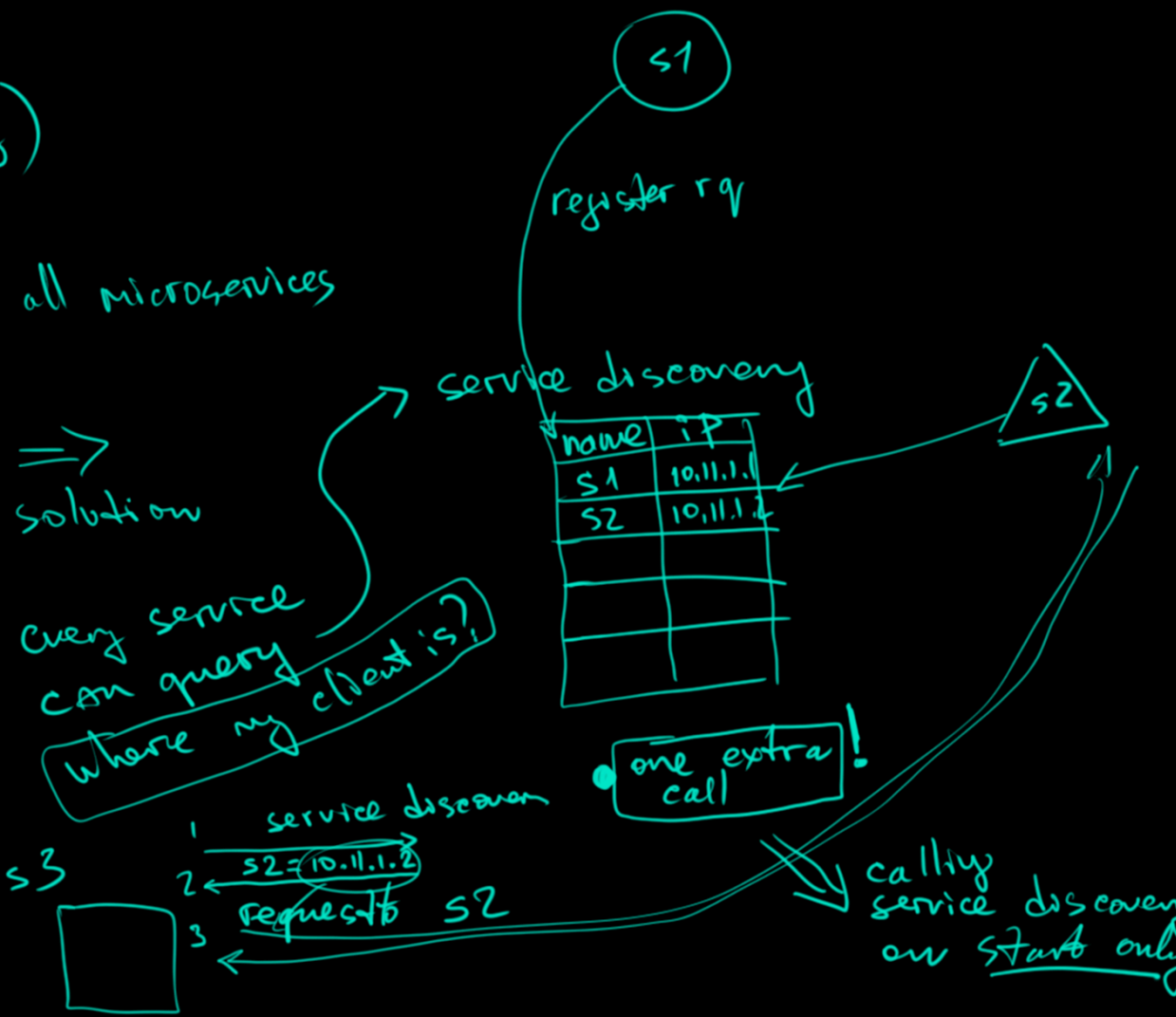
Another service location

http request (host, body)
IP

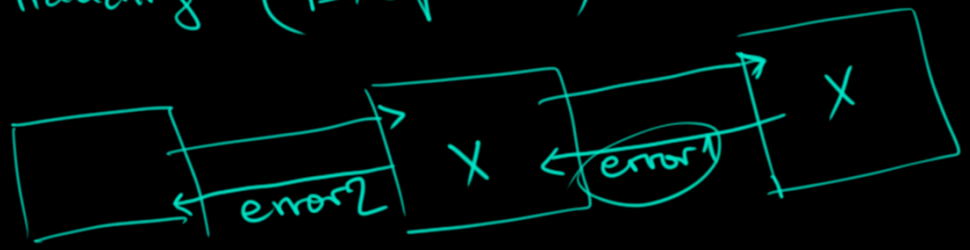
⇒ a lot of configuration for all microservices



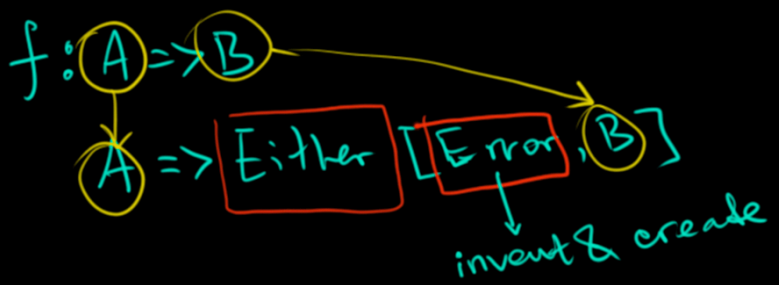
⇒ solution

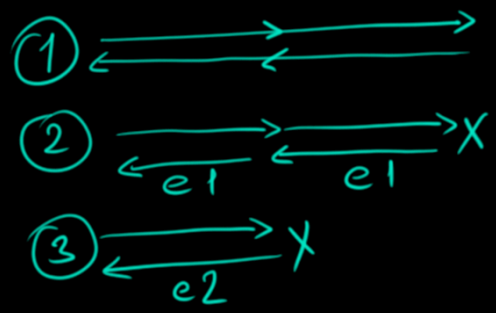


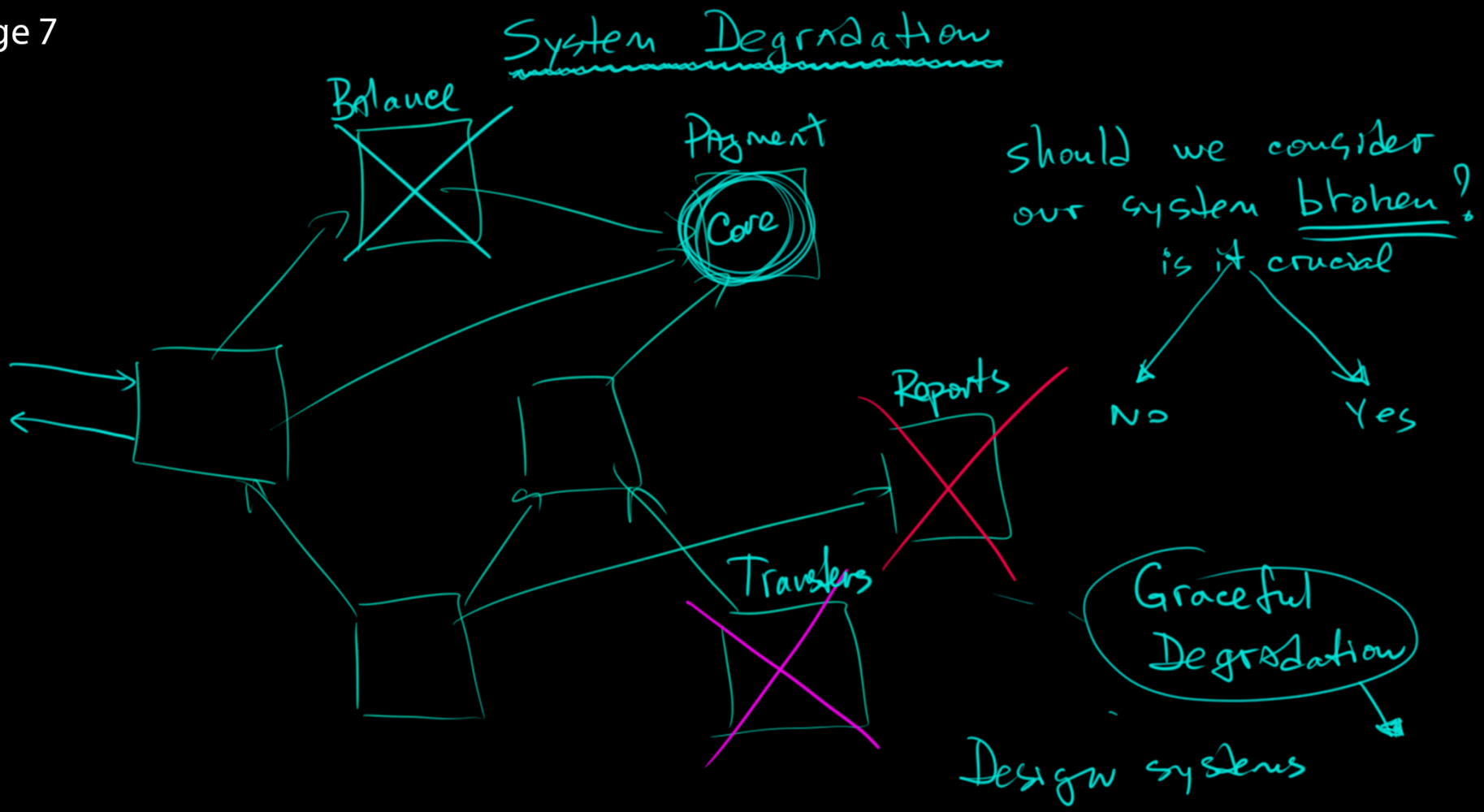
Errors handling (Exceptions)



int i = i + 1






System Degradation
Balancer
Payment
Core
Reports
Travelers
should we consider our system broken? is it crucial
No
Yes
Graceful Degradation
Design systems

Microservice Architecture

takes **cons**

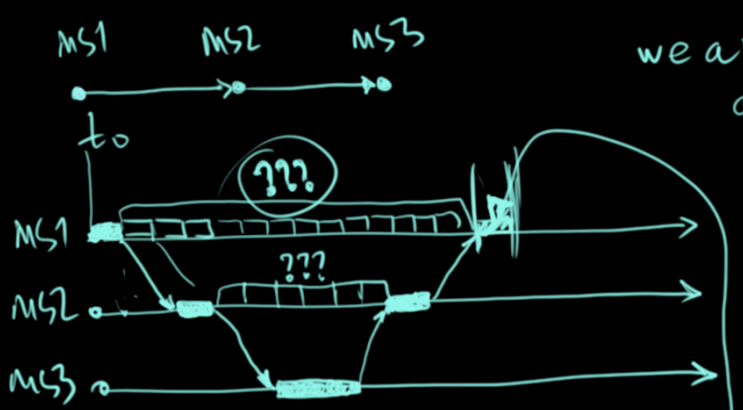
- 10x time
- 5x developers
- 10x money
- 10x complications

pros

- scalability → 1000000
- maintainability for big teams

Microservice ⟷ Monolith

Monolith → Structured Monolith → migrating to Microservices ONE by ONE



we are waiting and wasting time

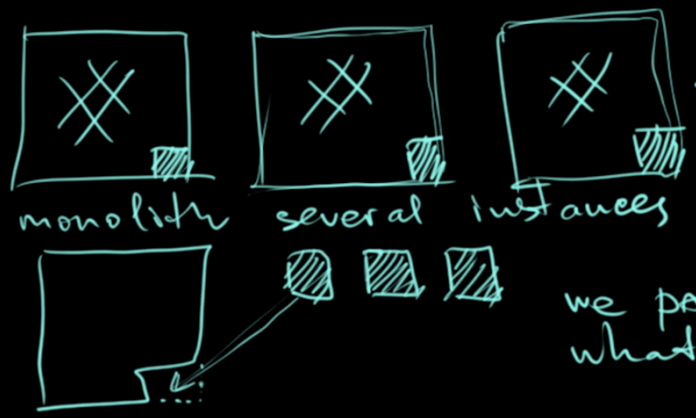
Spring Boot
Flux < A >

int doSmth( ) time cons.
Flux<Int>

t ⇒ to make everything asynchronous

MkReq (body, (is) ⇒ {
callback
})

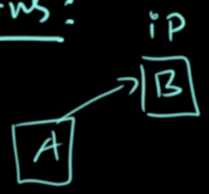
the time is the same but time & resources wouldn't be wasted

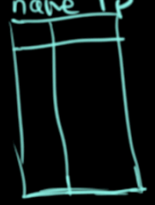


extremely inefficient
90% wasted

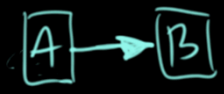
we pay exactly what we have used

Patterns:





- register (for a time period)
- unregister
- query

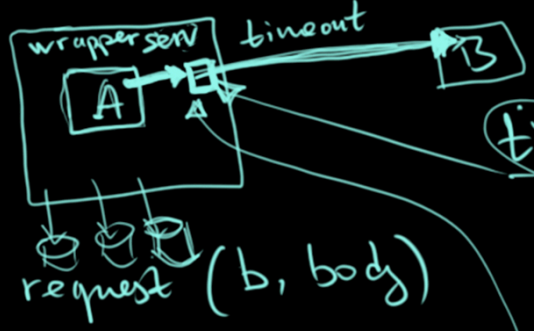


- time out
- retry

timeout forward / retry forward

Decorator (OOP)

side-car



timeouted Request (request (b, body))

we can flexible timeout calls without touching code

retry

repeat with configurable strategy

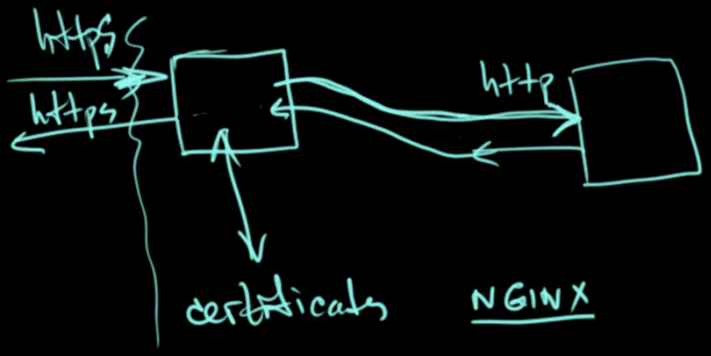
log time

time = current
forward Request
delta = current - time

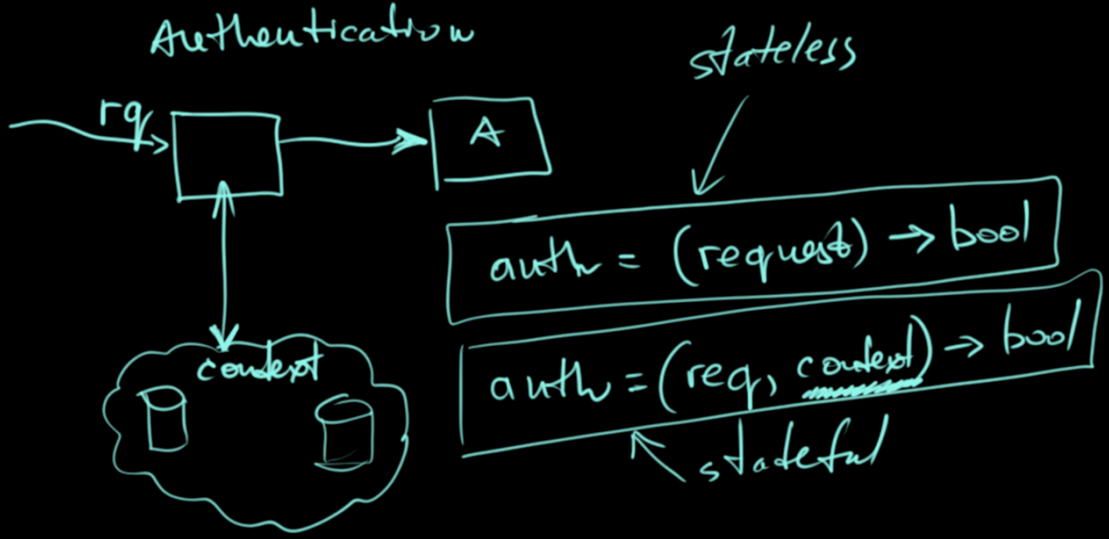
logging

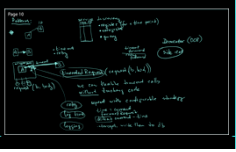
intercept, write them to db

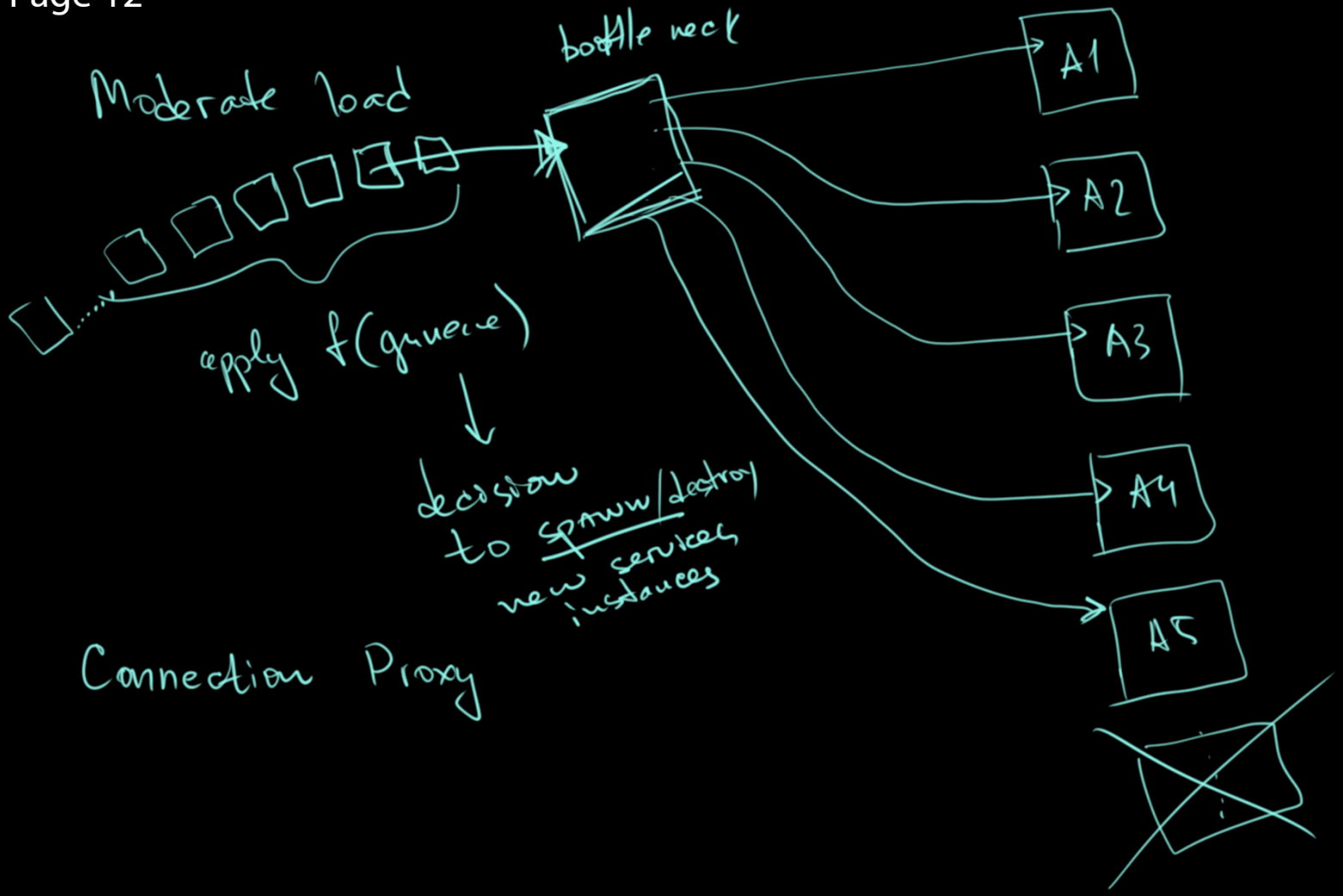
Facades / Proxies

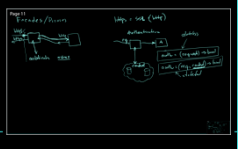


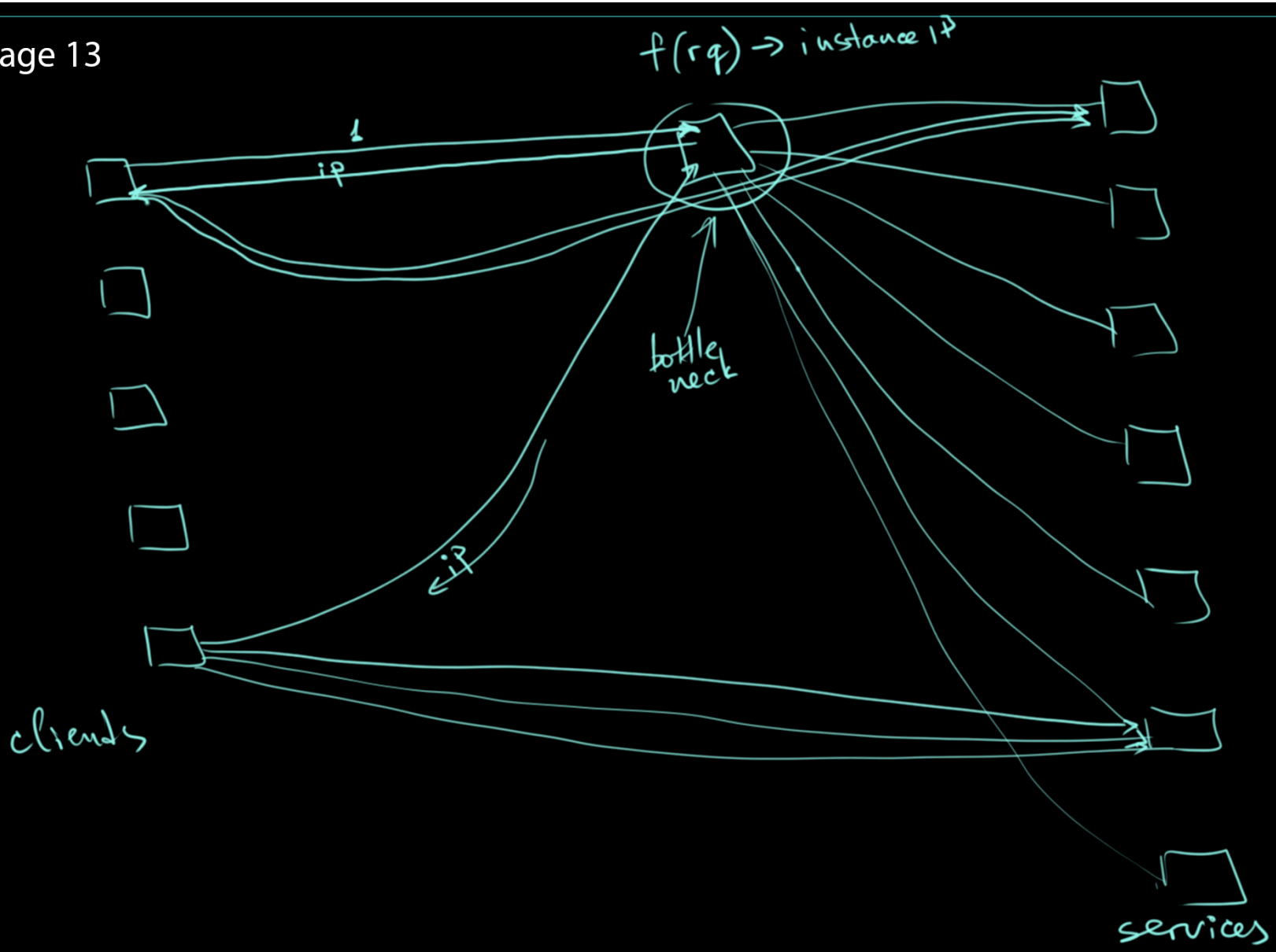
https = SSL (http)

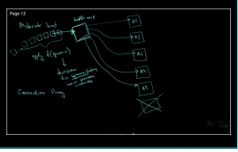




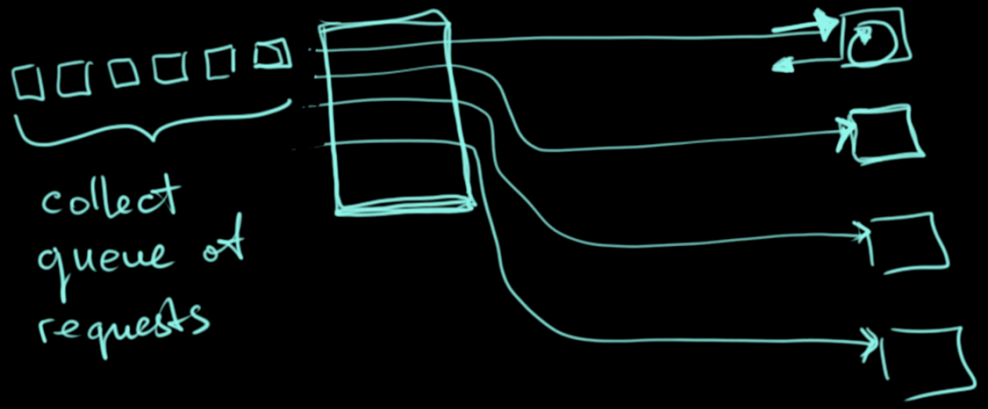

bottle neck
Moderate load
A1
A2
A3
A4
A5
apply f(queue)
decision
to spawn/destroy
new services
instances
Connection Proxy




f(rq) -> instance IP
1
ip
bottle neck
ip
clients
services



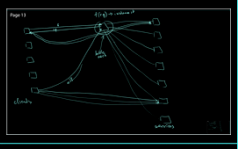
services can handle exactly one request at the time



backpressure

and forward the next one
ONLY when we have service ready to process

99% of patterns have direct "friends" in coding patterns
problems are same ⟷ solution a bit different
any service combination or pattern AS function composition in coding

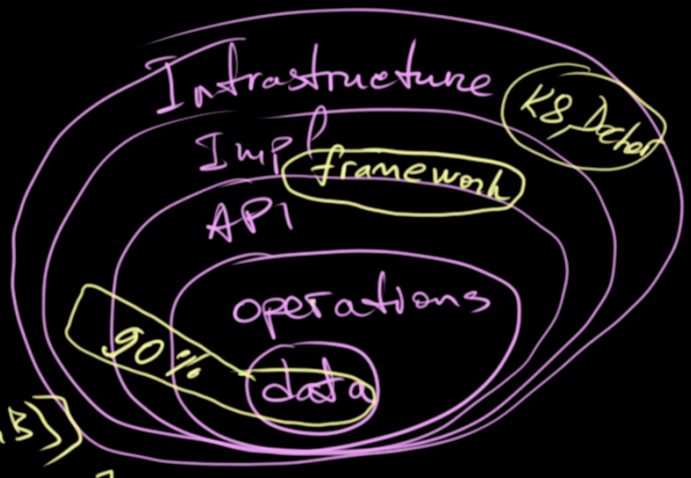


new project from a STRUCTURED MONOLITH

well structured code is always N-tier architecture
functional code is always N-tier architecture.

- domain
- relation
- operation
- persistence
- API
- implementation

} N-tier architecture



f: A => B
f: A => Either [Error, B]
f: A => Option[Either [Error, B]]
f: HttpRequest => Option[HttpResponse]