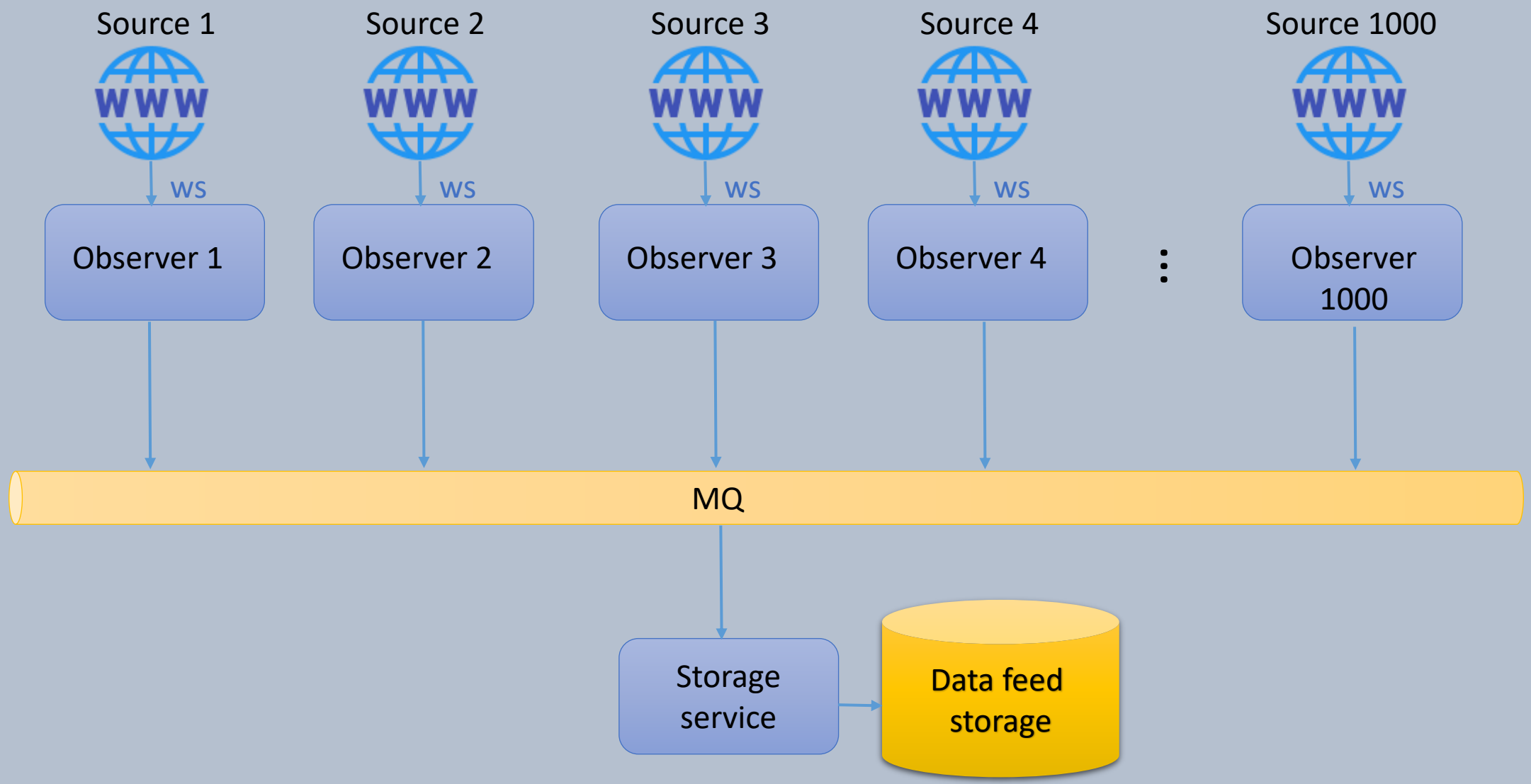
# OrchestratR

Оркестрация бесконечных задач или история о том, как сделать из очереди планировщик задач

# С чего все началось...



# Что есть бесконечная задача и где в природе они встречаются?

Бесконечная задача — это некий процесс (Job), который выполняет работу до тех пор, пока ему не скажут прекратить это. Аналогию можно провести с бесконечным циклом, который подписывается или запрашивает изменения из внешних систем и, исходя из полученных данных, выполняет полезную работу.

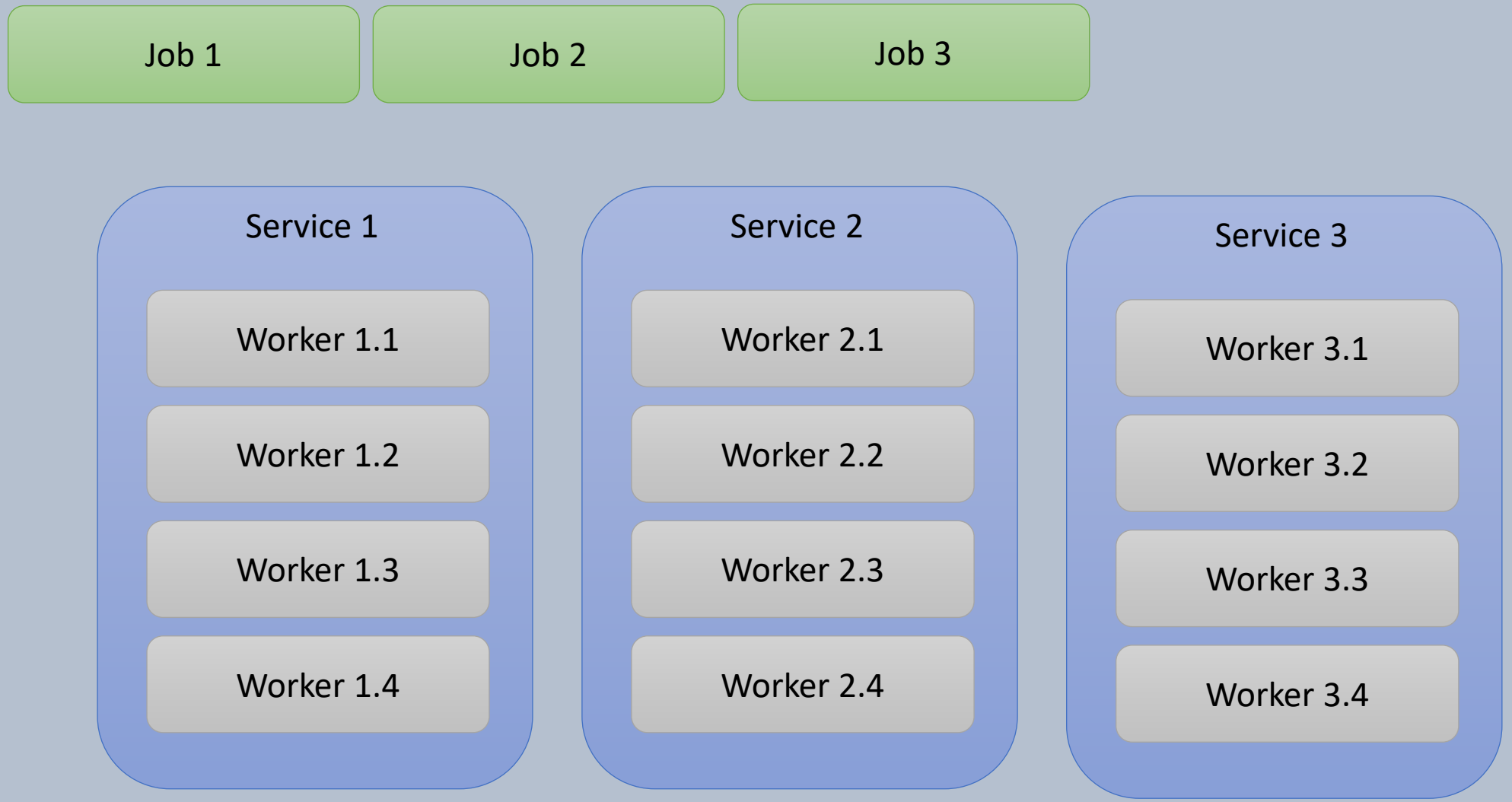
Например:

Нам необходимо следить за изменениями цен на бирже, повышением или понижением цены актива, вычислять и денормализовывать данные, отправлять полученные результаты в шину (далее по жизненному циклу).

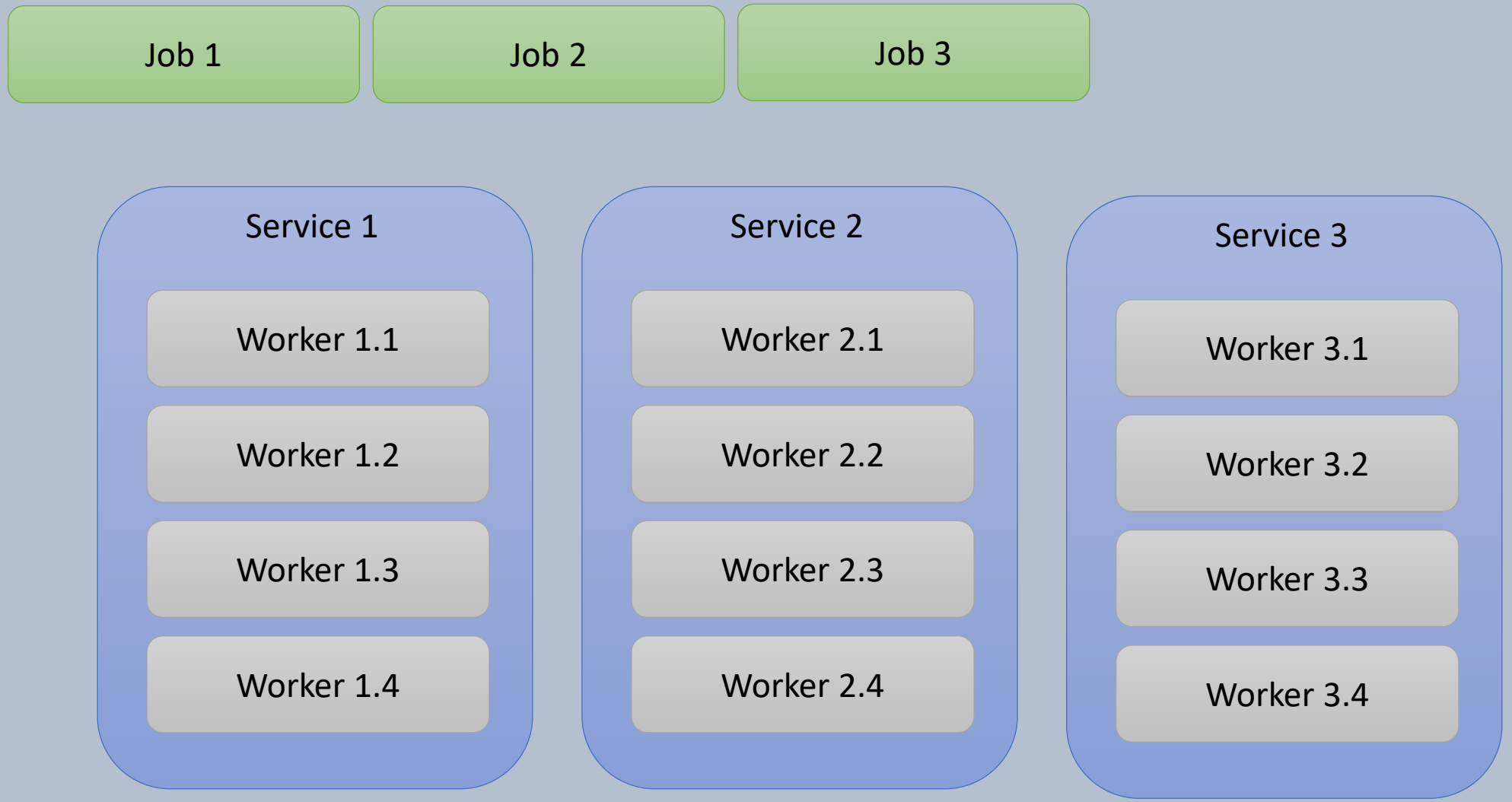
# Требования к системе:

- Задачи **должны быть управляемы**, то есть мы можем как добавить новую, так и прекратить работу существующей.
- Задачи **должны быть изолированными**, работа одной, никак не должна сказываться на работе других.
- Система **должна быть отказоустойчивой**, и горизонтально масштабируемой, мы должны иметь возможность распределить все задачи на разных серверах. Причем при отказе одного сервера, задачи, которые на нем находились, должны перераспределиться на другие, работающие сервера.
- Сервис/приложение, **должно иметь ограничение на количество задач**, которые могут быть в работе и задаваться данное значение должно через файл конфигурации или рассчитываться исходя из мощности сервера (количество ядер, RAM и т.д.), на котором сервис работает.

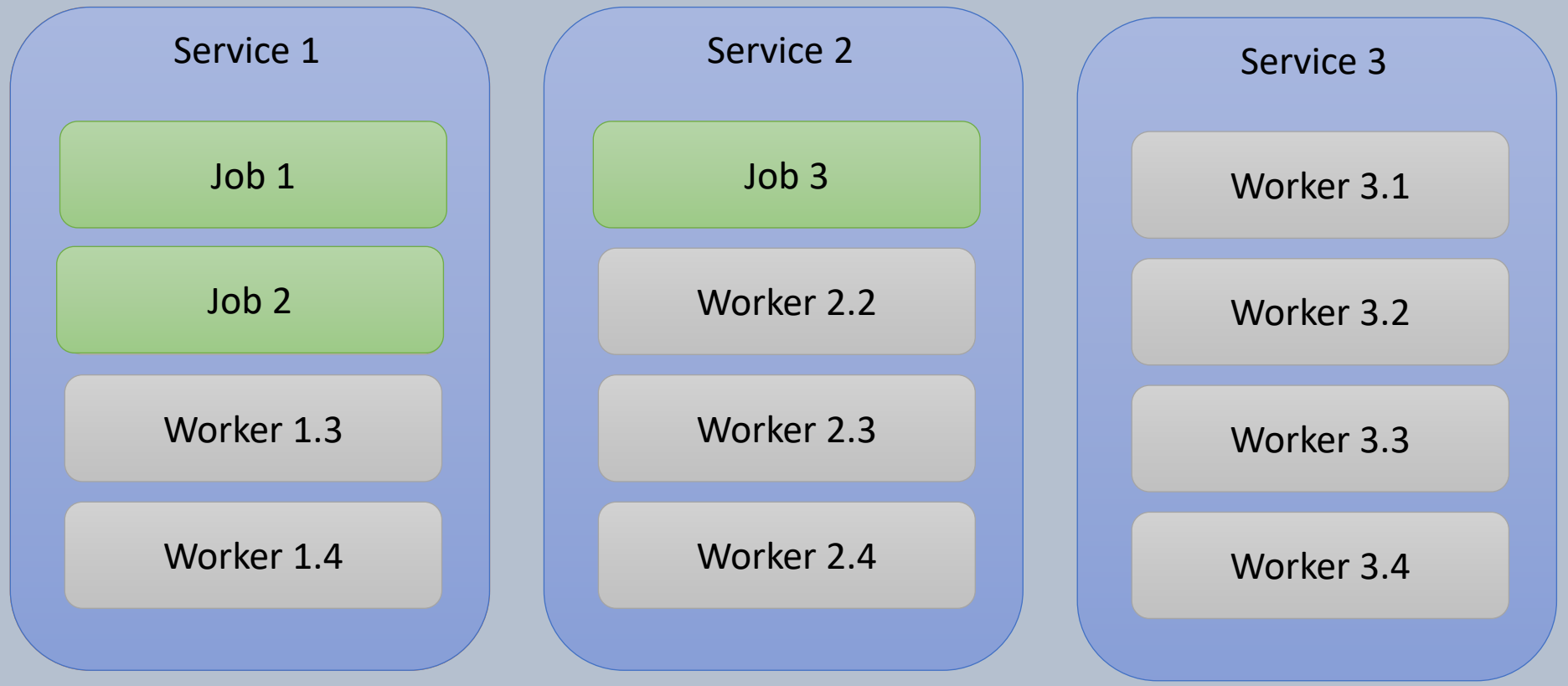
# Принцип работы системы и распределение задач между сервисами



# Принцип работы системы и распределение задач между сервисами



# Поведение задач при отказе Service 1



# Ищем готовые решения

Планировщики задач:

- Hangfire
- Quartz.net

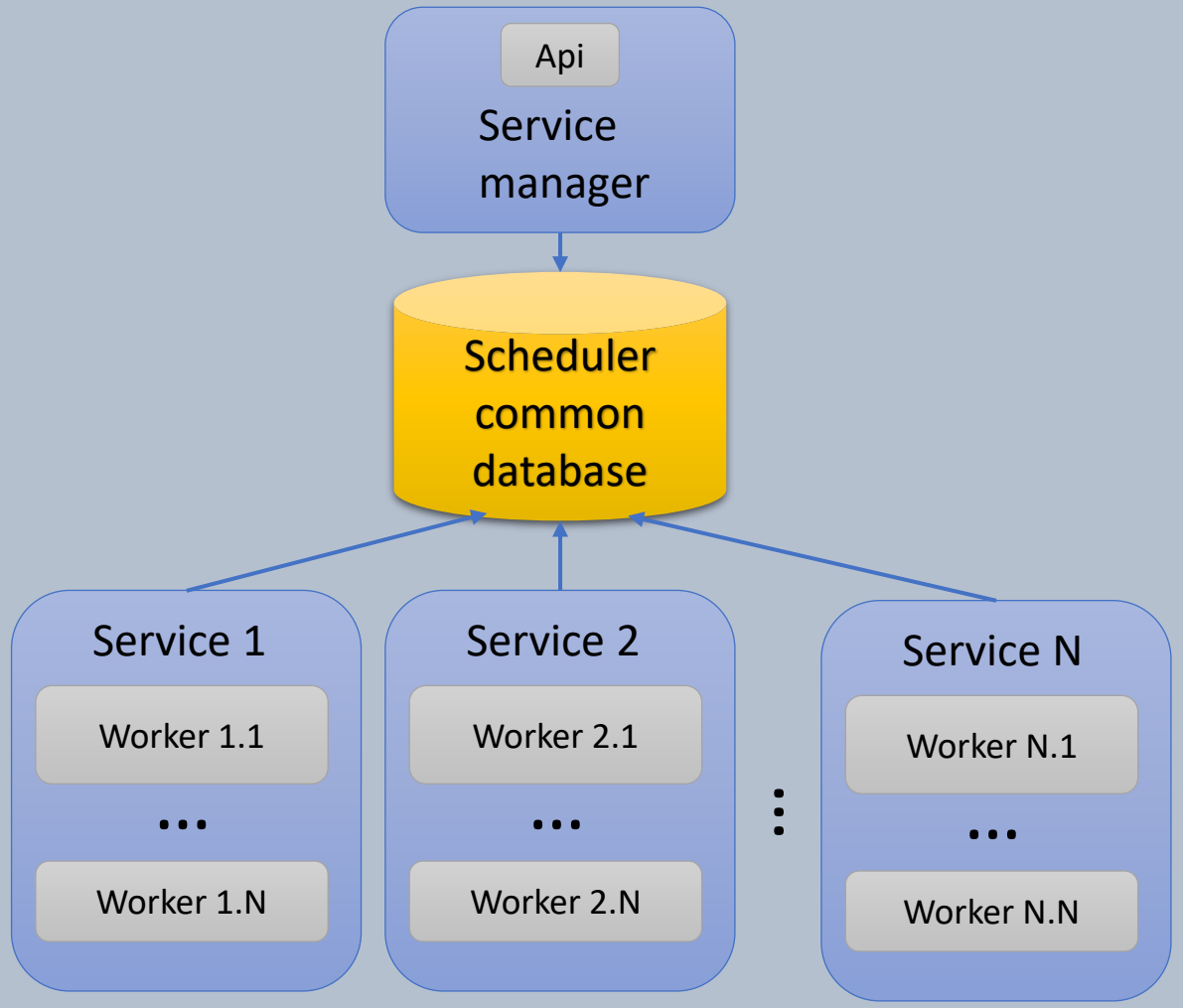
Акторы

- Orleans
- Akka.net

Nomad (Kubernetes)

OrchestratR

# Планировщики. Hangfire.




**1)** Общая сборка для менеджера и сервиса.

**2)** Высокая нагрузка на сервер. Каждый сервис опрашивает базу данных на предмет изменений

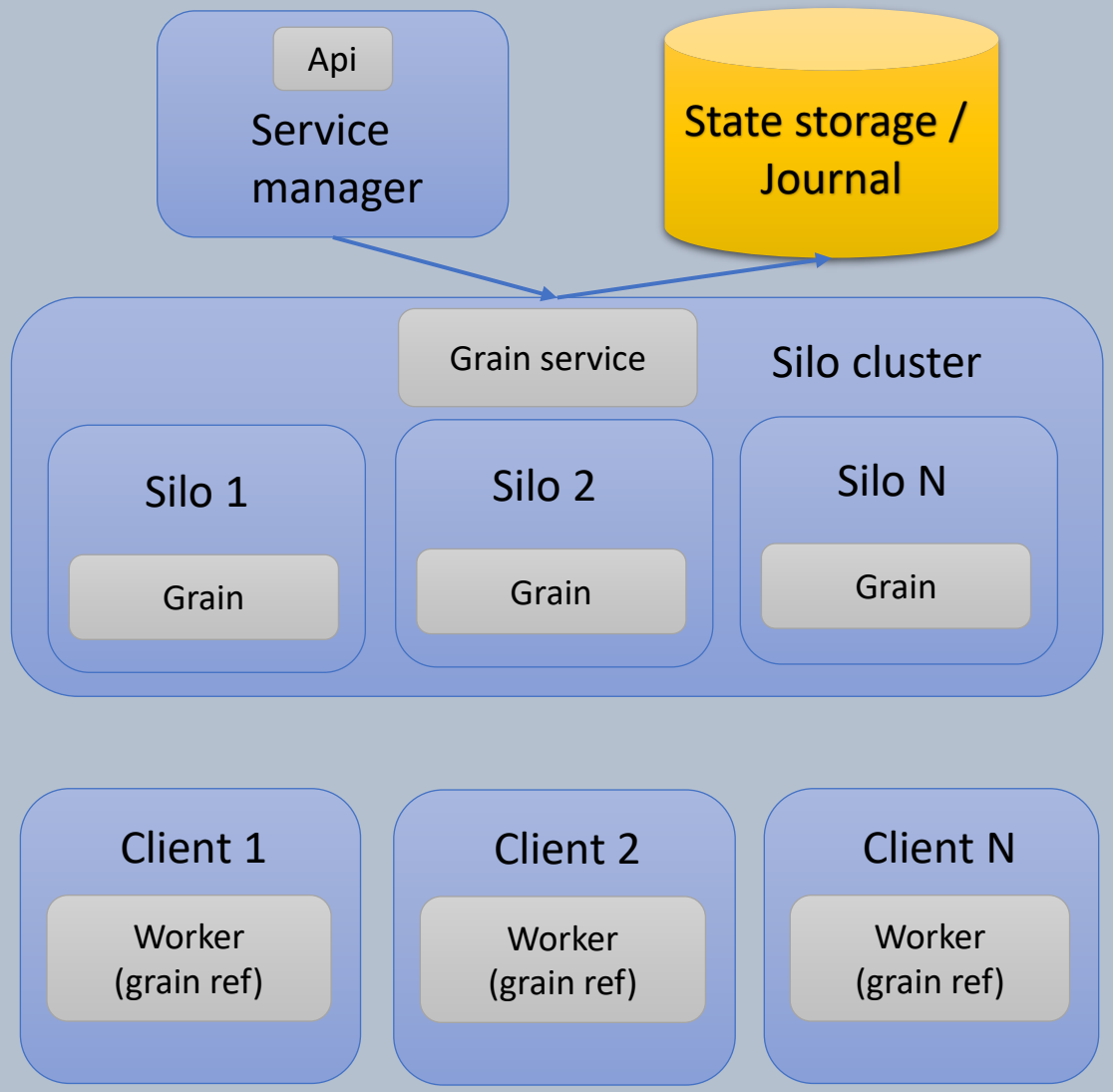
**3)** Добавление задачи в очередь возвращает числовой идентификатор и идентифицировать задачу можно только по нему. Нельзя задать свой custom-id, например, поиск по названию.

**4)** В случае ошибки, во время исполнения задачи, она автоматически будет перемещена в “default” очередь. Решается через **job-filters** или через **атрибуты**.

**5)** В случае если сервер откажет, задачи, которые на нем исполнялись, не будут перераспределены между работающими.

**6)** Отсутствие транзакционности.

# Orleans




Client может поключится к сило через:

- grain interface c->s (tcp) with response
- grain observer s<-c (tcp)
- stream (ext kafka)

- Client не знает о взятых задачах
- В Grain Service все равно необходимо реализовывать логику по распределению задач.
- Много Azure (а минус ли это?!)
- Нужно думать акторами

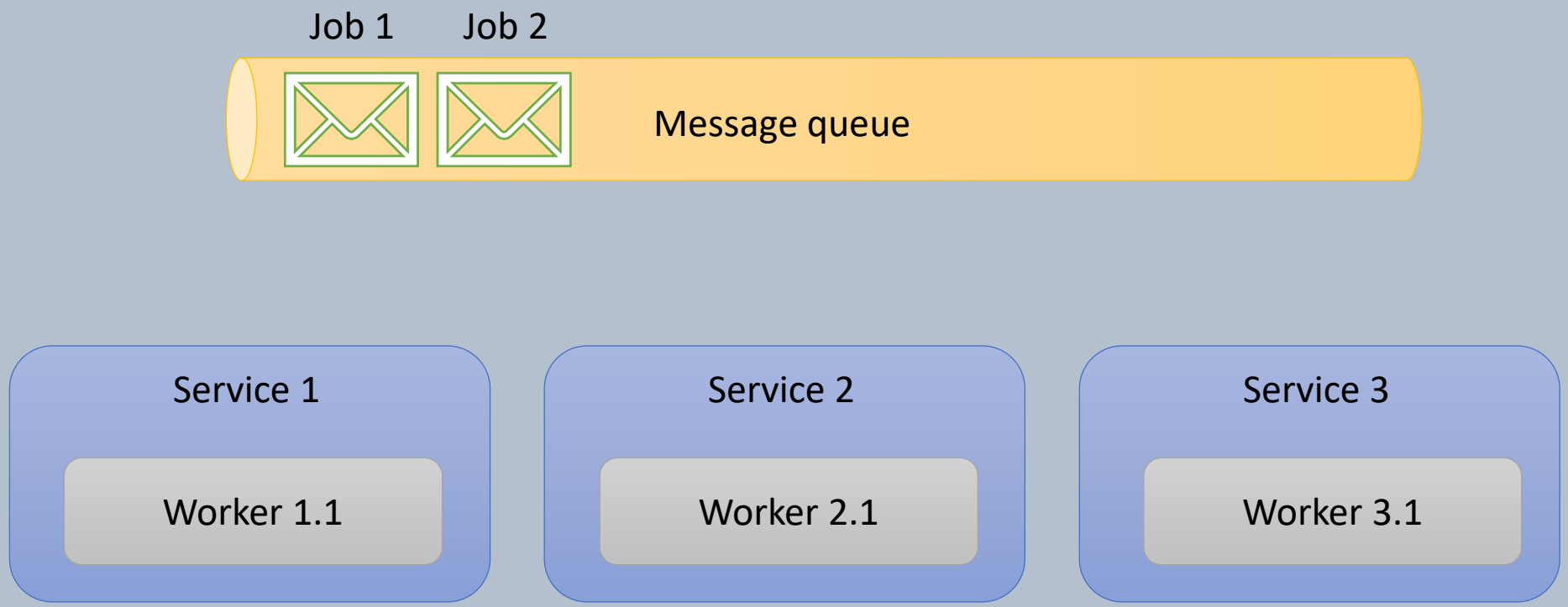
# OrchestratR. Очередь, которая думает что она планировщик.

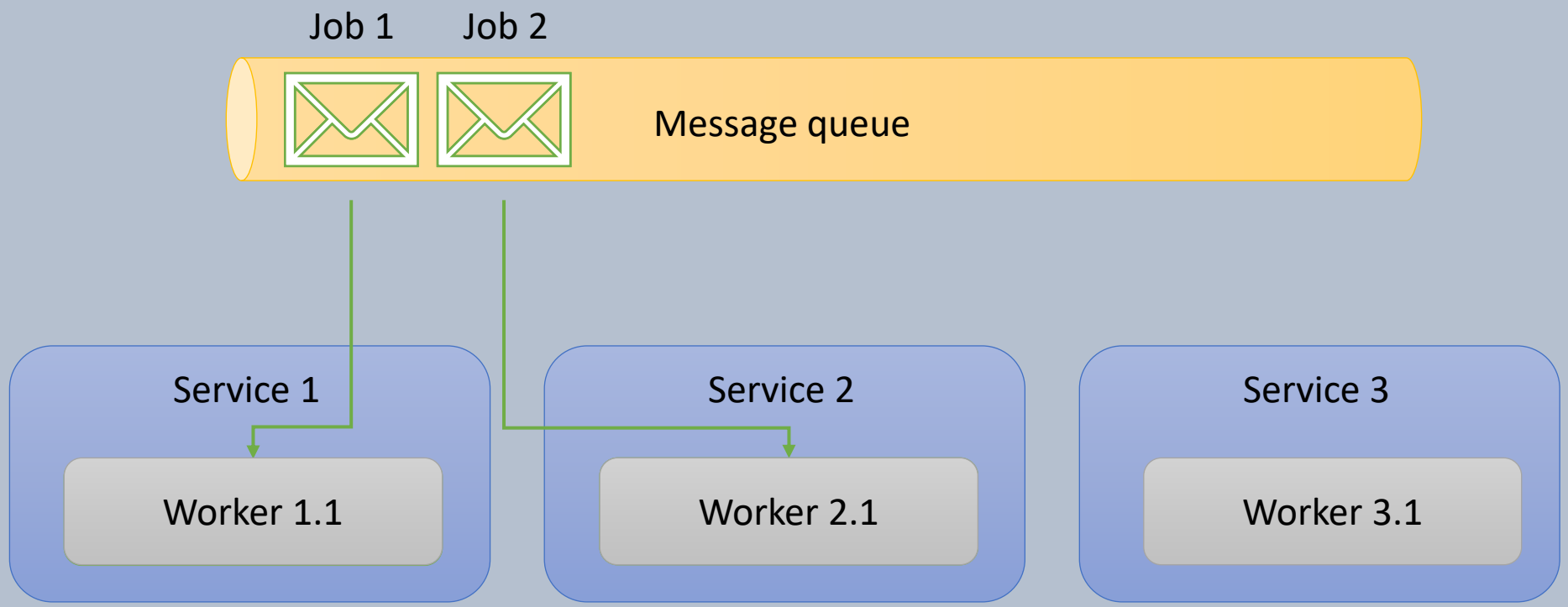
Как же сделать из очереди сообщений планировщик задач? Всё решается с помощью одной настройки - *PrefetchCount* и особенностью обработки сообщений.

- Когда сообщение попадает в очередь, оно имеет состояние Ready.
- Когда Conumer обрабатывает сообщение, оно переходит в состояние Unacked. И другой Consumer может взять следующие сообщение из очереди.
- Если в момент обработки сообщения происходит ошибка, оно помещается в **_Error** очередь.
- Если сообщение после обработки не было **acknowledged**, то оно возвращается обратно в очередь и его может прочитать любой другой Consumer.

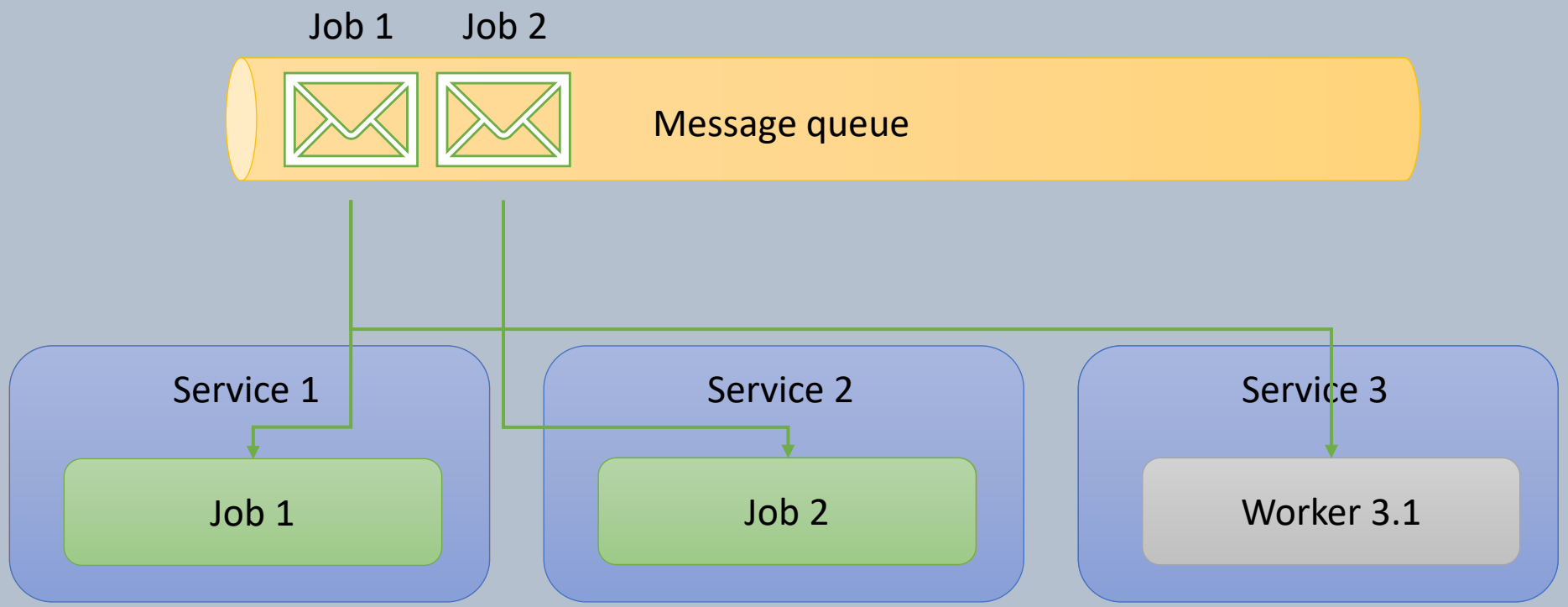
**И теперь главное** - *PrefetchCount* это количество одновременно обрабатываемых сообщений в очереди, а если сообщение никогда не будет дочитано (бесконечно обрабатывается), то его можно воспринимать как задачу в планировщике.

# Разберем на пальцах

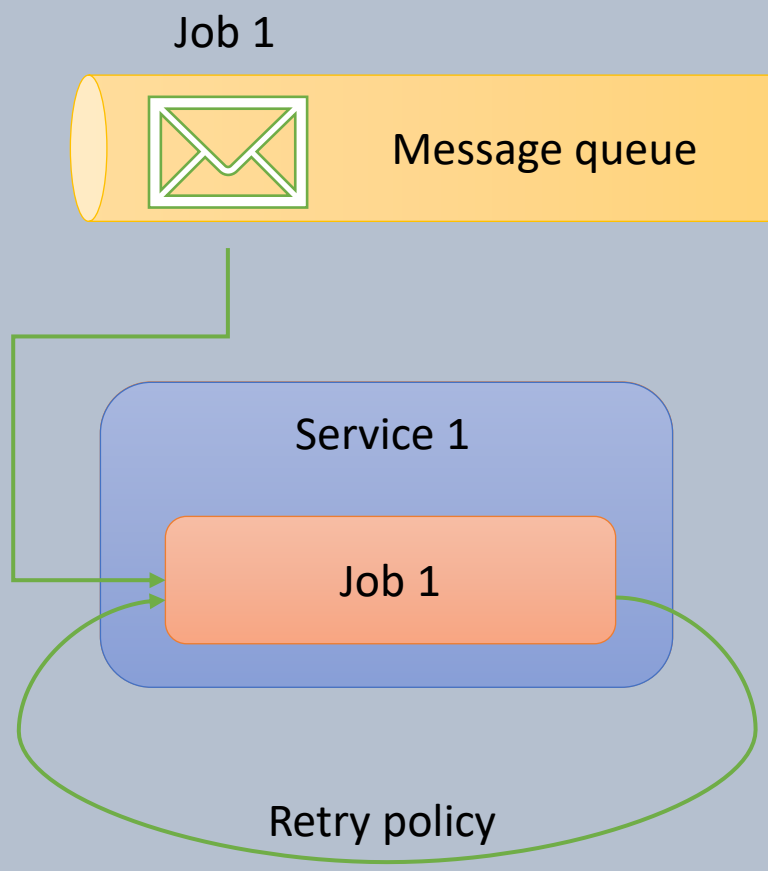



Job 1
Job 2
Message queue
Service 1
Worker 1.1
Service 2
Worker 2.1
Service 3
Worker 3.1

# Поведение очереди при отказе Service 1



# **MassTransit**: Retry policy, Redelivery



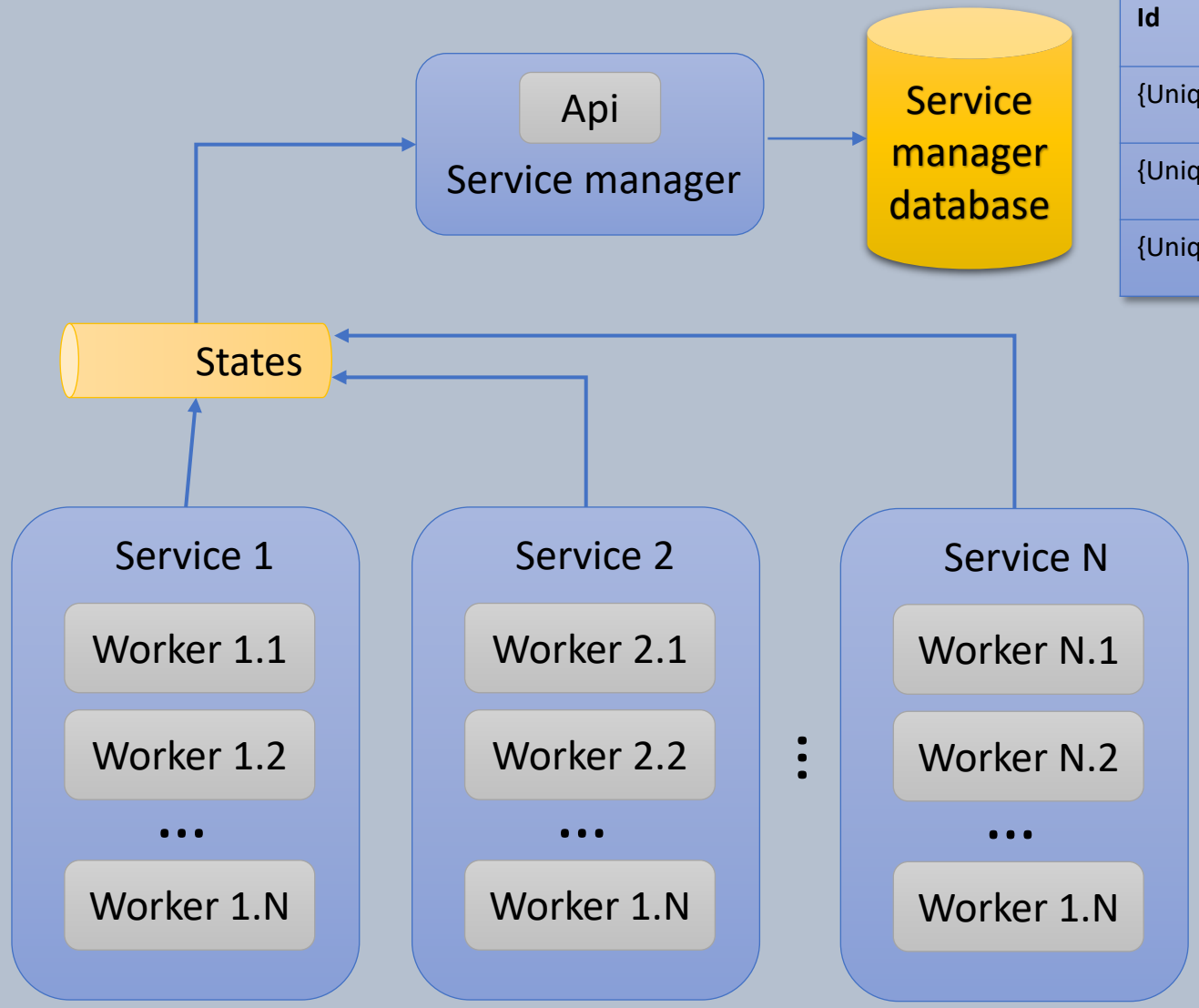

**1)** Retry policy

| **Policy** | **Description** |
|---|---|
| None | No retry |
| Immediate | Retry immediately, up to the retry limit |
| Interval | Retry after a fixed delay, up to the retry limit |
| Intervals | Retry after a delay, for each interval specified |
| Exponential | Retry after an exponentially increasing delay, up to the retry limit |
| Incremental | Retry after a steadily increasing delay, up to the retry limit |

**2)** Redelivery

**3)** Error queue

# А теперь не на пальцах



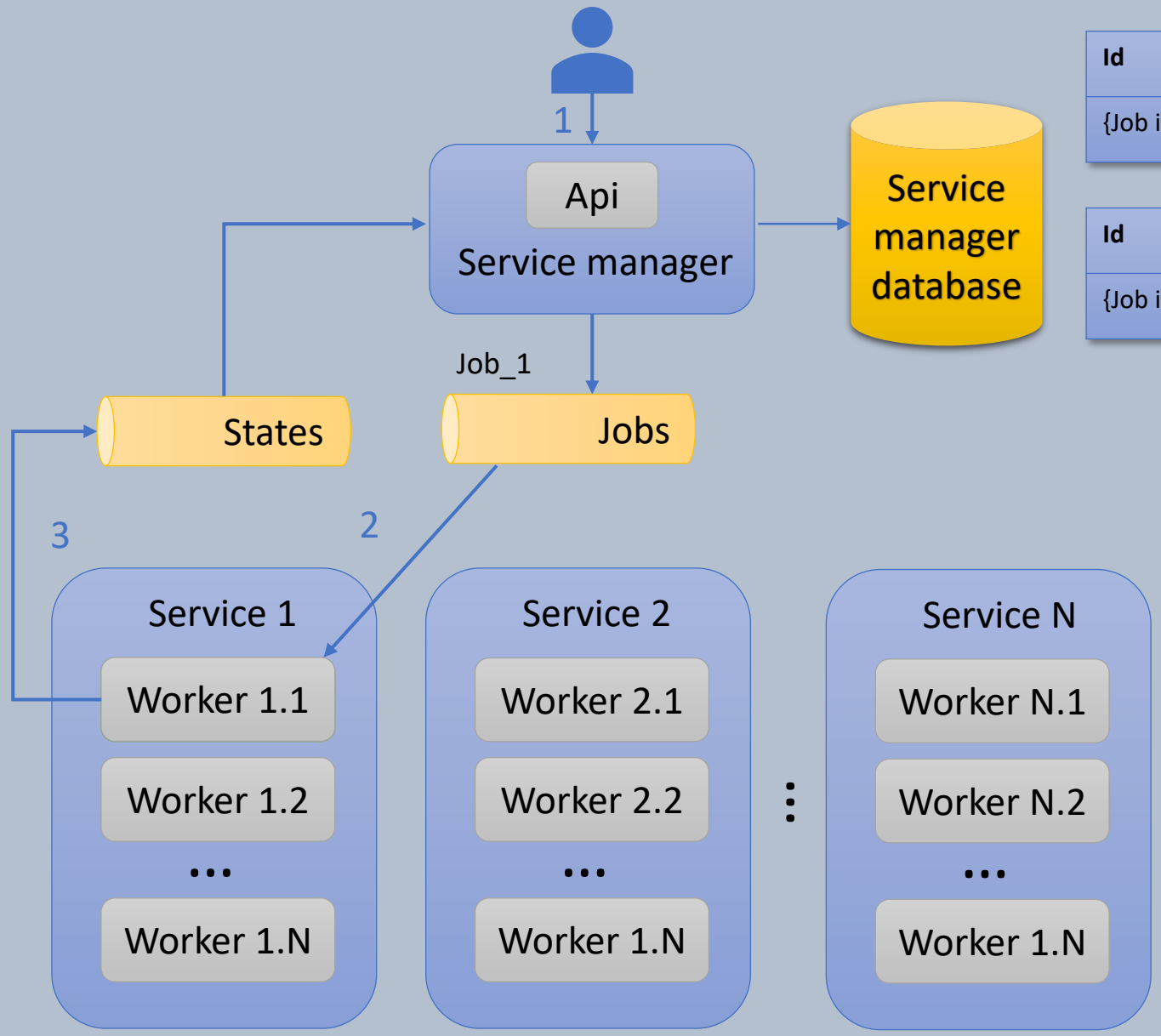

| Id | Name | WorkerCount | CreatedAt | ModifyAt | IsDeleted |
|---|---|---|---|---|---|
| {Unique id} | Service 1 | N | {some date} | null | false |
| {Unique id} | Service 2 | N | {some date} | null | false |
| {Unique id} | Service N | N | {some date} | null | false |

Каждый сервис при старте отправляет информацию менеджеру о своем существовании

Сервис отправляет информацию при завершении работы

В случае смерти сервиса (kill -9). Когда он будет перезапущен, менеджеру придет информация о вновь созданном существующим сервисе с уже существующим именем.

# Поведение системы при создании задачи




| Id | Name | CreatedAt | ModifyAt | ServiceId | Status |
|---|---|---|---|---|---|
| {Job id} | Job_1 | {created date} | null | null | Created |

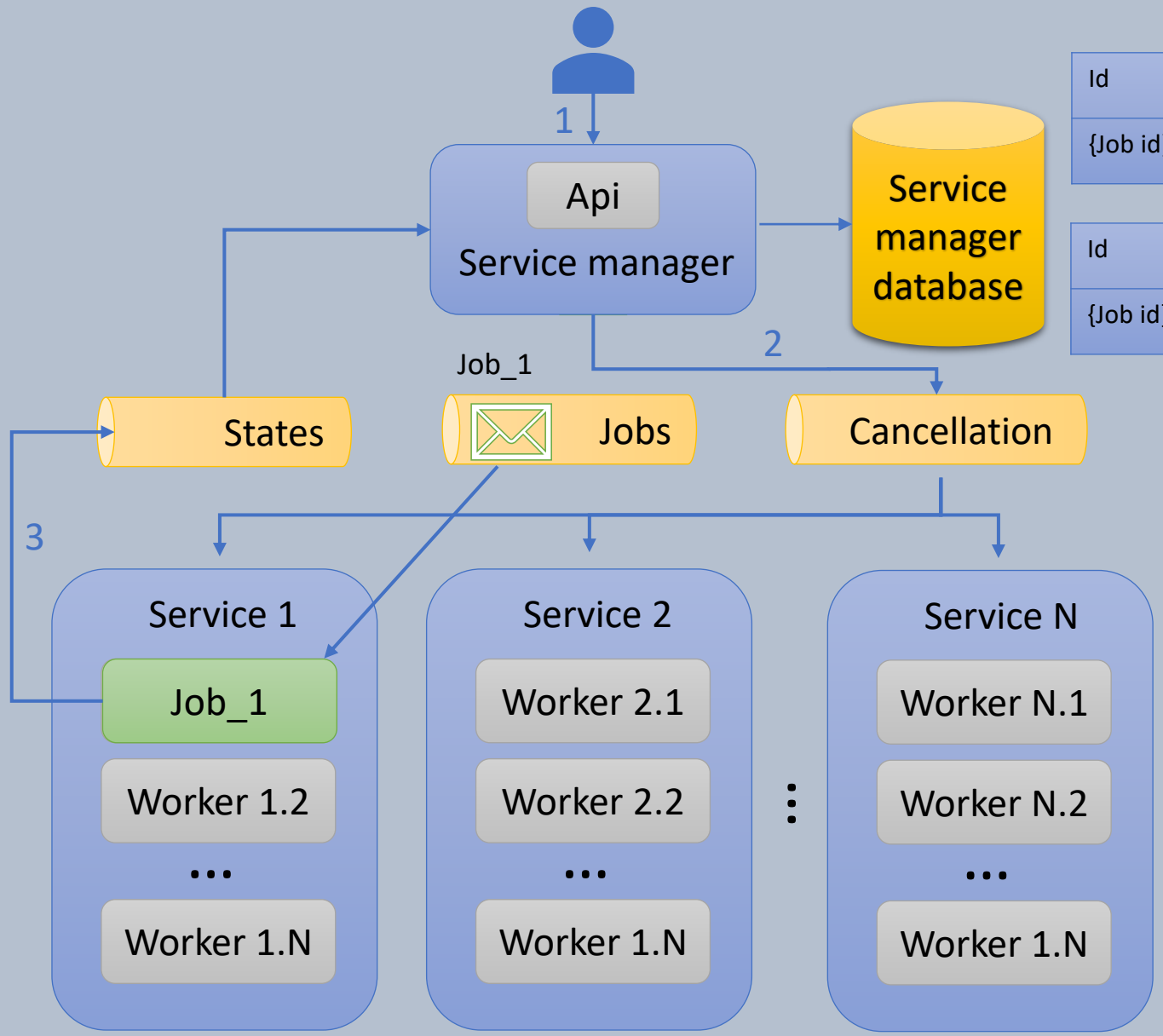
| Id | Name | CreatedAt | ModifyAt | ServiceId | Status |
|---|---|---|---|---|---|
| {Job id} | Job_1 | {created date} | {modify date} | {service id} | Processing |

**1)** Пользователь отправляет запрос на создание новой задачи – Job_1

**2)** Задача попадает в очередь и свободный Worker (Consumer) берет задачу в работу

**3)** Сервис, приняв сообщение в работу, уведомляет об этом менеджера

# Поведение системы при удалении задачи




| Id | Name | CreatedAt | ModifyAt | ServiceId | Status |
|---|---|---|---|---|---|
| {Job id} | Job_1 | {created date} | {modify date} | {service id} | OnDeleting |

| Id | Name | CreatedAt | ModifyAt | ServiceId | Status |
|---|---|---|---|---|---|
| {Job id} | Job_1 | {created date} | {modify date} | {service id} | Deleted |

**1)** Пользователь отправляет запрос на удаление задачи

**2)** Отправляются сообщения во все сервисы (Broadcast) об отмене задачи

**3)** Сервис завершает задачу и уведомляет о завершении менеджера задач

Статусная модель задачи:
- **Created**
- **Processing**
- **OnDeleting**
- **Deleted**

# Разбор поведения системы при критических сценариях

**1)** Отказала шина данных!

**2)** Полный blackout, везде нет света, все сервисы упали!

**3)** Вылетел один сервис

**4)** Отказал менеджер. В процессе пока тот бы неактивен, ломались сервера с задачами!

**5)** Попытка удалить задачу, в тот момент пока она перераспределялась!

**6)** Дедупликация сообщений в очереди!

# Разбор поведения системы при критических сценариях

**1)** Отказала шина данных!

**2)** Полный blackout, везде нет света, все сервисы упали!

**3)** Вылетел один сервис

**4)** Отказал менеджер. В процессе пока тот бы неактивен, ломались сервера с задачами!

**5)** Попытка удалить задачу, в тот момент пока она перераспределялась!

**6)** Дедупликация сообщений в очереди!

# Итог

Мы реализовали оркестратор задач, на базе механизма отправки сообщений.

- Очередь сама распределяет задачи между исполнителями, делая это событийно, а не через **polling** состояния, как это делают планировщики.
- Простое вертикальное масштабирование.
- Простое горизонтальное масштабирование.

https://github.com/dkzkv/OrchestratR

https://habr.com/ru/users/dkzkv/