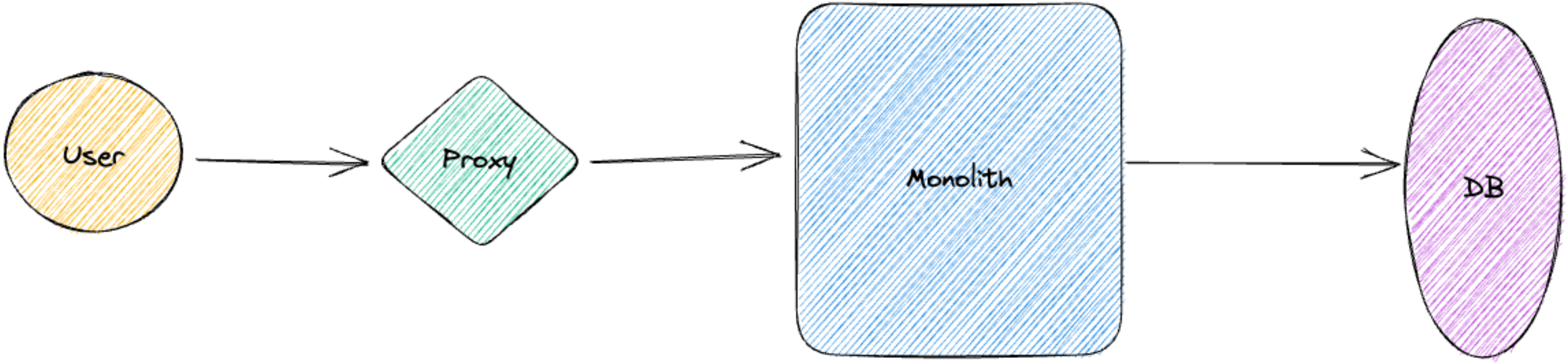
Basic patterns of reliable microservice architecture

@alexgaas

Gear Store – that’s going to be our example



- Gear Store is a system to delivery any gear you want from our gear store to your front porch
- We have this service as classic monolith already with simple schema – there is proxy/gateway in front of monolith and database behind it to store any data
- As **engineers** we decided to make any new features as separate microservices

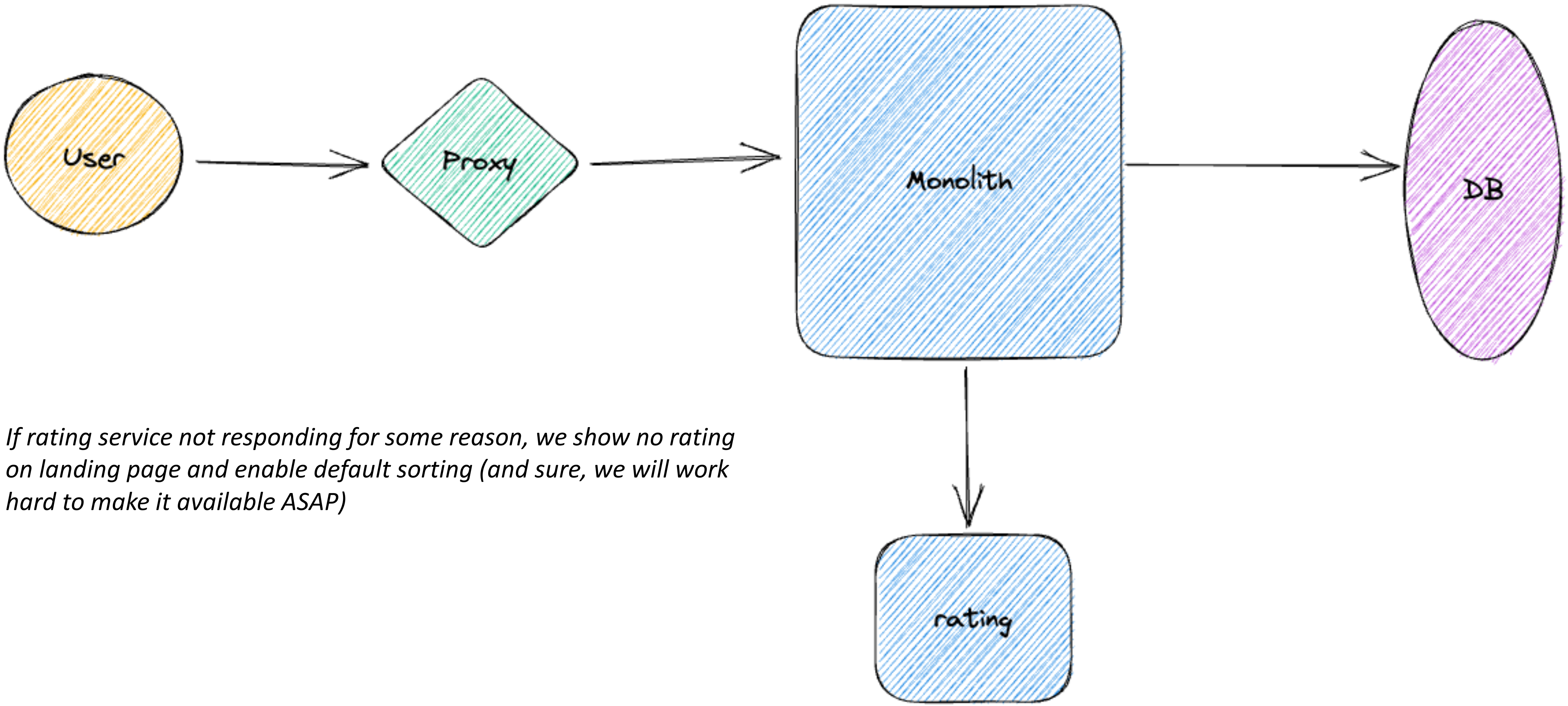
What we will look in aspect of reliable architecture patterns

- Retry
- Idempotency keys
- Deadlines
- Rate limiting
- Circuit breaker

RETRY

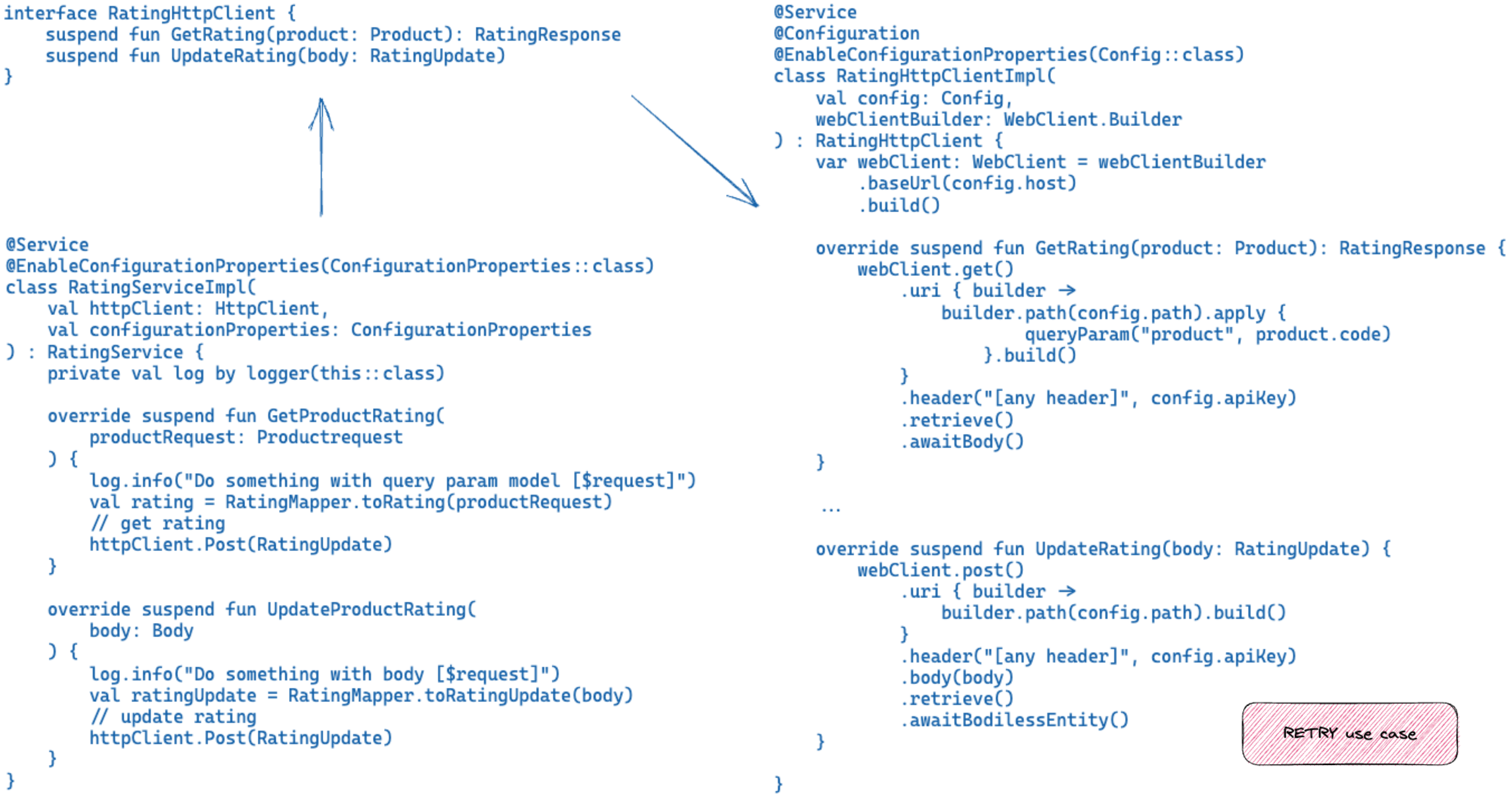
- We as **engineers** getting request from solution owner to implement system of dynamic rating for any slalom gear product (currently ratings for products been hardcoded in database)
- We need to implement a service to calculate rating dynamically for each product based on user feedback and use that rating numbers later to sort out product list
- Great! We understand this feature could be great candidate to build our first microservice out of monolith!

RETRY use case



If rating service not responding for some reason, we show no rating on landing page and enable default sorting (and sure, we will work hard to make it available ASAP)

RETRY use case

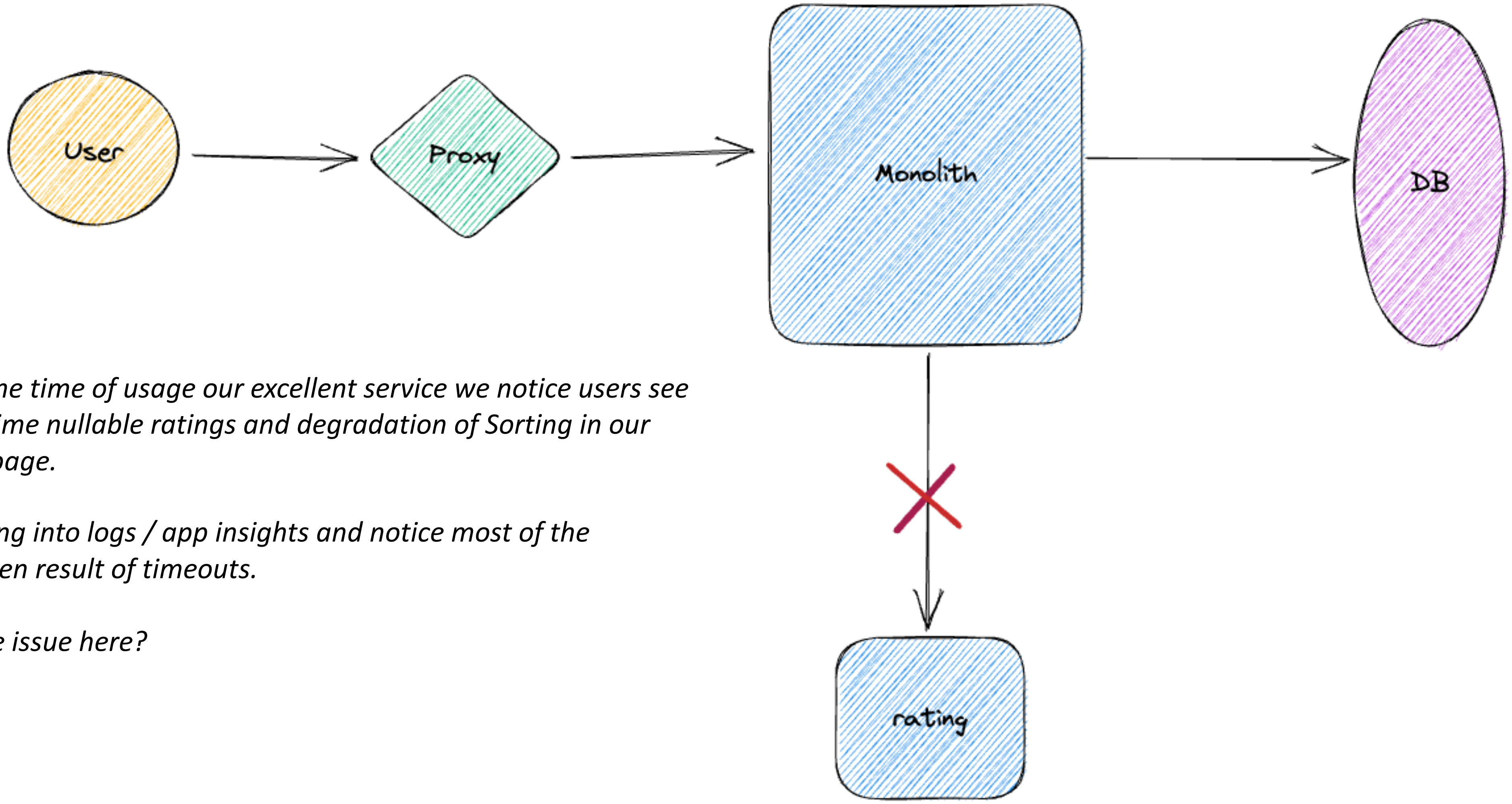


```
interface RatingHttpClient {
    suspend fun GetRating(product: Product): RatingResponse
    suspend fun UpdateRating(body: RatingUpdate)
}
```

```
@Service
@EnableConfigurationProperties(ConfigurationProperties::class)
class RatingServiceImpl(
    val httpClient: HttpClient,
    val configurationProperties: ConfigurationProperties
) : RatingService {
    private val log by logger(this::class)

    override suspend fun GetProductRating(
        productRequest: Productrequest
    ) {
        log.info("Do something with query param model [$request]")
        val rating = RatingMapper.toRating(productRequest)
        // get rating
        httpClient.Post(RatingUpdate)
    }

    override suspend fun UpdateProductRating(
        body: Body
    ) {
        log.info("Do something with body [$request]")
        val ratingUpdate = RatingMapper.toRatingUpdate(body)
        // update rating
        httpClient.Post(RatingUpdate)
    }
}
```

```
@Service
@Configuration
@EnableConfigurationProperties(Config::class)
class RatingHttpClientImpl(
    val config: Config,
    webClientBuilder: WebClient.Builder
) : RatingHttpClient {
    var webClient: WebClient = webClientBuilder
        .baseUrl(config.host)
        .build()

    override suspend fun GetRating(product: Product): RatingResponse {
        webClient.get()
            .uri { builder ->
                builder.path(config.path).apply {
                        queryParam("product", product.code)
                    }.build()
            }
            .header("[any header]", config.apiKey)
            .retrieve()
            .awaitBody()
    }

    ...

    override suspend fun UpdateRating(body: RatingUpdate) {
        webClient.post()
            .uri { builder ->
                builder.path(config.path).build()
            }
            .header("[any header]", config.apiKey)
            .body(body)
            .retrieve()
            .awaitBodilessEntity()
    }

}
```

RETRY use case



After some time of usage our excellent service we notice users see time to time nullable ratings and degradation of Sorting in our landing page.

We looking into logs / app insights and notice most of the errors been result of timeouts.

What the issue here?

RETRY use case

Problem:

Network errors

- 110 Connection timed out / …
- 100 Network is down / 101 Network is unreachable
- 111 Connection refused / …

Service timeout and errors:

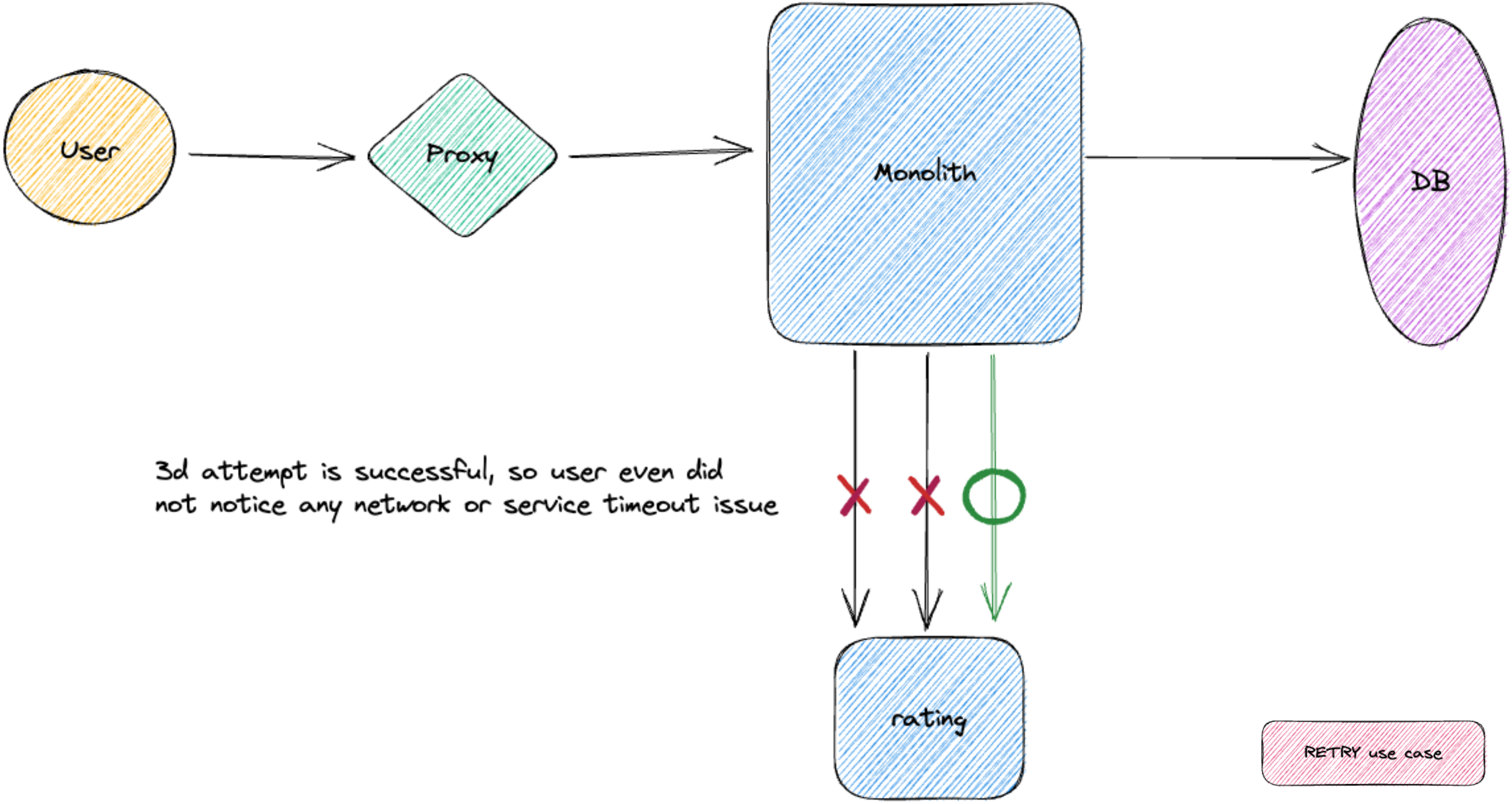
- Timeout: 504 Gateway Timeout / 408 Timeout occurred
- Retry-after: 503 Service Unavailable / 429 Too many requests
- 500 Internal service error (we all “love” this error!!)

RETRY use case

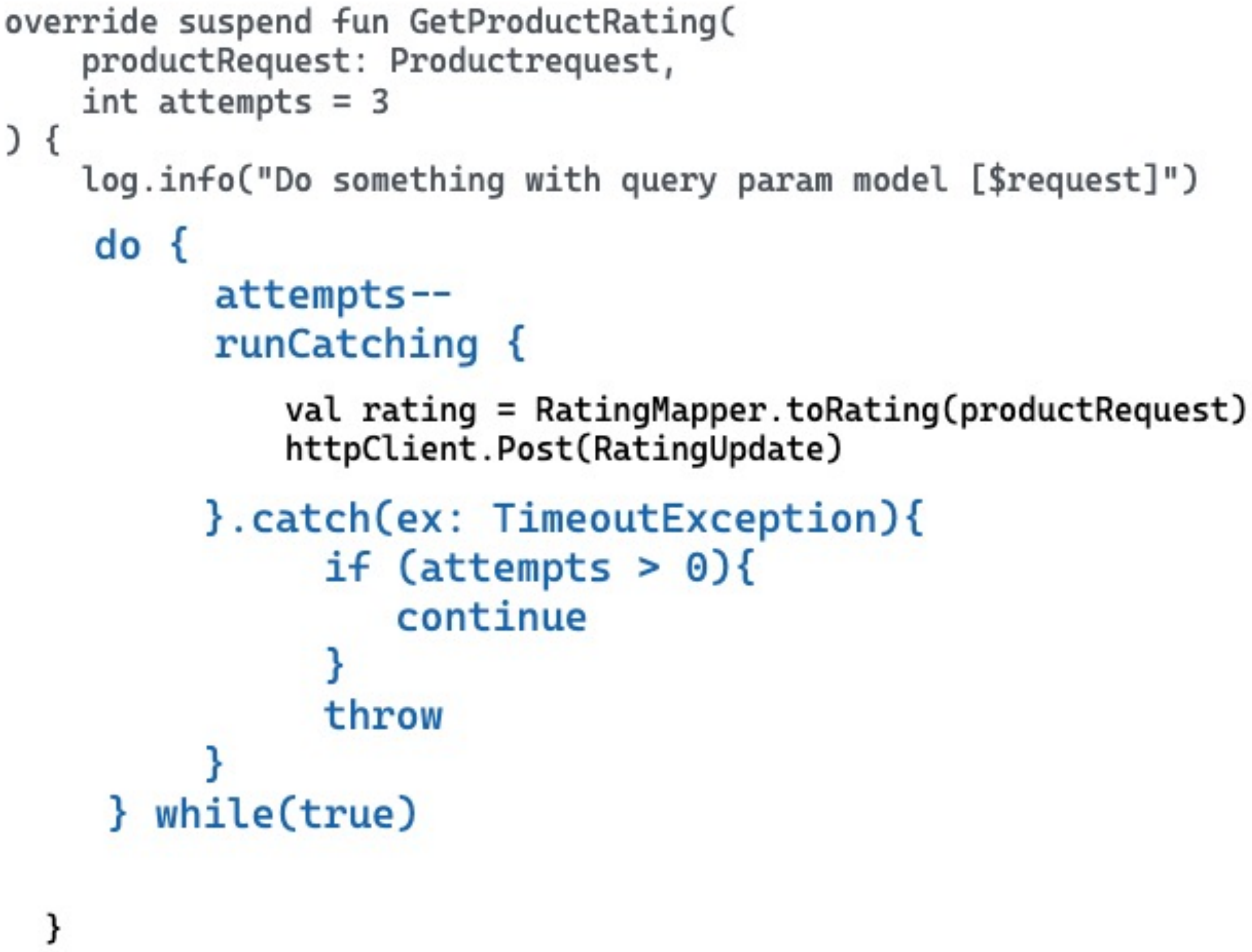
Solution

- RETRY request (we do not ask user make refresh every time when it fails, we just do it for him, and he even will not notice that!)
- With restriction of number of requests (we will strictly try 3 times by example)
- With restriction of general awaiting time (we ready to await rating by example 200ms including maximum number of retry attempts)
- Exponential backoff and jitter - https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/
- Follow http standard – pull out headers and find for Retry-After

RETRY use case



3d attempt is successful, so user even did not notice any network or service timeout issue



```
override suspend fun GetProductRating(
    productRequest: Productrequest,
    int attempts = 3
) {
    log.info("Do something with query param model [$request]")
    do {
          attempts--
          runCatching {

              val rating = RatingMapper.toRating(productRequest)
              httpClient.Post(RatingUpdate)

          }.catch(ex: TimeoutException){
                if (attempts > 0){
                    continue
                }
                throw
          }
     } while(true)


  }
```

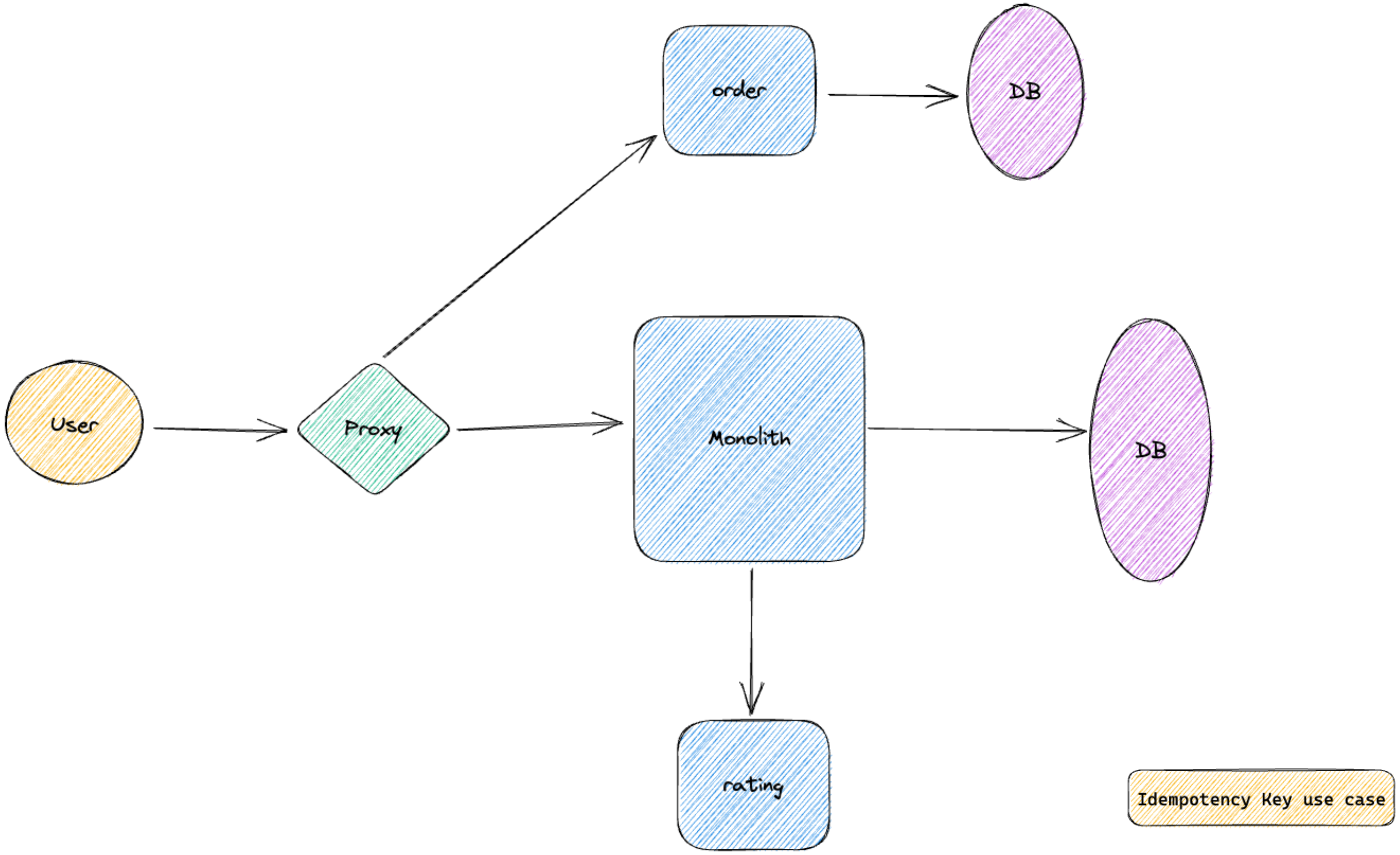
Spring Boot Retry implementation - **https://github.com/spring-projects/spring-retry**

RETRY use case

Idempotency key

- Our solution owner produces new idea – multi-basket which allow few users to share one order!
- That’s amazing opportunity for us to make a new service Order. Microservice design in action comes true!!!
- We keep in mind to implement retries for Order service is worth based on our previews experience with rating service. We using same code with 3 attempts as for rating service

Idempotency Key use case

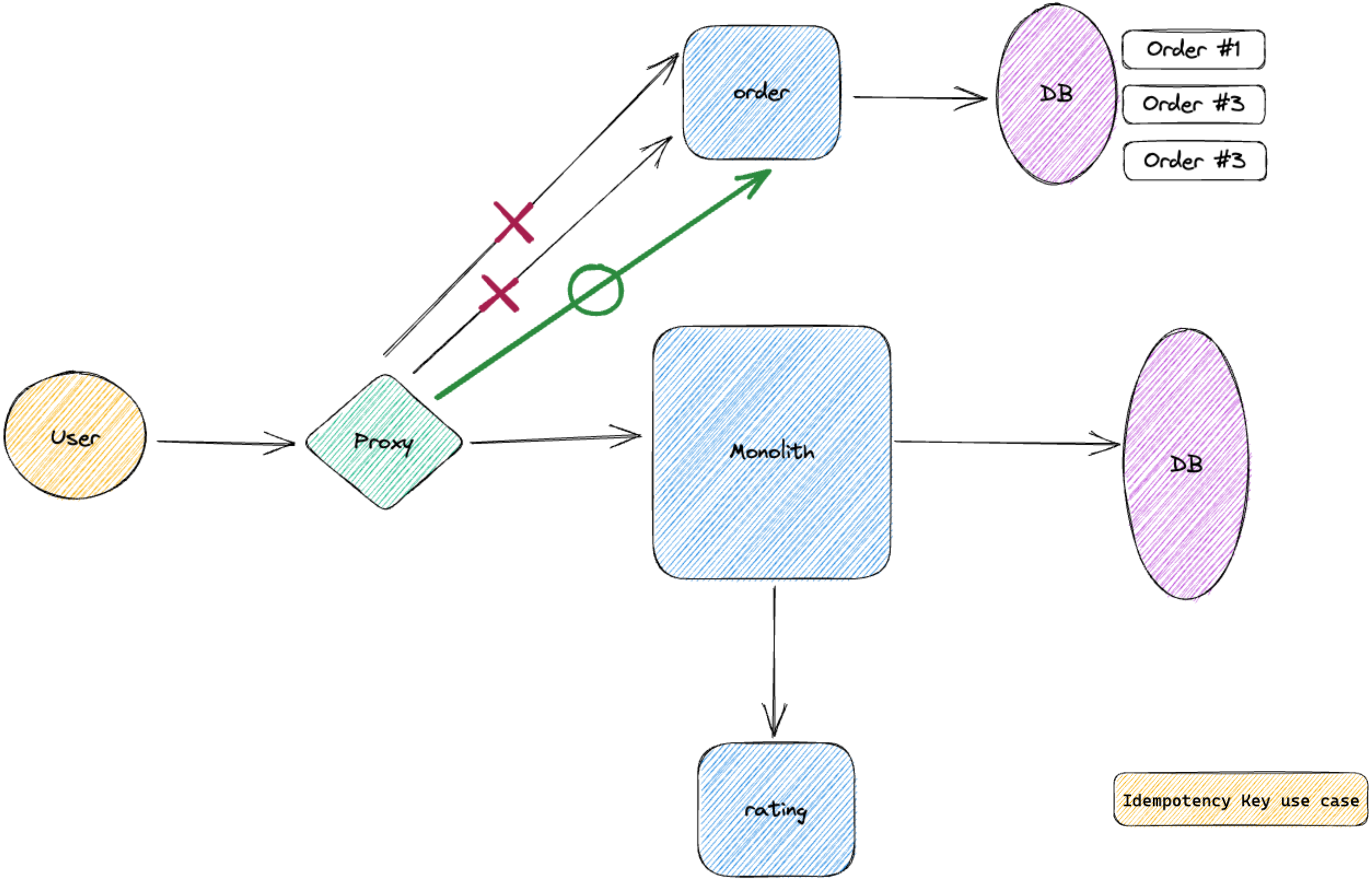


Idempotency Key use case

Problem

- User makes order first time and some error happen on the route between order service and proxy. But service successfully saves order
- User makes attempt again with same result. No 200k response but order been saved successfully by service
- User trying third time and getting back 200k. Success!
- But as result users gets three same purchases and money withdrawal 3 times
- User is NOT happy, he calls to customer support

Idempotency Key use case

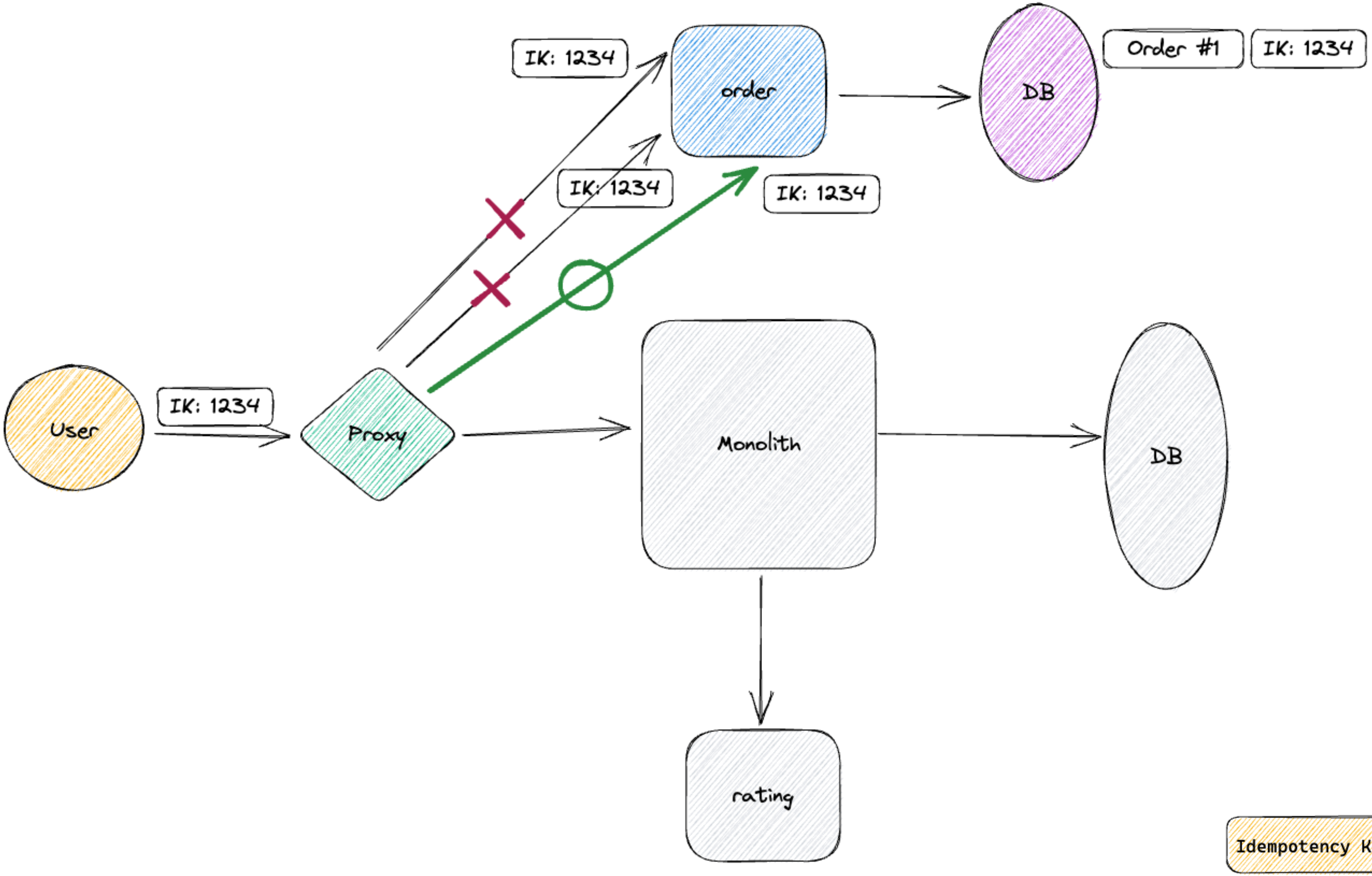


Idempotency Key use case

Solution

- What’s happened? Network can fail not only on sending but also on receiving of data and that’s usual situation.
- As solution we must ask our user/frontend to generate any random string (idempotency key)
- Order service just validates by idempotency key and do not create any duplicate of order in case of failure
- As outcome just adding one key between frontend and service, we implement semantic “at most once”
- Look on this short article from Stripe about their IK implementation findings **- https://stripe.com/blog/idempotency**

Idempotency Key use case



Idempotency Key use case

```
override suspend fun CreateOrder(
    order: OrderRequest,
    requestContext: RequestContext,
    idempotencyKey: String,
    attempts: int = 3
) {
    do {
        attempts--;
        runCatching {
            httpClient.CreateOrder(order, idempotencyKey)
        }.catch(ex: TimeoutException){
            // now we analyze not only number of retries
            // but also validate if http call is retryible
            if (attempts > 0 && HttpClient.IsRetryible(requestContext)){
                continue
            }
            throw;
        }
    } while(true)
}
```

```
override suspend fun CreateOrder(order: OrderRequest, idempotencyKey: String) {
    return webClient.get()
        .uri { builder ->
            ...
        }
        // We put a [X-Idempotency-Key] into the request
        .header("X-Idempotency-Key", idempotencyKey)
        .body(order)
        .retrieve()
        .awaitBodilessEntity()
}
```

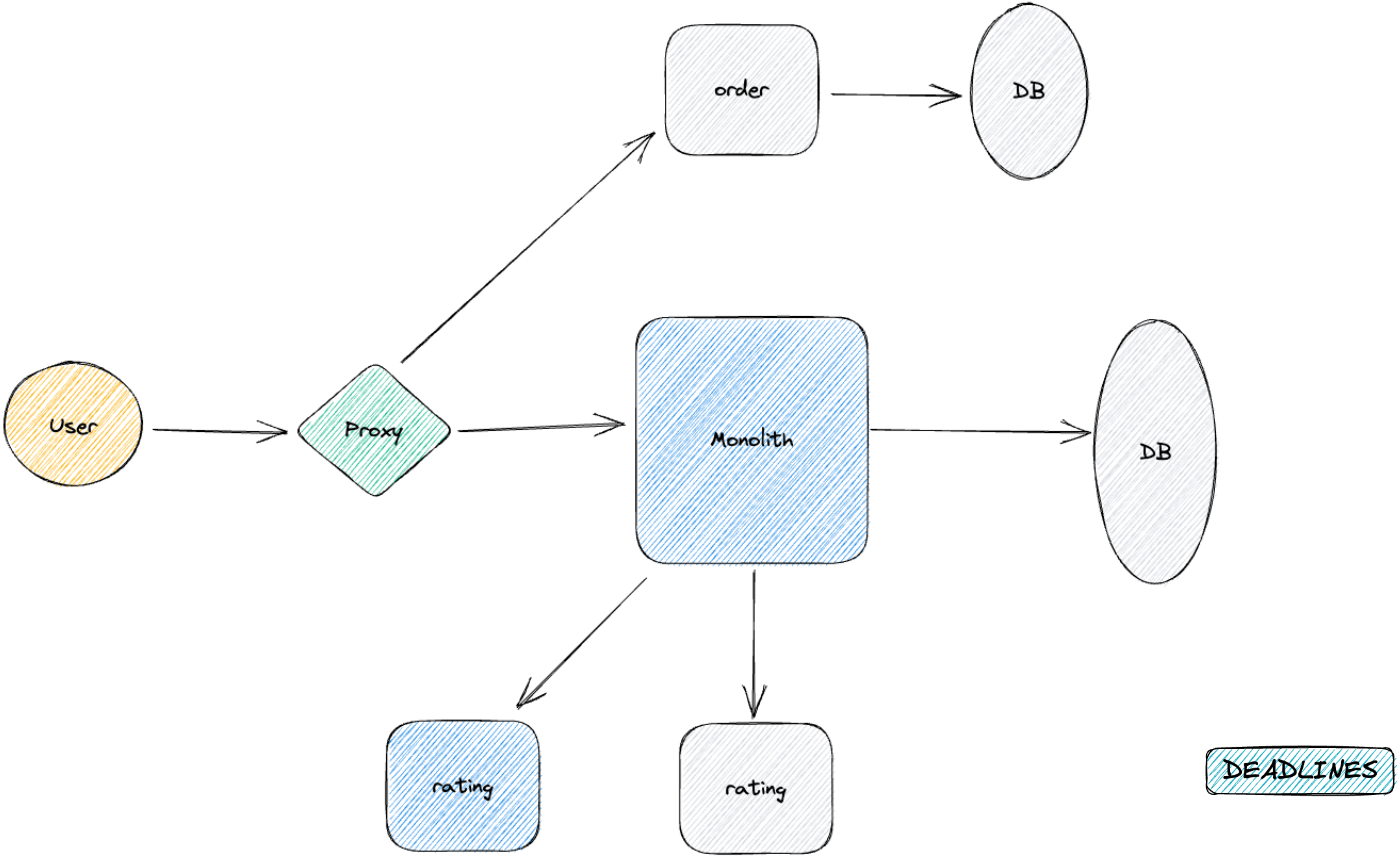
```
fun IsRetryible(context: HttpContext): Boolean {
    // since in according to HTTP standard that requests methods immutable
    // we can retry them N times without any side effects
    if (context.Method == "GET" || context.Method == "HEAD"
        || context.Method == "OPTIONS" || context.Method == "TRACE") {
        return true
    }

    // if we have IK it does not matter which method we use, we in chardge of retry since we get how that suppose to work
    if (context.Headers.Contains("Idempotency-Key") || context.Headers.Contains("X-Idempotency-Key")){
        return true
    }

    return false
}
```

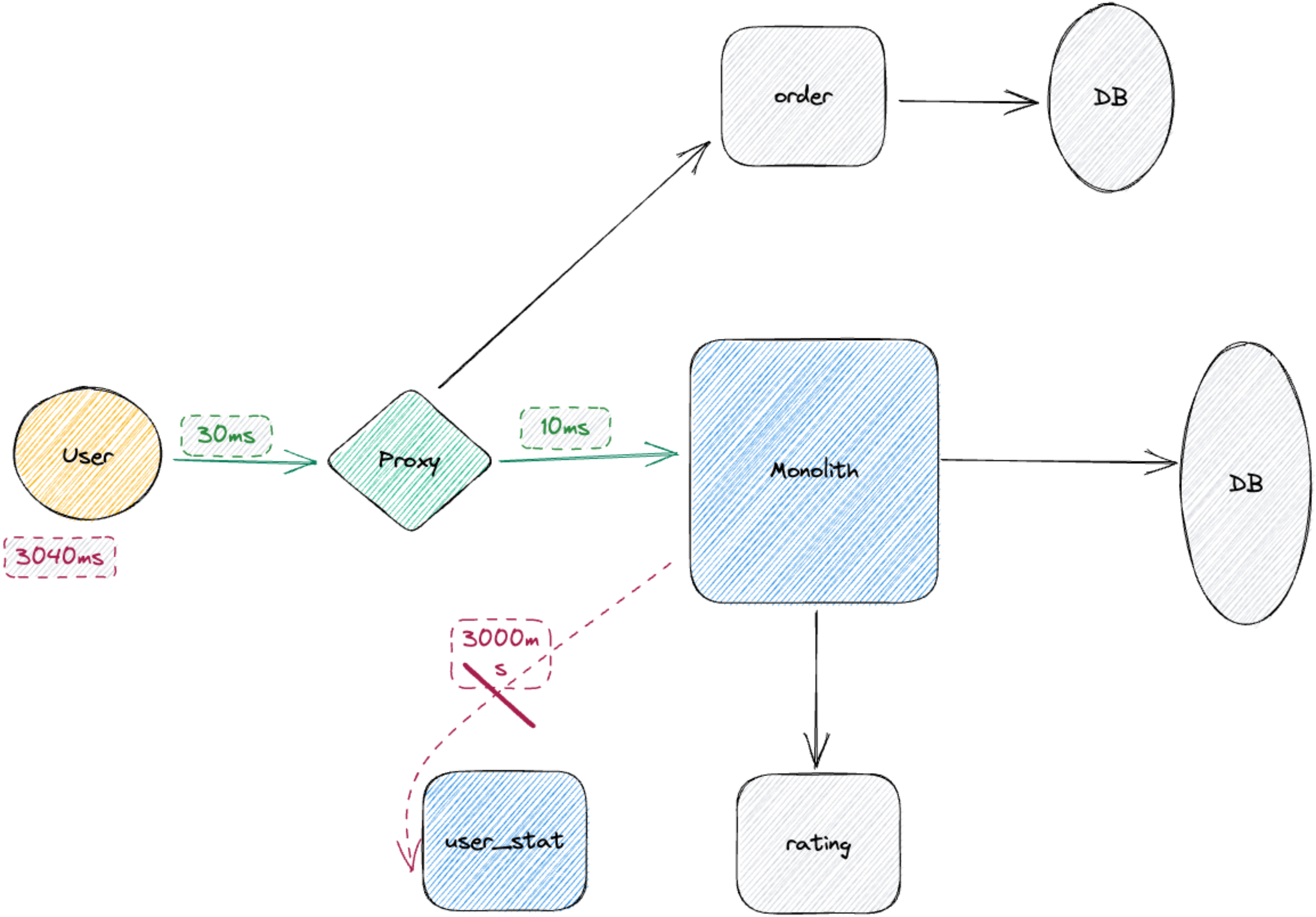
Idempotency Key use case

Deadlines

("Deadlines" as a term been introduced by Google when they released grpc framework)

- New feature from solution owner – user statistics. That statistics from user suppose to provide us data about user's favorite gear!
- In fact, that's aggregation of user statistics
- So far so good - as outcome of discuss with SO we make a new service "user_stat"


order
DB
User
Proxy
Monolith
DB
rating
rating
DEADLINES


order
DB
User
30ms
Proxy
10ms
Monolith
DB
3040ms
3000m
s
user_stat
rating

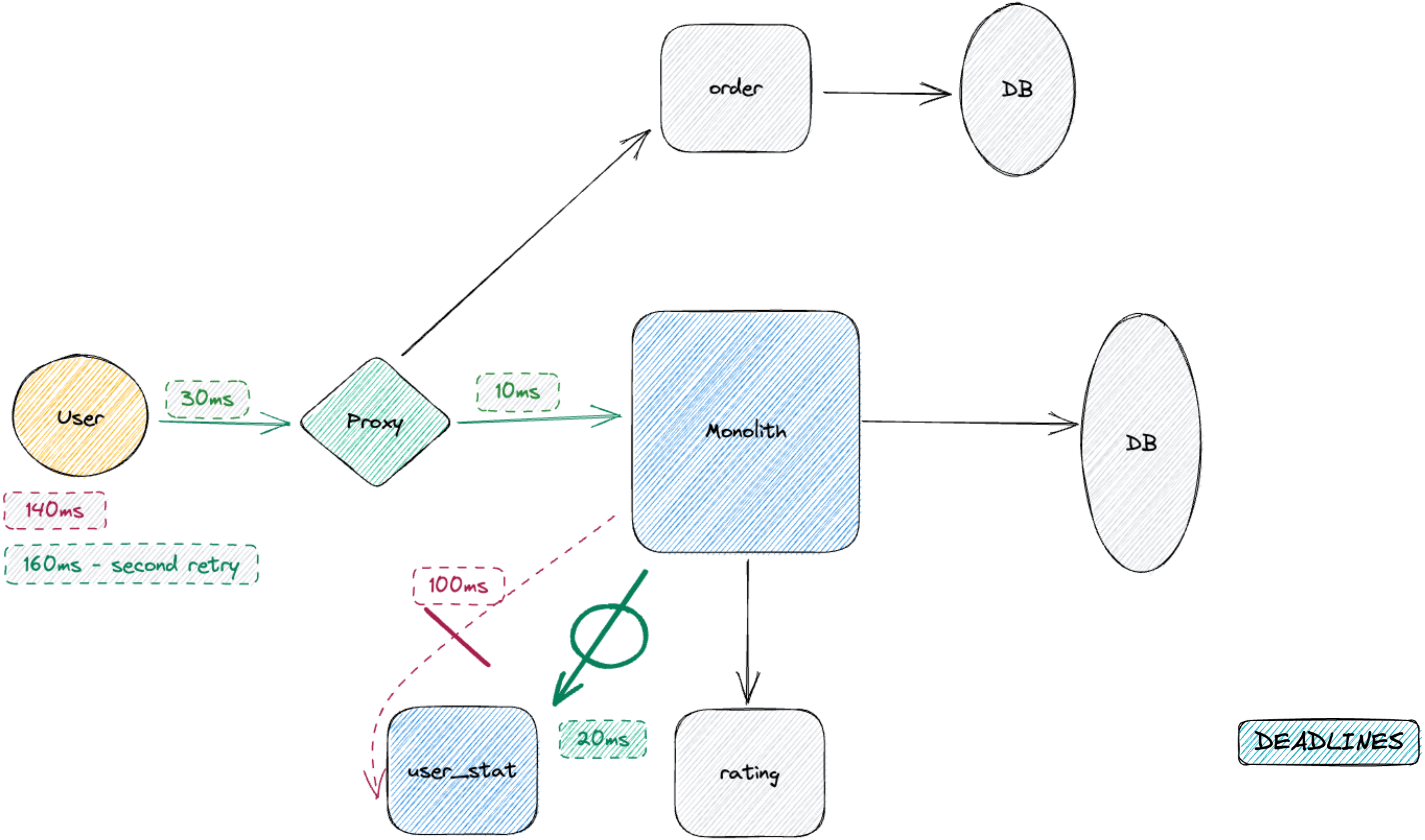
DEADLINES

Problem

- Network timeouts
- Connection drop
- Instance/pod issue (issue with container or cloud, problem with your source code)
- Requested resource is not available

Solution

- Define await time on client side (we on monolith as client ready to wait no more then 100ms by example. If time of awaiting is more then 100ms we break up connection and try again)
- Setup deadline propagation. We explicitly say to requester service how long we expect to await response from him. Using that information requester server can fallback to simpler workflow based on estimation / etc.
- As part of solution, we may setup only for one retry or either general time based on our strategy (we ready to wait 100ms per request or 300ms in general to await response)
- Configure timeouts based on service demand. Each service must have his own configurable timeouts depends on use case
- In addition, setup deadlines by 99% percentile .


order
DB
User
30ms
Proxy
10ms
Monolith
DB
140ms
160ms - second retry
100ms
20ms
user_stat
rating
DEADLINES

```
interface Context {
 // set context with timeout
 SetContext(deadline: Instant)

 // cancel context
 fun Cancel()
 fun Canceled(): Boolean

 // get deadline info
 fun Deadline(): Instant

 // start lambda with context
 fun Execute(function: (ctx: Context) → (Unit))
}
```

```
fun Canceled(): Boolean {
 if (_canceled){
  return true
 }

 if (_deadline){
  return DateTime.Now().Instant ≥ _deadline
 }

 return false
}
```

our API (monolith service impl)

```
override suspend fun GetUserStat(
 userStatRequestBody: UserStatRequestBody
): Response {
    log.info("Do something with model [$request]")


    val deadline = Instant.Now().toEpochMilli() + 100;
    val context = Context(deadline: deadline)

    val userStatRequest = UserStatMapper.toUserStatRequest(productRequest)


    // client will call ctx.Deadline to get deadline point
    return httpClient.Get(ctx, userStatRequest)
}

 ...
```

On user stat service:

UserStatController → UserStatService →

```
fun RetrieveUserStat(request: Request){
 // Parse X-DEADLINE header
 val ctx = req.GetContextFrom(request)

 if (ctx.Canceled()){
  throw BadRequestException("Client closed request", status_code: 499)
 }

 return ctx.Execute((ctx: Context)→{
  val future = ...  // database call here to get some data

  val deadline = ctx.Deadline()
  if (deadline ≠ null && future.wait_until(deadline == future.status.timeout)){
   throw DeadlineExceededException()
  }

  return future.get()
 })
}
```

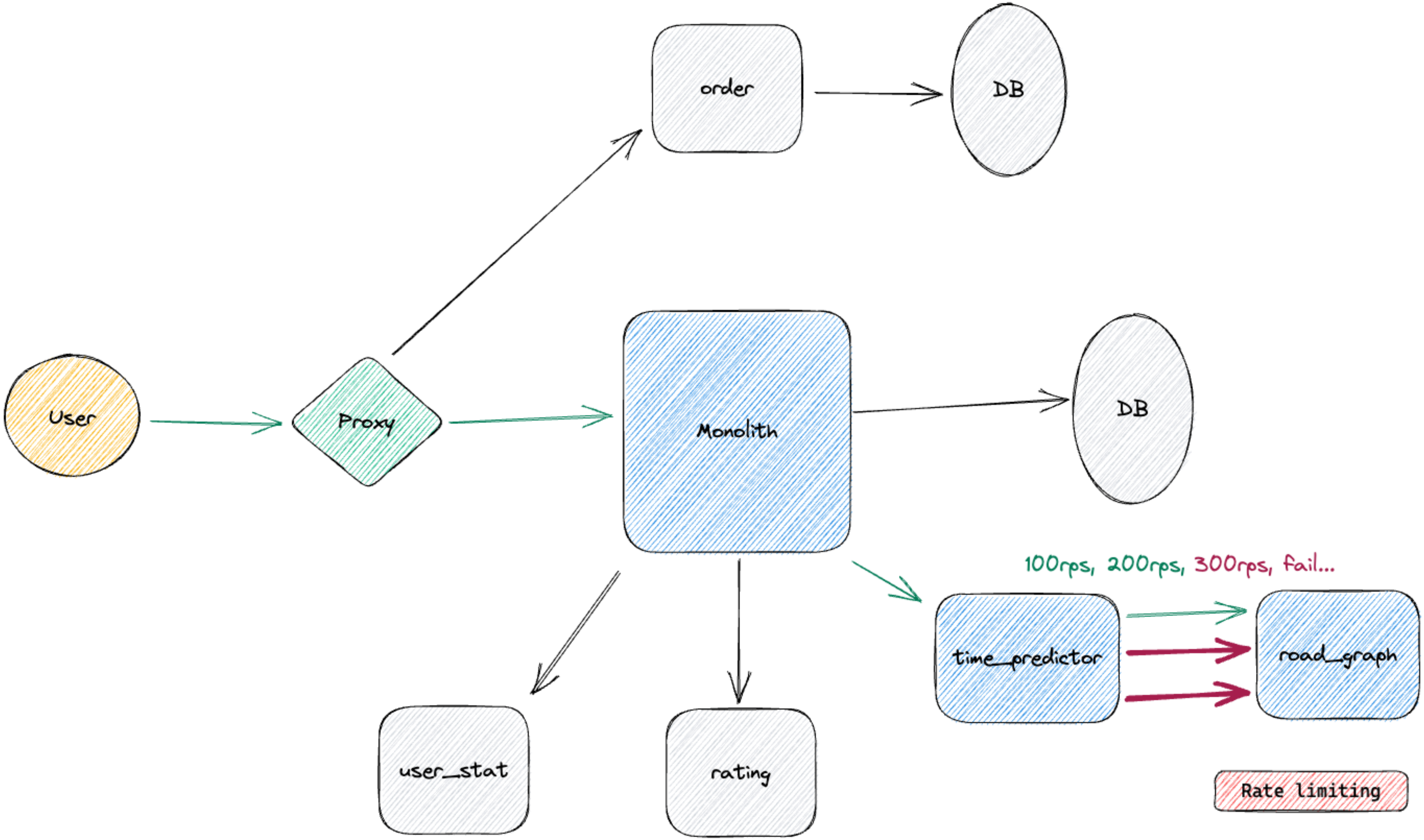
httpClient also must setup header as:

```
X-Deadline: 2023-06-09:00:00:000Z
```

DEADLINES

Rate limiting

- New use case from our solution owner – we need to predict time of delivery of our gear to provide best customer expectations. Wow Slalom Gear Store looks like Uber or Doordash now, so exciting!!
- We decided to have two new microservices – **time_predictor** and **road_graph**. Time Predictor provides user average time of delivery from Slalom Store. Road Graph behind Time Predictor will calculate for us real delivery time from point A to point B on the city map.


order
DB
User
Proxy
Monolith
DB
100rps, 200rps, 300rps, fail...
time_predictor
road_graph
user_stat
rating
Rate limiting

Problem

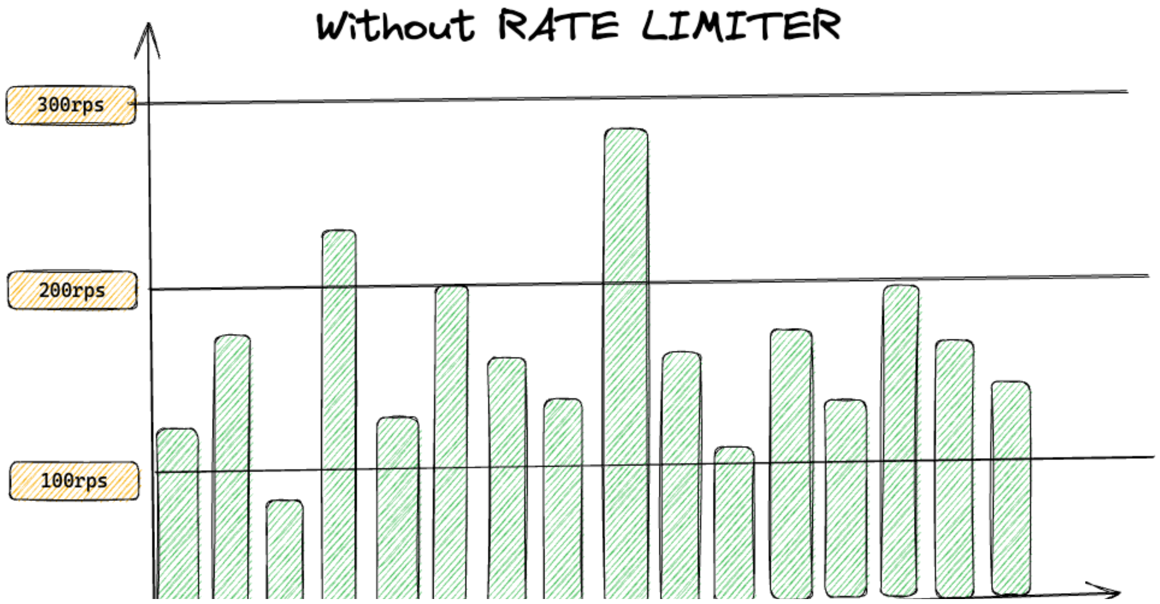
- Service (hardware) resource limits
- Spike traffic growing (one of data centers outage as the reason)
- Unplanned traffic growing: DDOS attack or just natural growing
- Broken client (like infinite loop in client)

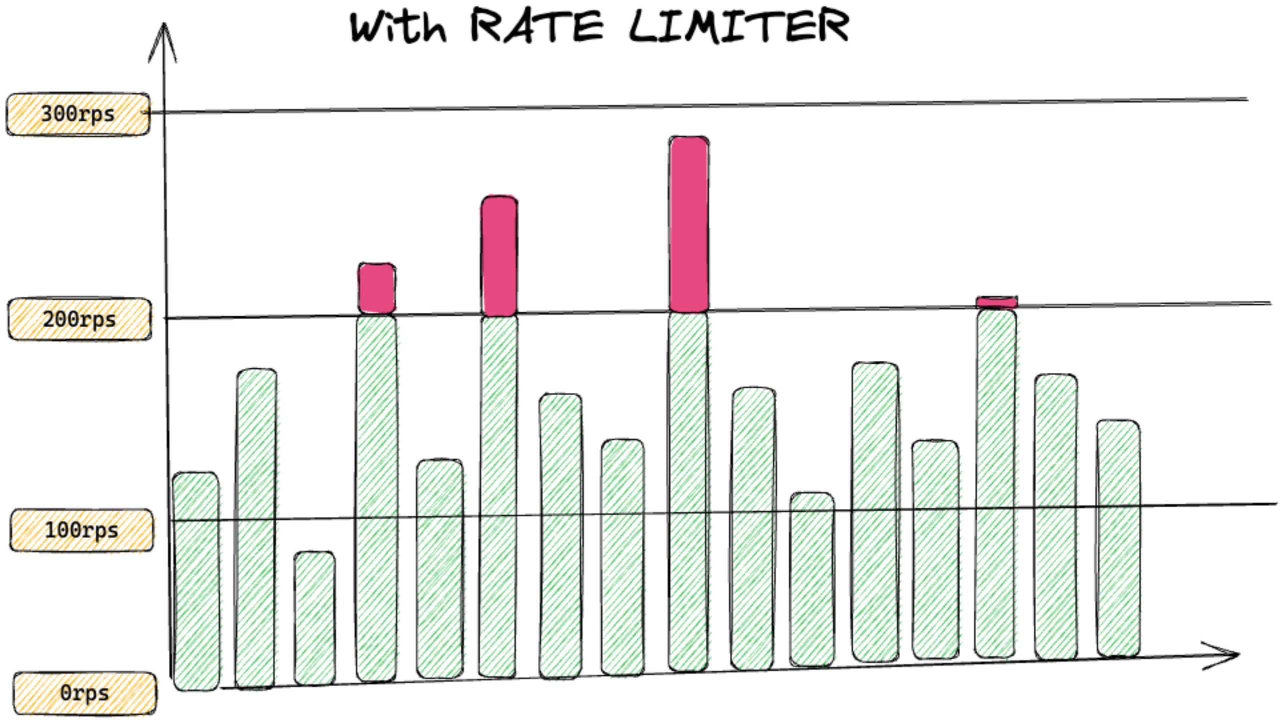
Solution

- Rate limiting – client or server side (limit by quote like 2000rps per 3 seconds or limit by rate – no more then 3 requests per sec by example)
  - Limit rates by number of parallel requests
  - Quote for rate limits (X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset)
  - Standard HTTP code is 429 Too Many Requests
- Burst limiting

  Limit correctly spikes

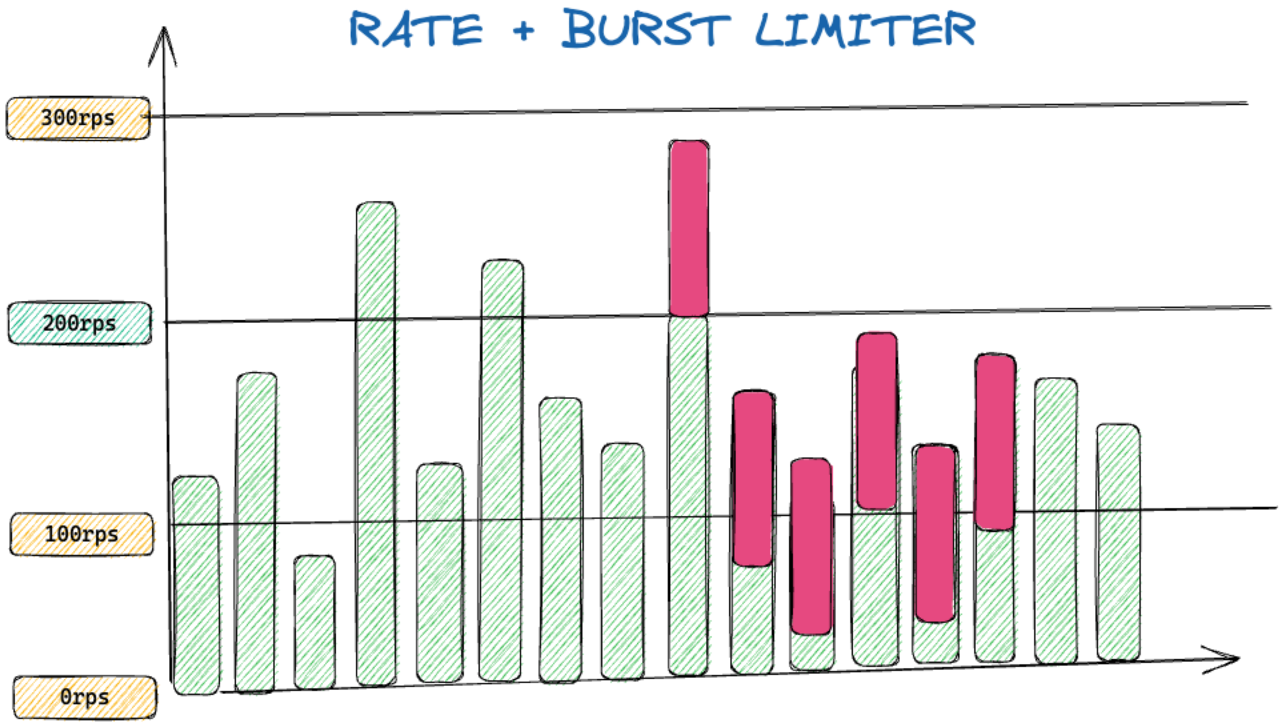
Rate limiting





As you may notice if request just a little bit bigger than rate limit, request will be rejected, and user will need to make retry. To avoid this issue, we may apply burst limiting

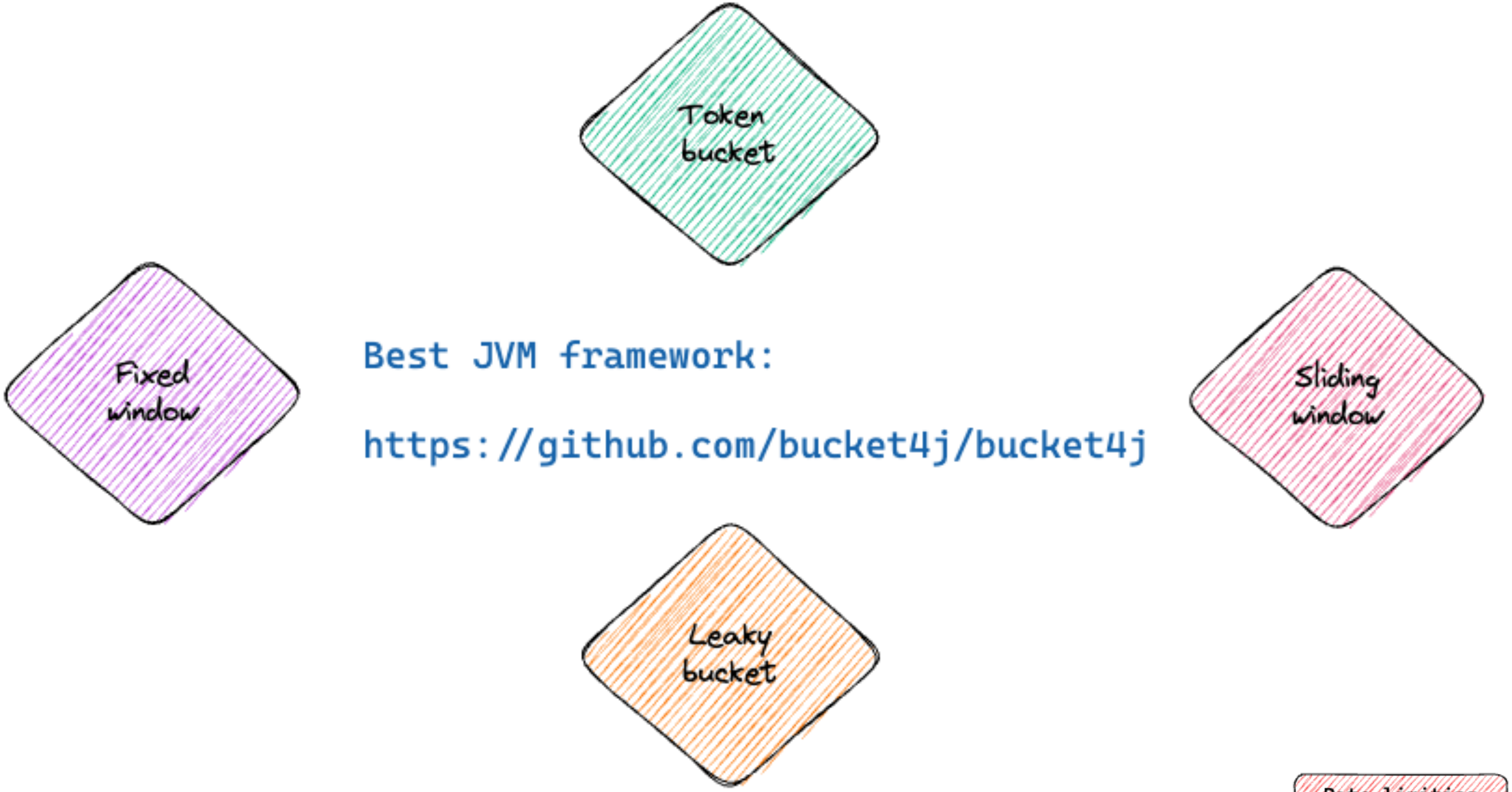
Rate limiting



When we use rate + burst limiting, we define two limits – common rate limit which defined usually not as simple 100rps but 1k requests per 10 seconds. In other word we increase period of rate calculation.
Second limit (burst limiting itself) stricter – if request trying to pass that limit, we immediately must reject that request.
As outcome here – even if user over limited some small time he can “take in debt” assuming his activity will be reduced later. But he going out of the quote or even going out the burst we decrease his possibility to make requests per time.

Rate + Burst limiting distributes our loading more evenly what in most time make user happier (except he is going out of Burst limit).

Rate limiting



Token bucket

Fixed window

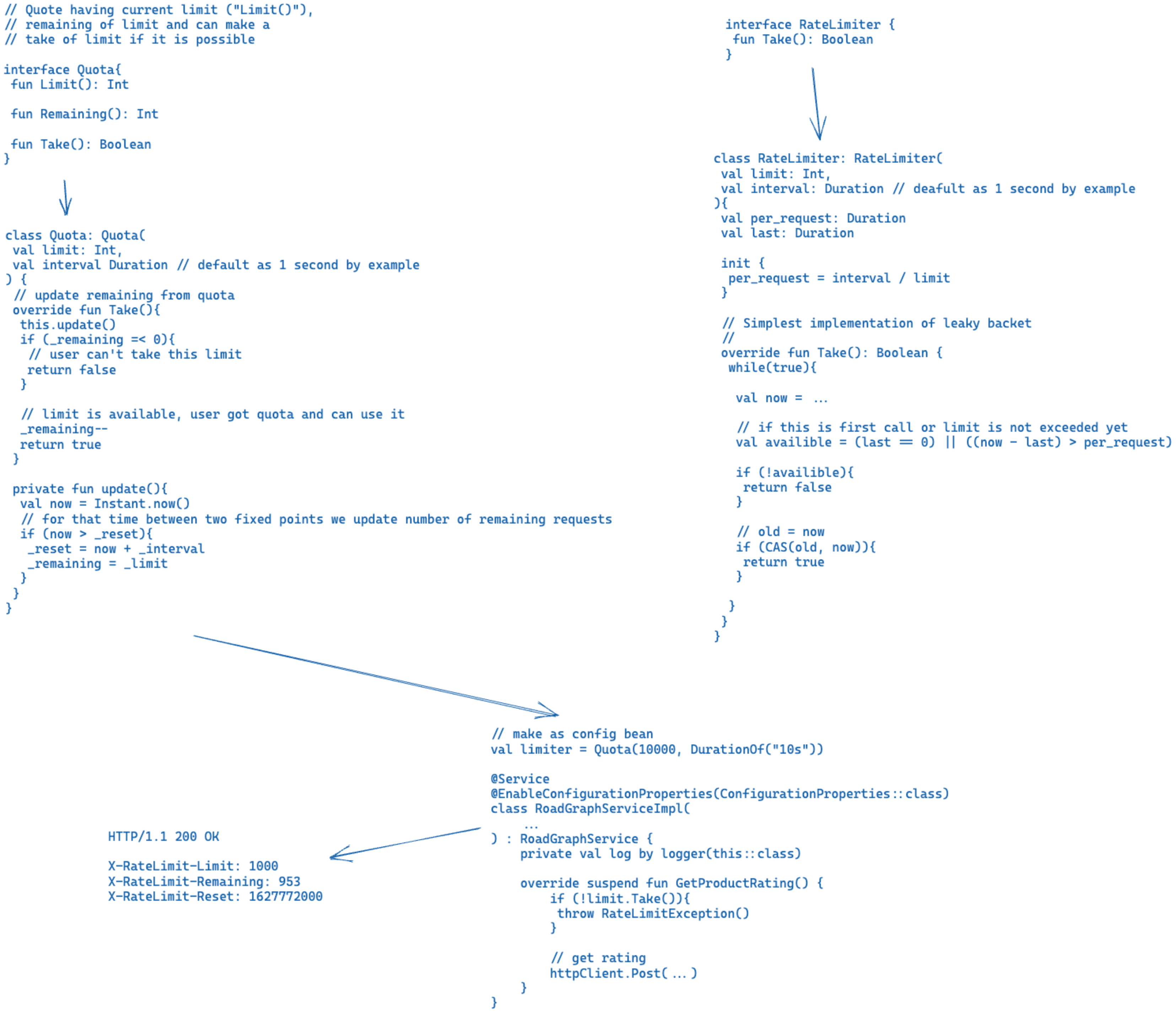
Best JVM framework:

https://github.com/bucket4j/bucket4j

Sliding window

Leaky bucket

Rate limiting



```
// Quote having current limit ("Limit()"),
// remaining of limit and can make a
// take of limit if it is possible

interface Quota{
 fun Limit(): Int

 fun Remaining(): Int

 fun Take(): Boolean
}
```

```
class Quota: Quota(
 val limit: Int,
 val interval Duration // default as 1 second by example
) {
 // update remaining from quota
 override fun Take(){
  this.update()
  if (_remaining =< 0){
   // user can't take this limit
   return false
  }

  // limit is available, user got quota and can use it
  _remaining--
  return true
 }

 private fun update(){
  val now = Instant.now()
  // for that time between two fixed points we update number of remaining requests
  if (now > _reset){
   _reset = now + _interval
   _remaining = _limit
  }
 }
}
```

```
interface RateLimiter {
 fun Take(): Boolean
}
```

```
class RateLimiter: RateLimiter(
 val limit: Int,
 val interval: Duration // deafult as 1 second by example
){
 val per_request: Duration
 val last: Duration

 init {
  per_request = interval / limit
 }

 // Simplest implementation of leaky backet
 //
 override fun Take(): Boolean {
  while(true){

   val now =  ...

   // if this is first call or limit is not exceeded yet
   val availible = (last == 0) || ((now - last) > per_request)

   if (!availible){
    return false
   }

   // old = now
   if (CAS(old, now)){
    return true
   }

  }
 }
}
```

```
// make as config bean
val limiter = Quota(10000, DurationOf("10s"))

@Service
@EnableConfigurationProperties(ConfigurationProperties::class)
class RoadGraphServiceImpl(
    ...
) : RoadGraphService {
    private val log by logger(this::class)

    override suspend fun GetProductRating() {
        if (!limit.Take()){
         throw RateLimitException()
        }

        // get rating
        httpClient.Post( ... )
    }
}
```

```
HTTP/1.1 200 OK

X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 953
X-RateLimit-Reset: 1627772000
```

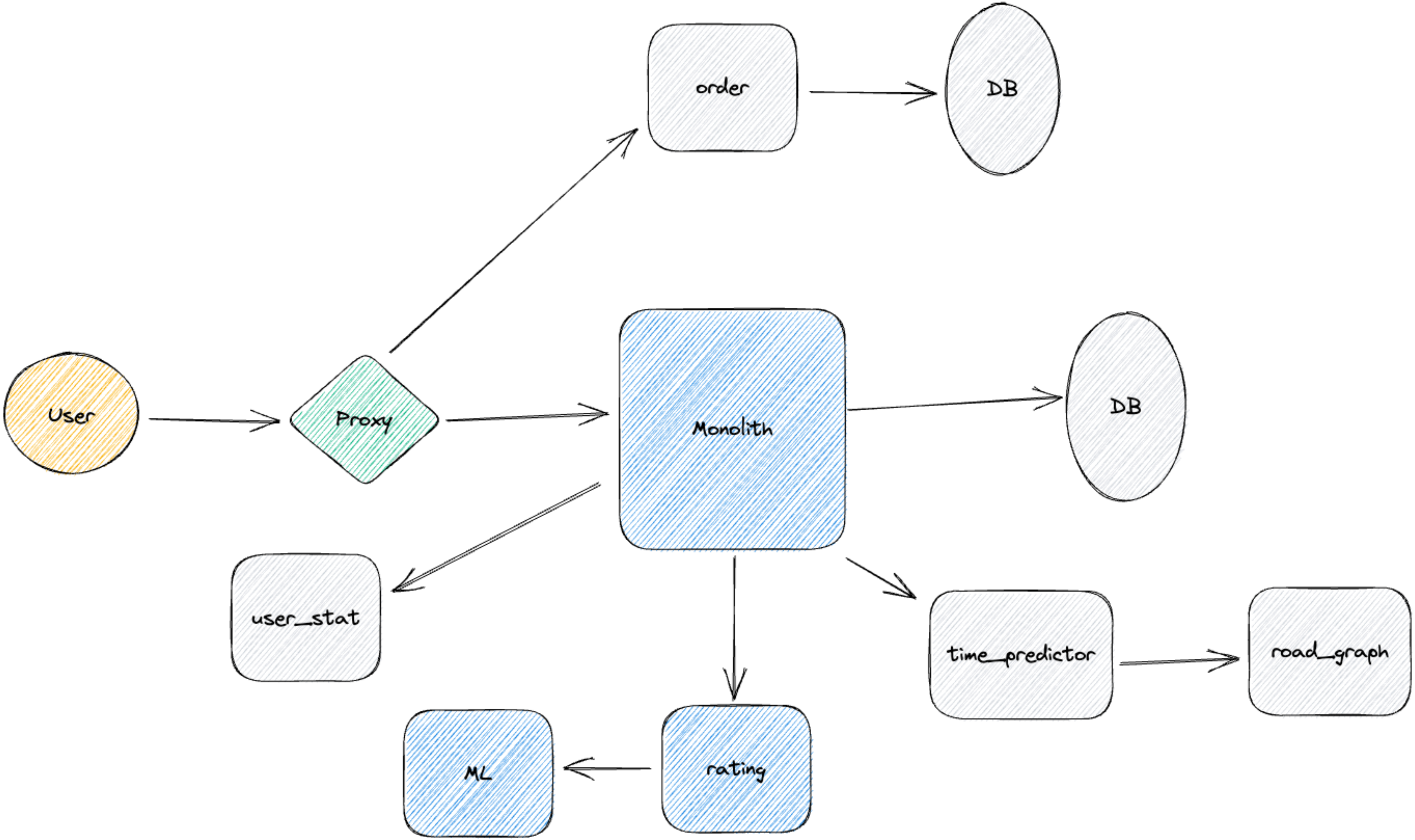
Rate liming finalization

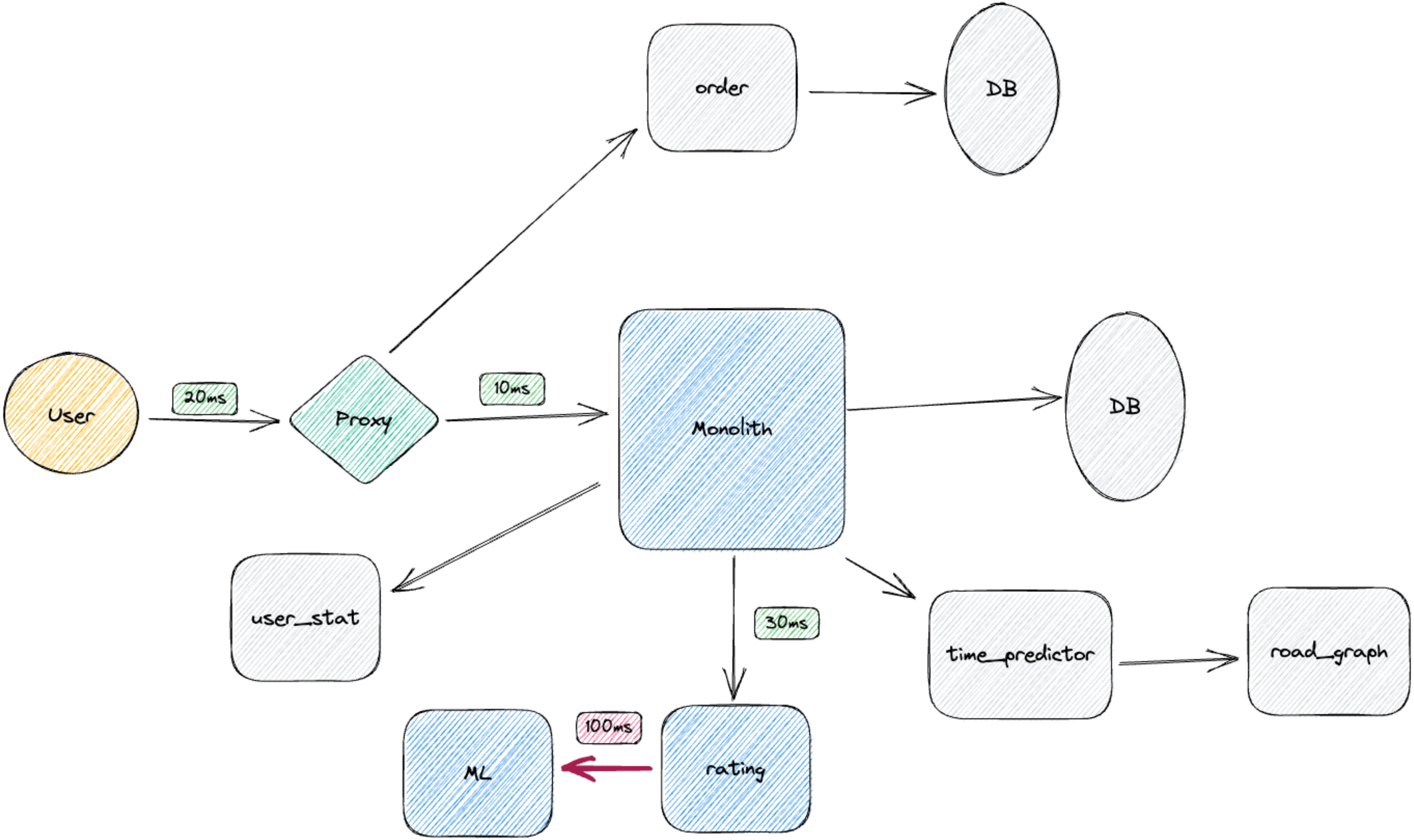
- There is one problem you can meet building your own rate limiting system - if you have many instances of service as part one big system (k8s as example), quotes and remaining of limits should distribited evenly between instances.
- As outcome here - if you making rate limit system you must distribute it evenly between all nodes of your system.
- There are plenty of proxies and cloud solution which provide you rate liming out of box – NGINX, Envoy.

Rate limiting

Circuit breaker

- Ok I got pulled by my solution owner again and he said – our Slalom Build ML department wants add to Slalom Gear new amazing ML service to calculate rating of gear using AI
- Wait what? AI? Have you consulted with Elon already regarding it? He got concerns though…
- Ok but now I'm going to setup retries, make deadlines, settle down rate limiter – all past lessons learnt perfectly.
- But something again not right…


order
DB
User
Proxy
Monolith
DB
user_stat
time_predictor
road_graph
ML
rating

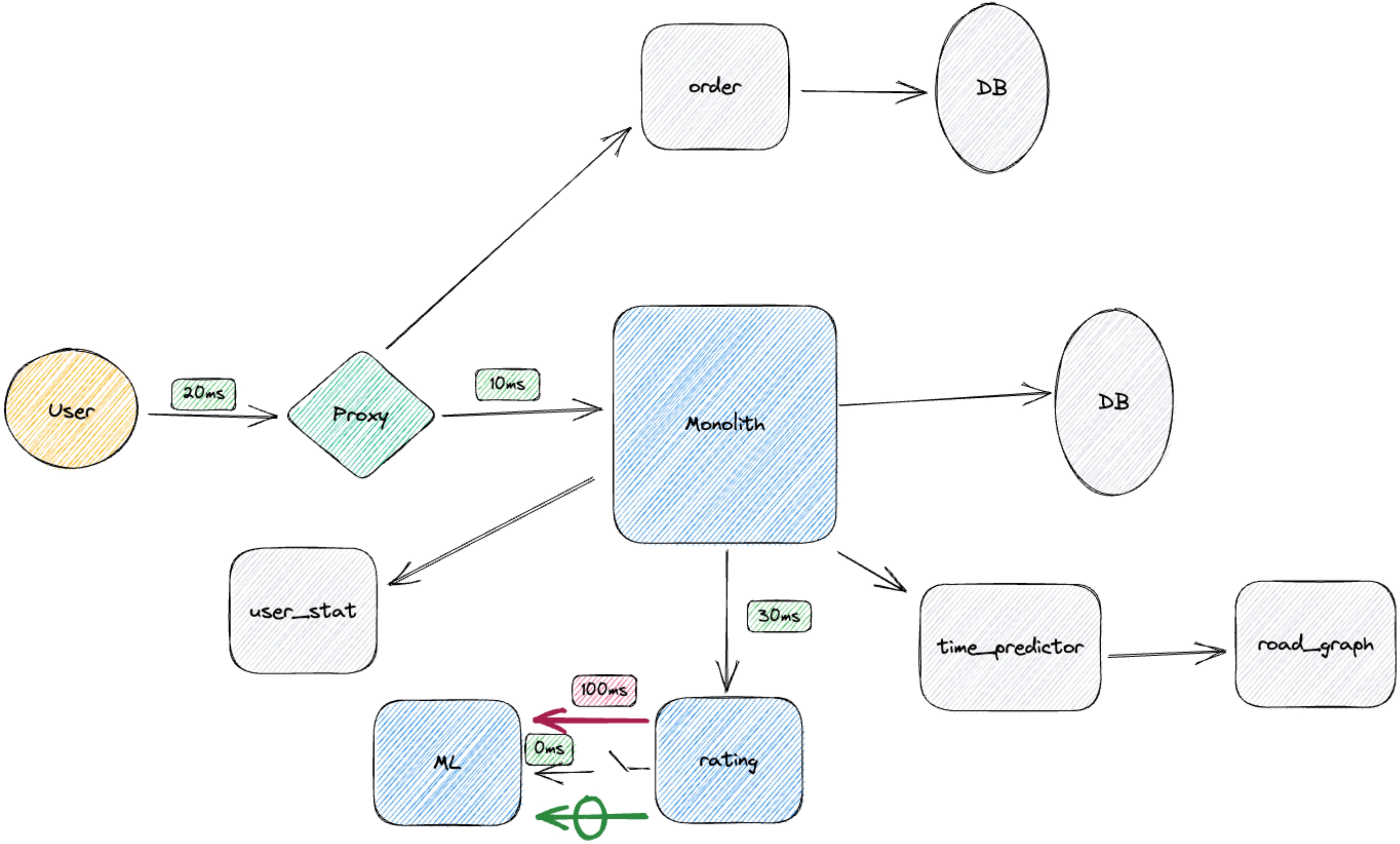

order
DB
User
20ms
Proxy
10ms
Monolith
DB
user_stat
30ms
time_predictor
road_graph
ML
100ms
rating

Problem

- ML service just going down. All requests return back 500 errors
- As result our user is waiting for +100ms just to recognize something is wrong
- And keep and keep add requests to the ML service overloading already broken service
- Every retry in this will make it worst

Solution – circle breaker

- Turn off service/dependency in case of spike / repeatable errors
- Validation of service/dependency recovery
  - By timeout
  - Network / traffic probe
  - Health checking


order
DB
User
20ms
Proxy
10ms
Monolith
DB
user_stat
30ms
time_predictor
road_graph
100ms
ML
0ms
rating

```
interface<T> CircuitBreaker {
 fun Execute(function: <T>() -> (Unit)): T
}

enum class Status {
 CLOSED,
 OPEN,
 HALF_OPEN
}
```

https://martinfowler.com/bliki/CircuitBreaker.html

https://github.com/Netflix/Hystrix/wiki/How-it-Works

```
class CircuitBreaker<T>(
 val interval: Duration
): CircuitBreaker<T> {
 val status: Status = Status.CLOSED
 val checking: Boolean = false

 val last: Duration

 override T Execute(function: <T>() -> (Unit)){
  if  (status == Status.HALF_OPEN || checking){
   throw CurcuitBreakerHalfOpenException()
  }

  if (status == Status.OPEN){
   val now = ...
   if ((now - last) < interval) {
    throw CurcuitBreakerOpenException()
   }

   // compare and exchange checking with true value, if we couldn't that means we checking it already
   if (!CAS(checking, true)) {
    throw CurcuitBreakerHalfOpenException()
   }

   last = now
   status = Status.HALF_OPEN
  }

  runCatching {
          fn()

                checking = false
             status = Status.CLOSED
        }
        .catch(ex: Exception){
             checking = false
             status = Status.OPEN
        }
    }
}
```

Summary

Out of context:

- Rich Client pattern
- Dummy/failover pattern
- Health checking (nginx, envoy, consul, custom service discovery)
- Exponential backoff and jitter
- Caching and failover caching
- Dynamic configuration (consul, custom paxos/raft based client)
- Service Mesh patterns
- Devops patterns and practices (blue green deployment, canary, etc, sidecar solutions)
- Detailed analysis of solutions in cloud environments and what can offer other frameworks (such as https://github.com/resilience4j/resilience4j)
- Integration with web frameworks

Thank you!