

# **Развертывание ML-сервиса**



# План на неделю



Контейнеризация приложений



# План на неделю



Контейнеризация приложений



Инструмент контейнеризации Docker



# План на неделю



Контейнеризация приложений



Инструмент контейнеризации Docker



Docker Swarm и оркестрация кластера

# План на неделю



Контейнеризация приложений



Инструмент контейнеризации Docker



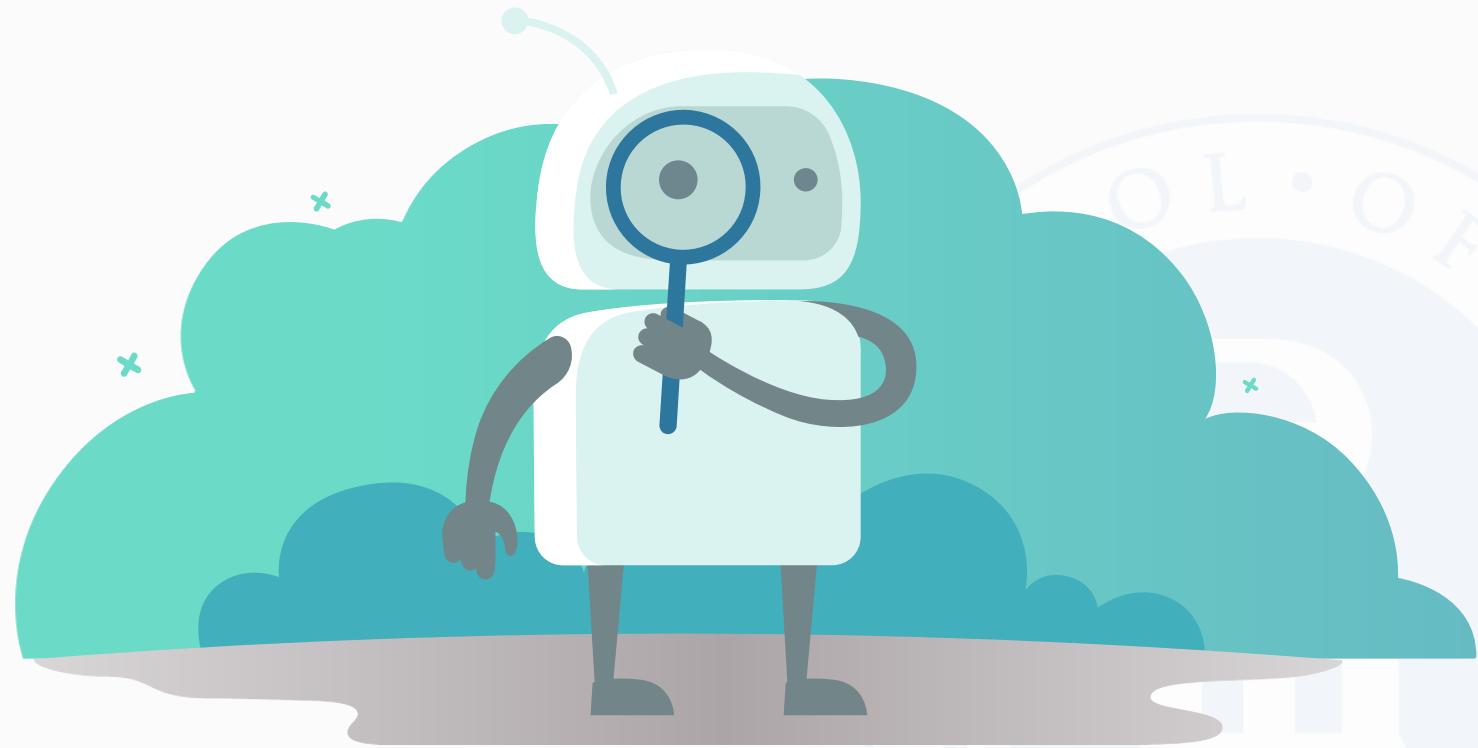
Docker Swarm и оркестрация кластера



Синхронный и асинхронный подход к проектированию  
ML-сервисов

# Зачем вообще нужны веб-сервисы?

- Обученная модель машинного обучения должна решать какую-то пользовательскую проблему



# Зачем вообще нужны веб-сервисы?



В настоящее время больше половины жителей Земли имеют [доступ в Интернет](#)

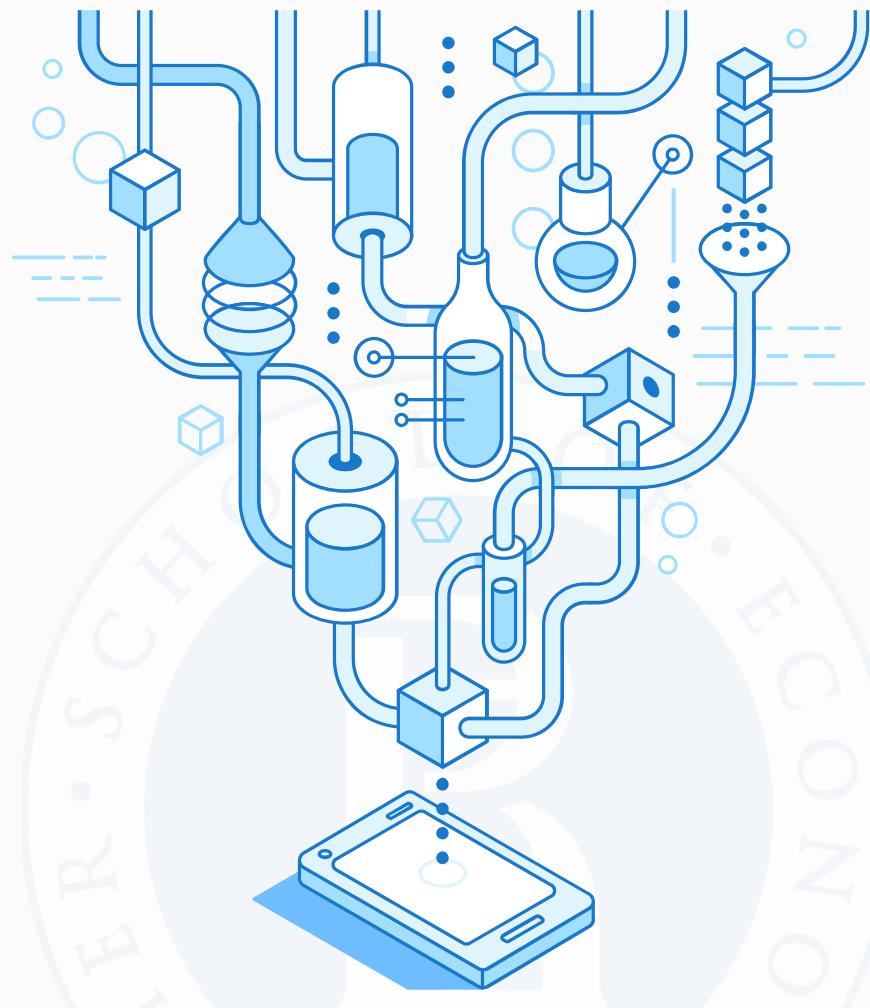


Поэтому веб-сервис — **самый быстрый и удобный способ** предоставить свою услугу на базе **искусственного интеллекта**



# Архитектура веб-сервиса

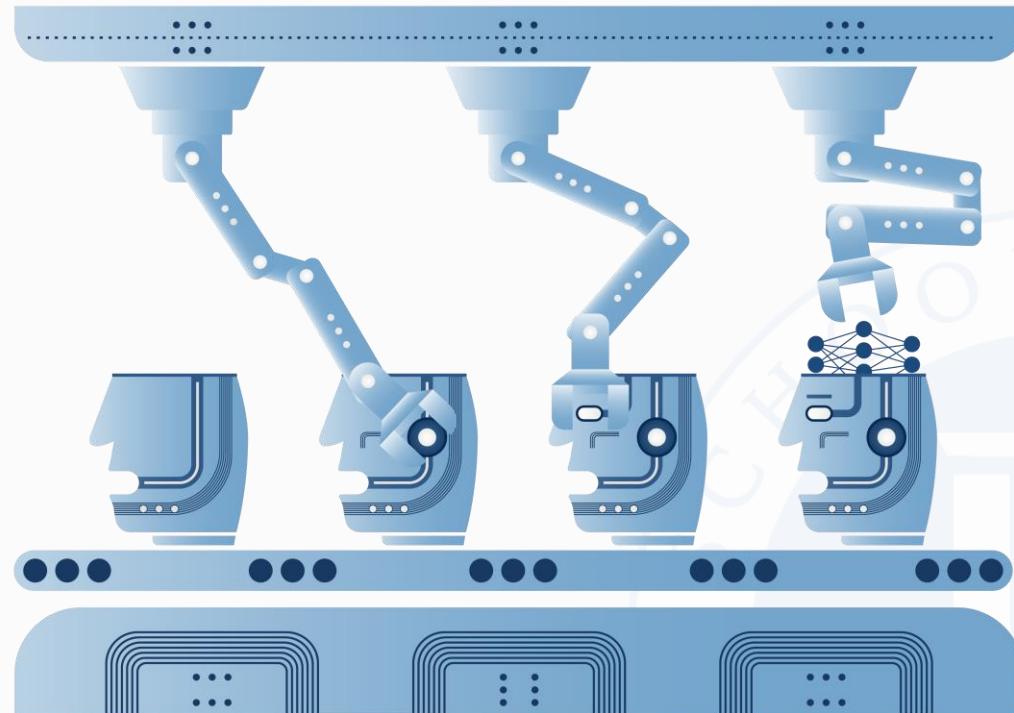
→ Чтобы сервис мог быстро обрабатывать запросы от большого числа клиентов, нужно его **правильно спроектировать**



# Развертка сервиса



Разработанный сервис необходимо уметь разворачивать на реальных серверах

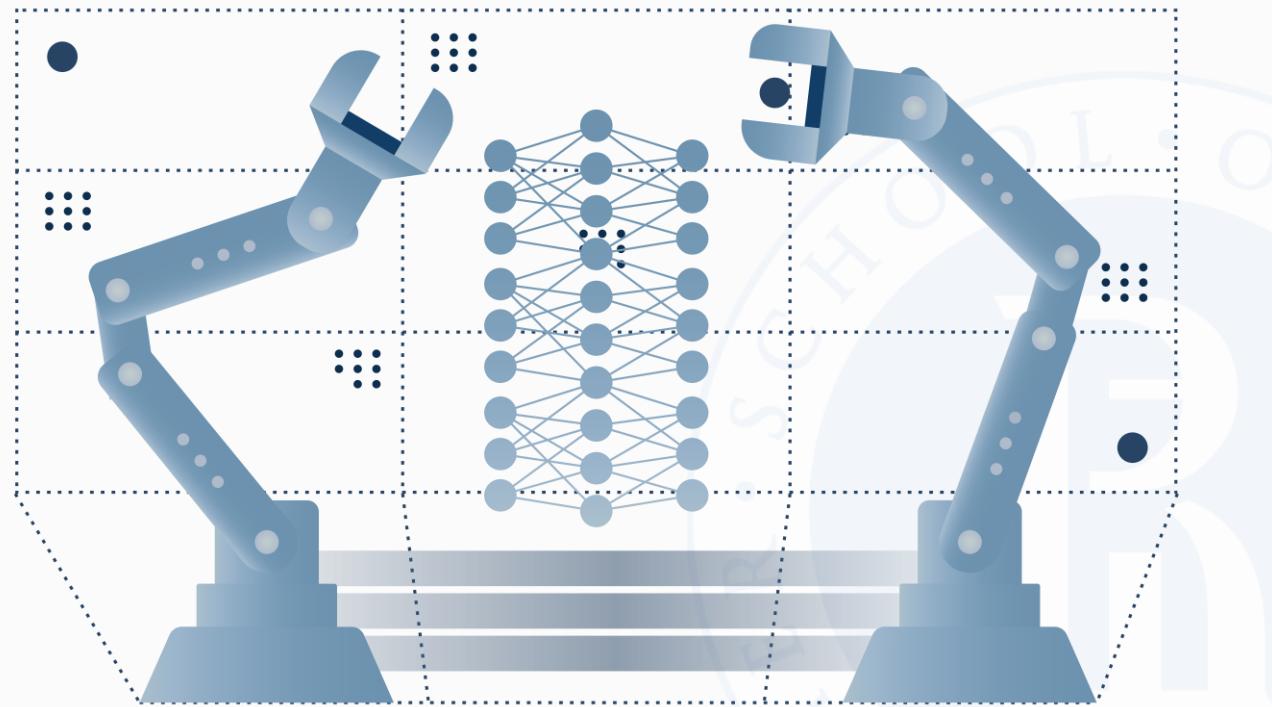


В идеале этот процесс должен быть автоматизированным, чтобы легко можно было вносить улучшения

# Что изучим на неделе



Все эти вопросы разработки и эксплуатации ML-систем мы затронем на этой неделе



# Docker



# Старые подходы к развертке приложения

- Раньше приложения развертывались **системным администратором**



- Он **вручную** устанавливал все необходимое на сервер и запускал написанный код

# Сложности со старым подходом

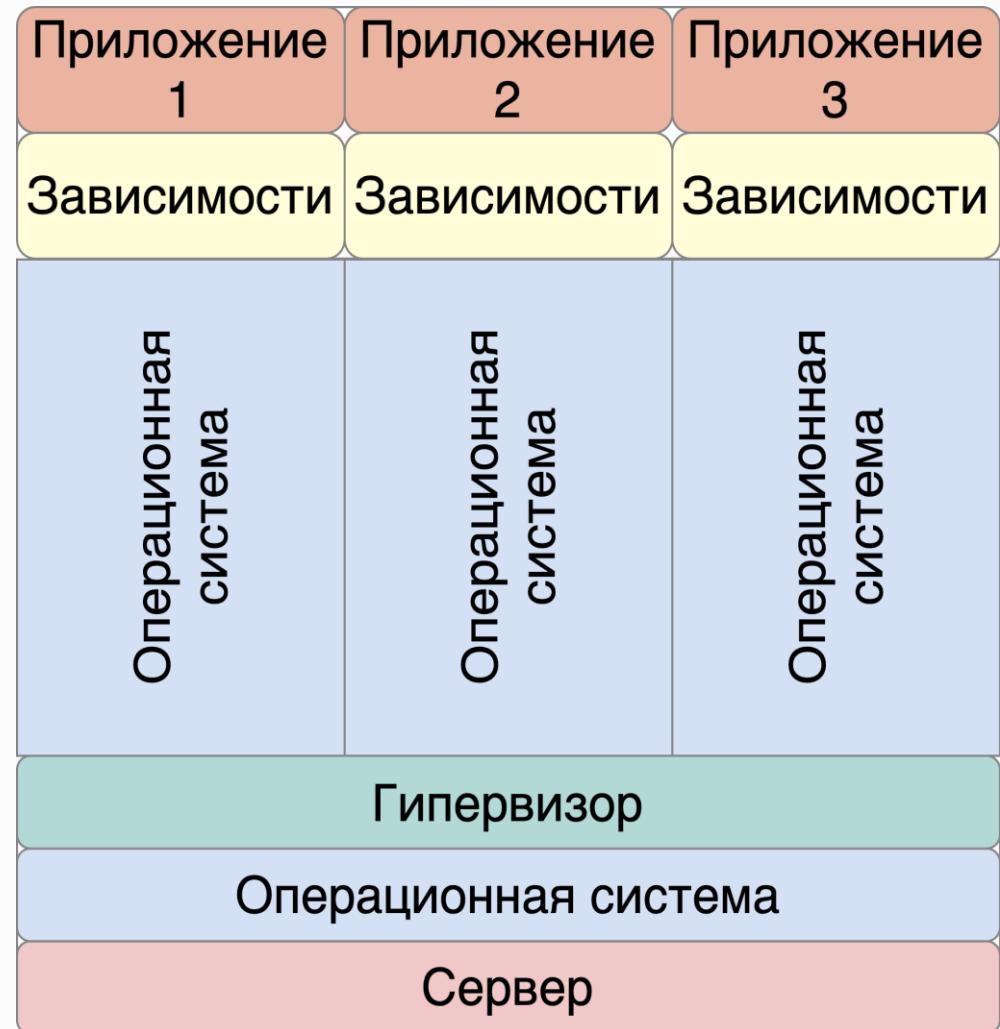
- Ручная конфигурация сервера для всех частей системы — сложная задача



- Несовпадения по версиям ОС, различные зависимости приложения и сопутствующие компоненты делали процесс развертки очень **трудозатратным**

# Виртуализация

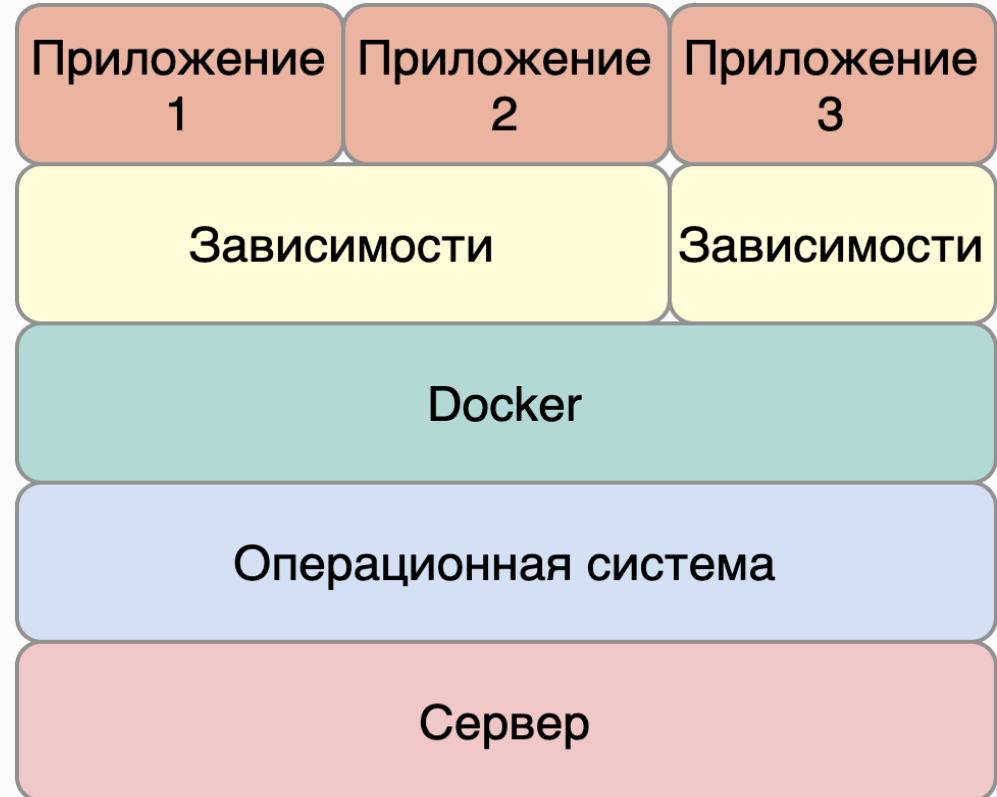
- Использование виртуальных машин упрощало процесс поставки ПО
- Однако такой подход очень требователен к ресурсам машины



# Контейнеризация

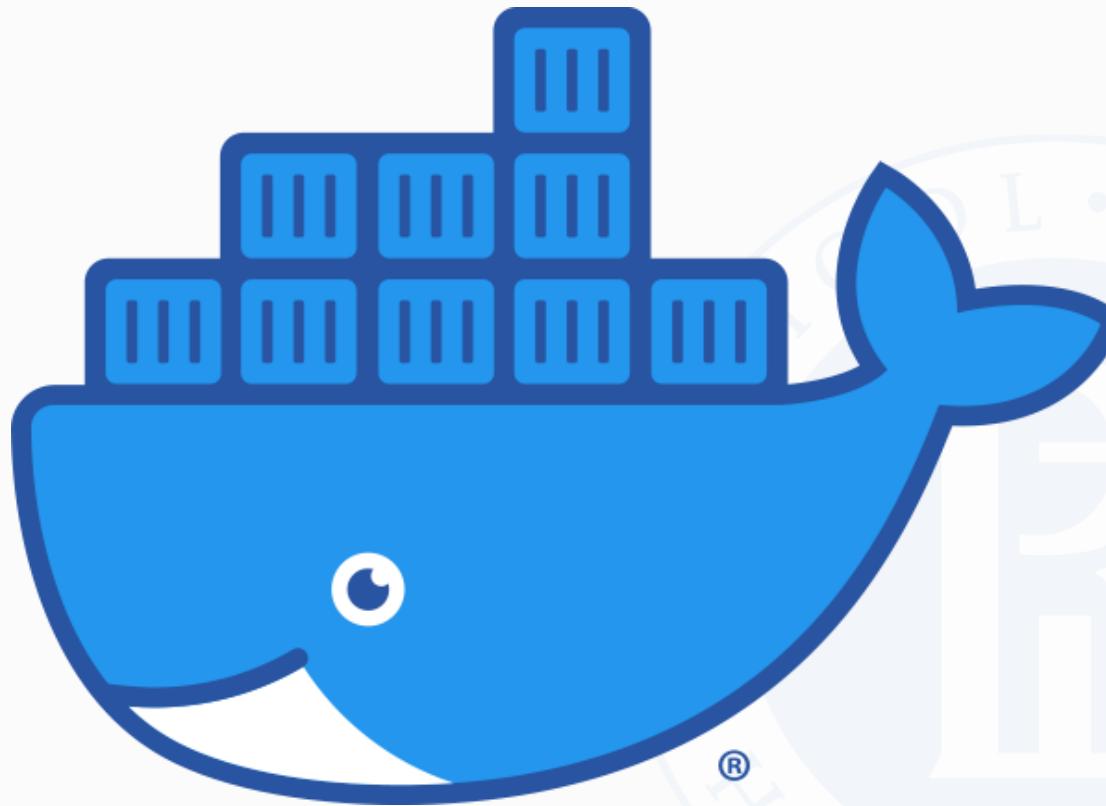
→ Контейнеры также позволяют запускать изолированные приложения

→ Однако для этого не требуется полноценная виртуальная машина



# Docker

→ Docker — наиболее популярный инструмент  
для контейнеризации приложений



# Dockerfile

→ Чтобы собрать контейнер, необходимо прописать последовательность действий для сборки в специальном файле 'Dockerfile'

```
FROM ubuntu:16.04
```

```
ENTRYPOINT [ "/bin/bash", "-c", "echo hello" ]
```

# Упакуем веб-приложение на Flask

→ Этот Dockerfile собирает простое приложение на Flask — веб-сервере на Python

```
FROM ubuntu:16.04
```

```
RUN apt-get install python3 python3-pip
```

```
RUN pip install Flask
```

```
COPY server.py /app/server.py
```

```
ENTRYPOINT [ "python3", "/app/server.py" ]
```

# Упакуем веб-приложение на Flask

→ Указываем базовый образ — Ubuntu 16.04

```
FROM ubuntu:16.04
```

```
RUN apt-get install python3 python3-pip
```

```
RUN pip install Flask
```

```
COPY server.py /app/server.py
```

```
ENTRYPOINT [ "python3", "/app/server.py" ]
```

# Упакуем веб-приложение на Flask

→ Базовый образ совсем [пустой](#). Python необходимо установить [самостоятельно](#)

```
FROM ubuntu:16.04

RUN apt-get install python3 python3-pip

RUN pip install Flask

COPY server.py /app/server.py

ENTRYPOINT [ "python3", "/app/server.py" ]
```

# Упакуем веб-приложение на Flask

→ Устанавливаем библиотеку Flask

```
FROM ubuntu:16.04
```

```
RUN apt-get install python3 python3-pip
```

```
RUN pip install Flask
```

```
COPY server.py /app/server.py
```

```
ENTRYPOINT [ "python3", "/app/server.py" ]
```

# Упакуем веб-приложение на Flask

→ Копируем написанный нами сервер на Flask

```
FROM ubuntu:16.04
```

```
RUN apt-get install python3 python3-pip
```

```
RUN pip install Flask
```

```
COPY server.py /app/server.py
```

```
ENTRYPOINT [ "python3", "/app/server.py" ]
```

# Упакуем веб-приложение на Flask

- Сообщаем, что при запуске необходимо запустить наш сервер

```
FROM ubuntu:16.04
```

```
RUN apt-get install python3 python3-pip
```

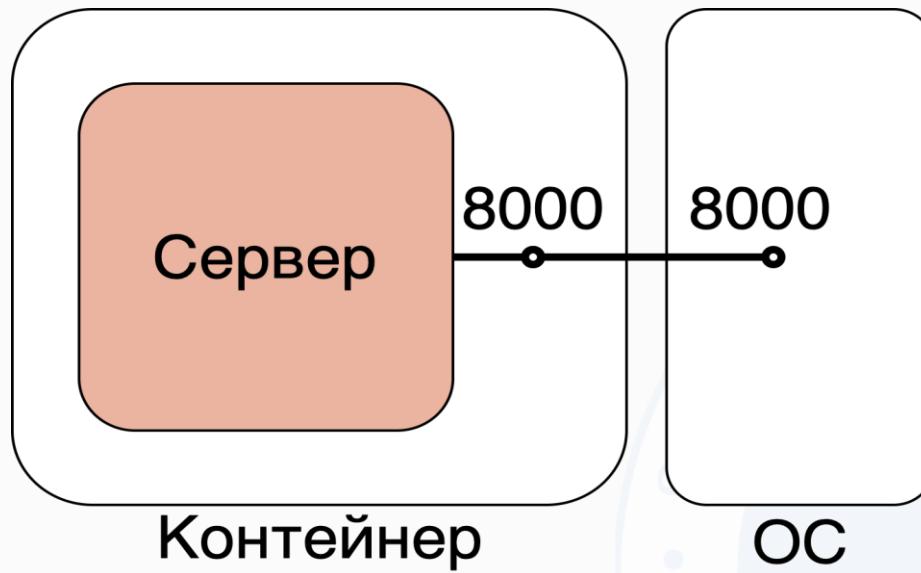
```
RUN pip install Flask
```

```
COPY server.py /app/server.py
```

```
ENTRYPOINT [ "python3", "/app/server.py" ]
```

# Запуск контейнера

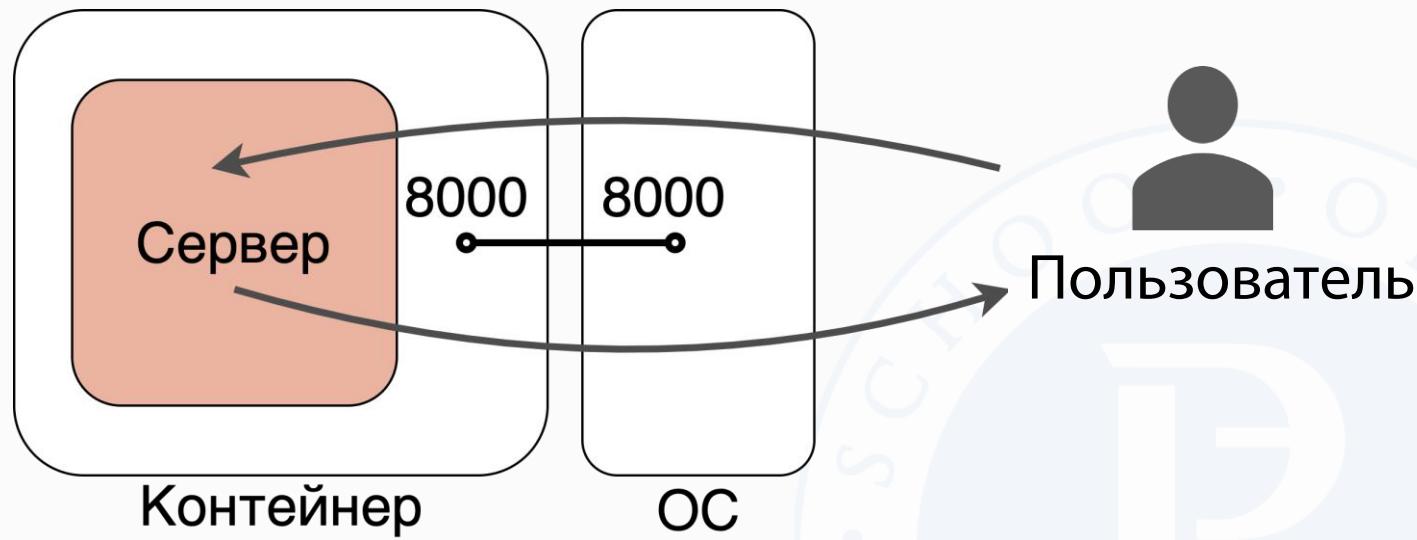
→ Запущенный контейнер по умолчанию **полностью изолирован**



→ Чтобы коммуницировать с приложением в контейнере, необходимо специально **соединить** его с основной ОС

# Запуск контейнера

- После соединения все сетевые запросы к ОС по указанному порту будут **направляться внутрь** контейнера



- Таким образом пользователи смогут **пользоваться** нашим контейнеризированным приложением

# Плюсы контейнеризации



Изолированные зависимости

# Плюсы контейнеризации

- Изолированные зависимости
- Ресурсоэффективность

# Плюсы контейнеризации

- Изолированные зависимости
- Ресурсоэффективность
- Воспроизводимость



# Плюсы контейнеризации

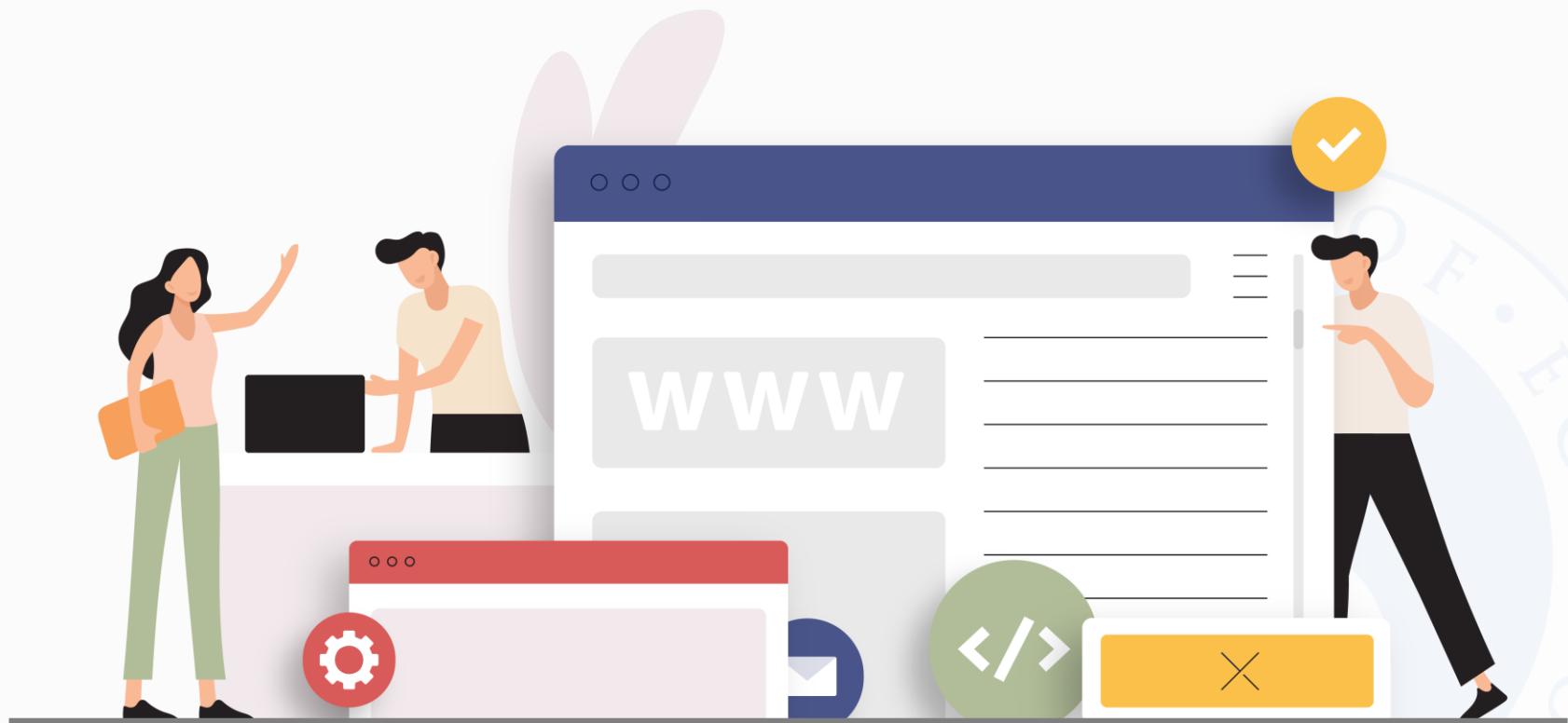
- Изолированные зависимости
- Ресурсоэффективность
- Воспроизводимость
- Защита от аварий



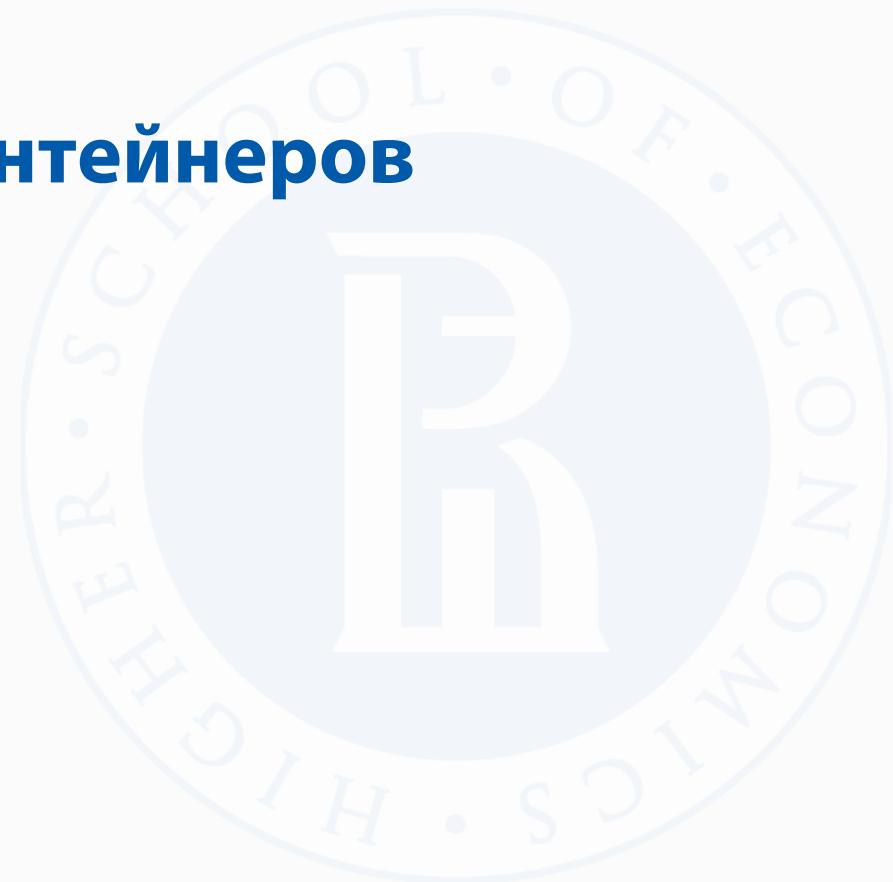
# Практическая сторона вопроса



В следующем видео рассмотрим, как на практике собирать образы и как их потом запускать

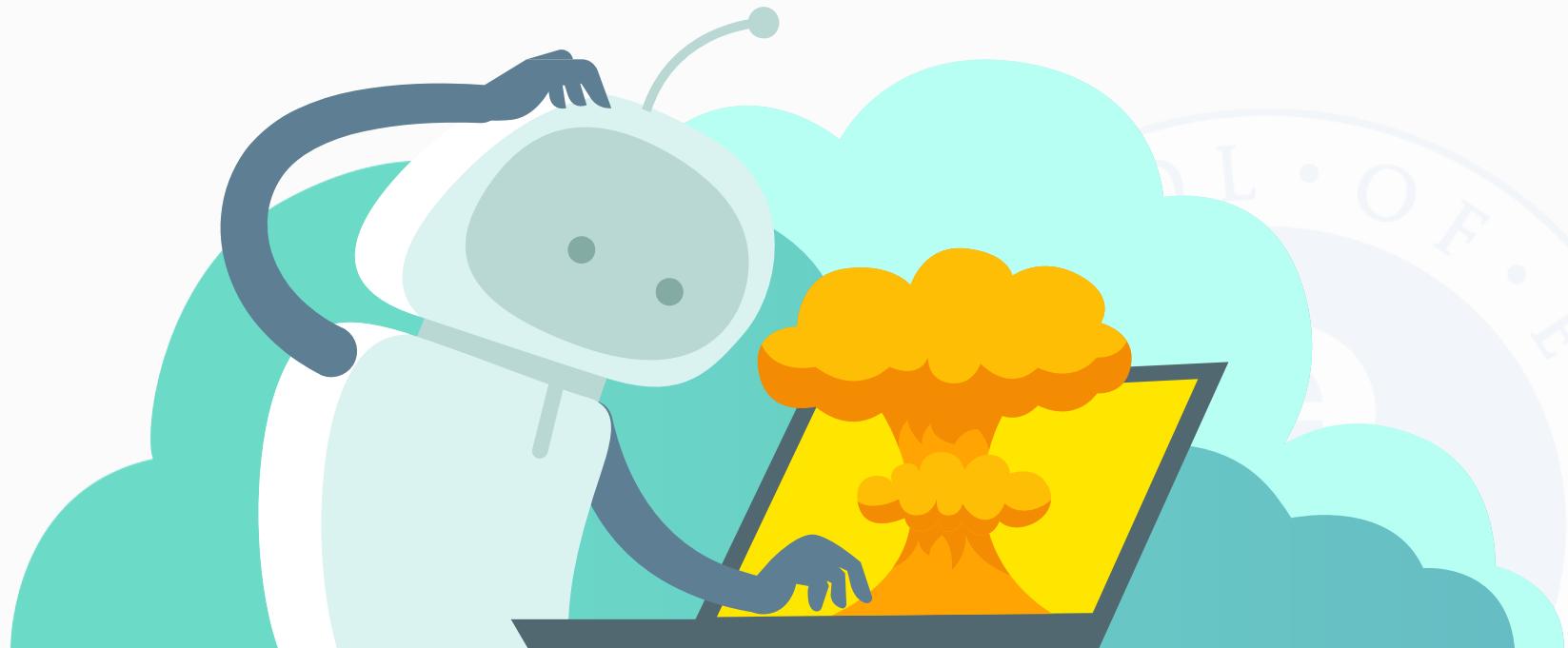


# **Оркестрация контейнеров**



# Масштабирование

→ Одной машины может не хватить для эффективной обработки запросов от большого числа пользователей



# Масштабирование



Чтобы выдерживать нагрузки, необходимо масштабировать сервис — увеличивать количество серверов для обработки



# Docker Compose

→ Docker Compose — это инструмент для управления несколькими контейнерами



docker  
Compose

# Docker Compose

- Специальный файл docker-Compose.yaml содержит конфигурацию контейнеров, которые необходимо запустить

```
version: 3.7
services:
  application:
    build:
      context: ./source-code
    ports:
      - 8000:8000

  database:
    image: mongo:4
    ports:
      - 27017:27017
```

# Docker Compose

→ Необходимо указать версию docker Compose для этой конфигурации

```
version: 3.7
services:
  application:
    build:
      context: ./source-code
    ports:
      - 8000:8000

  database:
    image: mongo:4
    ports:
      - 27017:27017
```

# Docker Compose

→ В секции `services` необходимо перечислить сервисы, которые мы хотим запускать. В данномм случае — это **приложение и база данных**

```
version: 3.7
services:
  application:
    build:
      context: ./source-code
    ports:
      - 8000:8000

  database:
    image: mongo:4
    ports:
      - 27017:27017
```

# Docker Compose

→ Для каждого сервиса необходимо указать, [откуда брать контейнер](#). Для приложения мы указали, что его нужно собрать из директории [./source-code](#)

```
version: 3.7
services:
  application:
    build:
      context: ./source-code
    ports:
      - 8000:8000

  database:
    image: mongo:4
    ports:
      - 27017:27017
```

# Docker Compose

→ Для базы данных мы указали уже готовый образ с MongoDB

```
version: 3.7
services:
  application:
    build:
      context: ./source-code
    ports:
      - 8000:8000

  database:
    image: mongo:4
    ports:
      - 27017:27017
```

# Docker Compose



Мы также можем сразу указать порты, которые необходимо пробросить. Для приложения — это 8000, для базы данных — 27017

```
version: 3.7
services:
  application:
    build:
      context: ./source-code
    ports:
      - 8000:8000

  database:
    image: mongo:4
    ports:
      - 27017:27017
```

# Docker Compose



Для запуска написанной конфигурации достаточно запустить [одну команду](#)

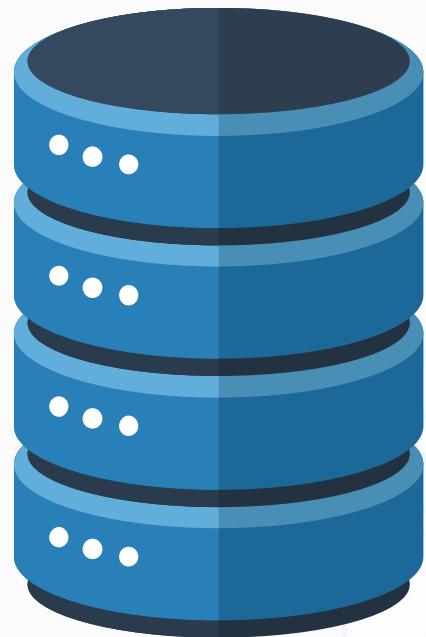
```
docker-compose up
```



[Команда запустит](#) все контейнеры на текущей машине с указанной конфигурацией

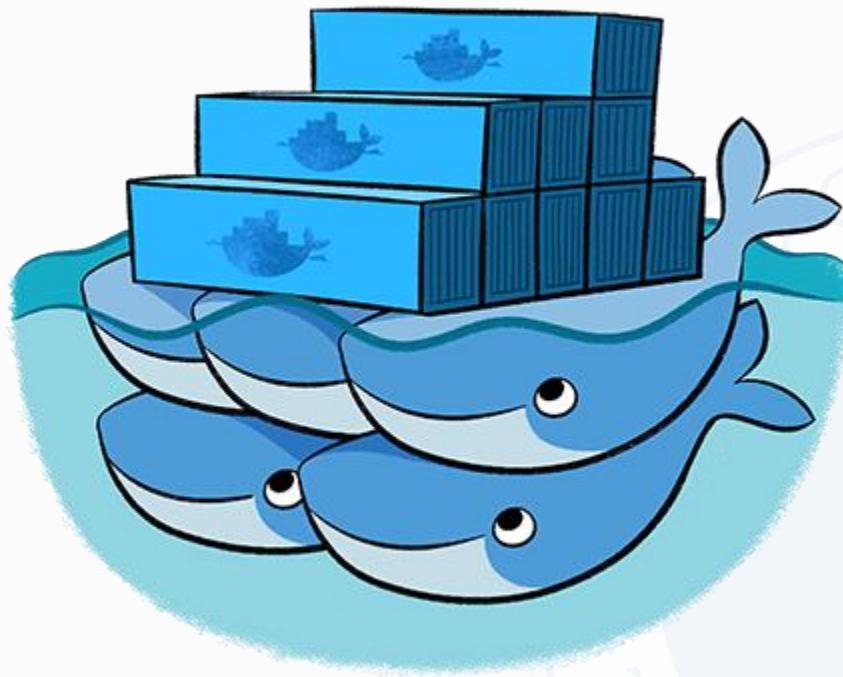
# Docker Compose

- Docker Compose сам по себе умеет запускать контейнеры только на [одной](#) машине



- Чтобы управлять контейнерами сразу на нескольких машинах, необходим более продвинутый инструмент

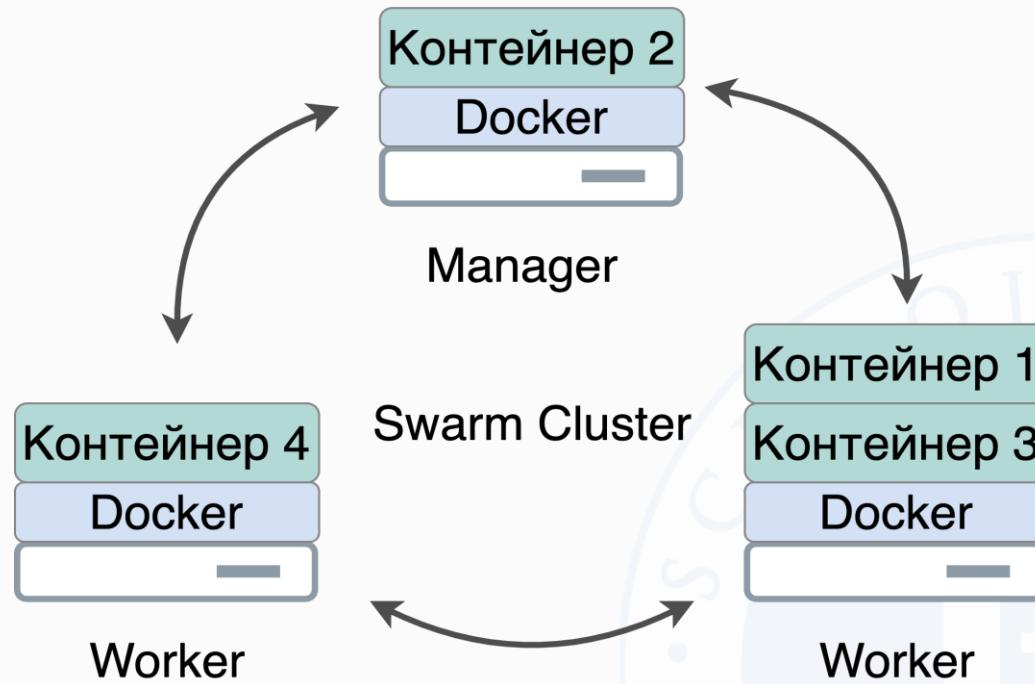
# Docker Swarm



[Hub.docker.com](https://Hub.docker.com)

# Docker Swarm

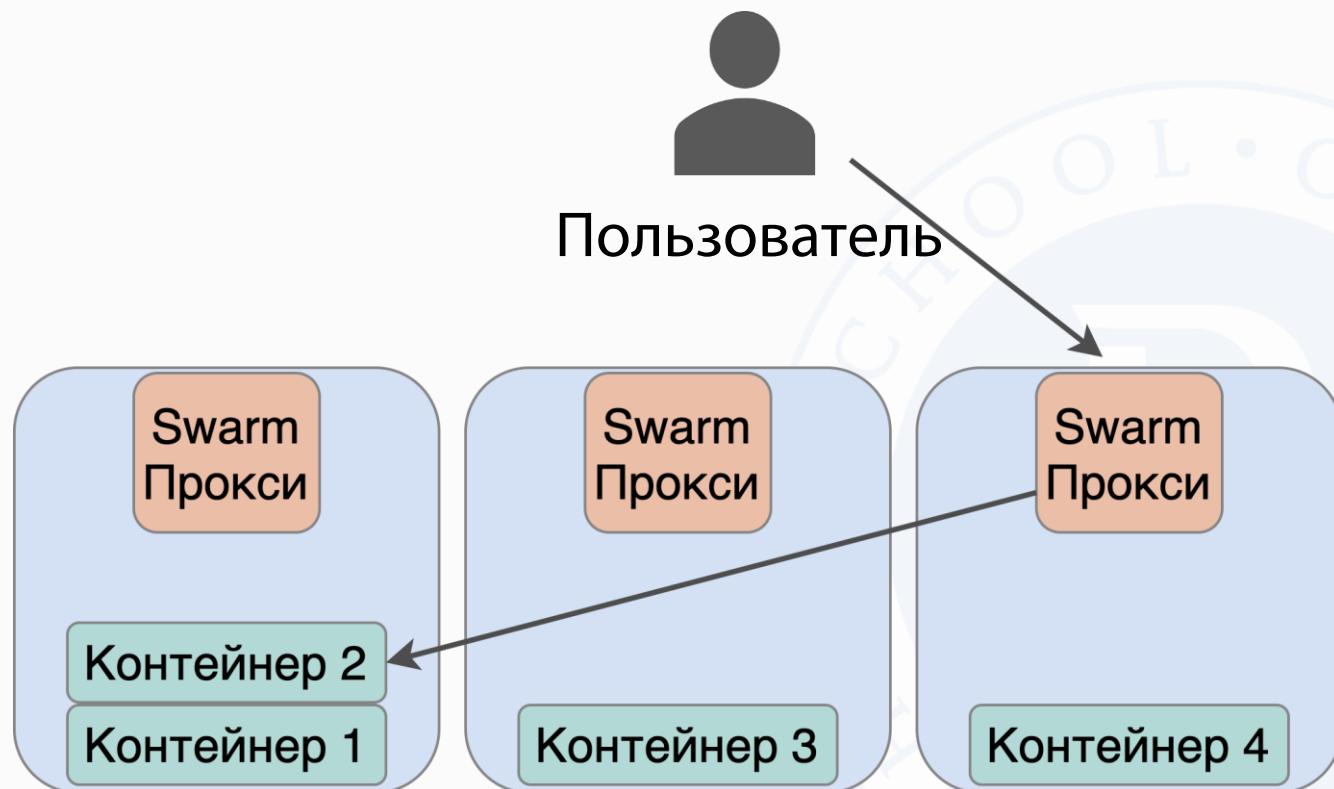
→ Swarm позволяет объединить несколько машин в один кластер



→ Планировщик сам умеет распределять контейнеры по машинам

# Docker Swarm

→ Пользователь может обращаться к любой машине кластера — Swarm **сам перенаправляет** запросы к нужному контейнеру



# Docker Swarm

→ Swarm использует ту же самую конфигурацию, что и Docker Compose

```
services:  
  application:  
    image: my-application:1.0.2  
    ports:  
      - 8000:8000  
    deploy:  
      mode: replicated  
      replicas: 3  
  database:  
    image: mongo:4  
    ports:  
      - 27017:27017
```

# Docker Swarm

→ В конфигурации можно дополнительно указывать параметры развертывания

```
services:  
  application:  
    image: my-application:1.0.2  
    ports:  
      - 8000:8000  
  deploy:  
    mode: replicated  
    replicas: 3  
  database:  
    image: mongo:4  
    ports:  
      - 27017:27017
```

# Docker Swarm

→ В этой конфигурации мы указали, что хотим три копии нашего приложения в кластере

```
services:  
  application:  
    image: my-application:1.0.2  
    ports:  
      - 8000:8000  
    deploy:  
      mode: replicated  
      replicas: 3  
  database:  
    image: mongo:4  
    ports:  
      - 27017:27017
```

# Docker Swarm



Для развертывания написанной конфигурации по кластеру достаточно [запустить одну команду](#)

```
docker stack deploy --compose-file  
docker-compose.production.yaml
```

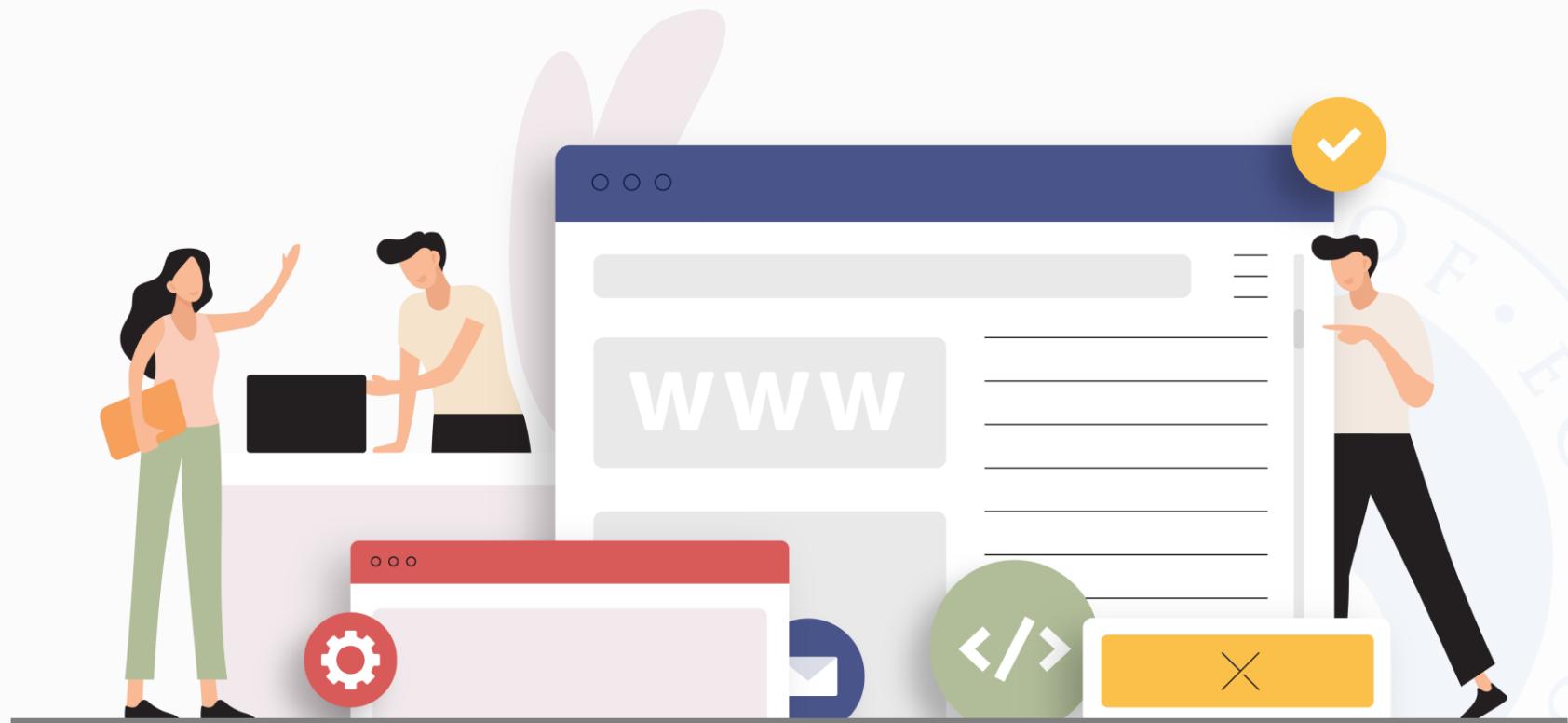


Команда [распределит и запустит](#) контейнеры на машинах, которые были объединены в кластер

# Практическая сторона вопроса



В следующем видео рассмотрим, как на практике запускать и развертывать кластеры



# **Обработка запросов в реальном времени**



# Классическая модель «клиент — сервер»

- Обычные веб-сервисы используют классическую модель «клиент — сервер»
- Часто эту же модель можно использовать и для ML-системы — клиент посылает **запрос с данными**, сервер возвращает **ответ с предсказанием**

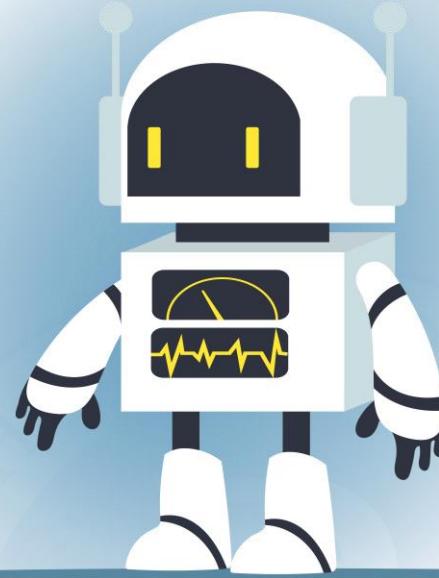


# Классическая модель «клиент — сервер»



Для эффективной работы такого подхода нужно, чтобы:

- Модель была **простая**
- Объем данных, для которых необходимо сделать предсказание, был **небольшим**



# Классическая модель «клиент — сервер»



В противном случае сервер будет слишком **долго** производить вычисления, что приведет к:



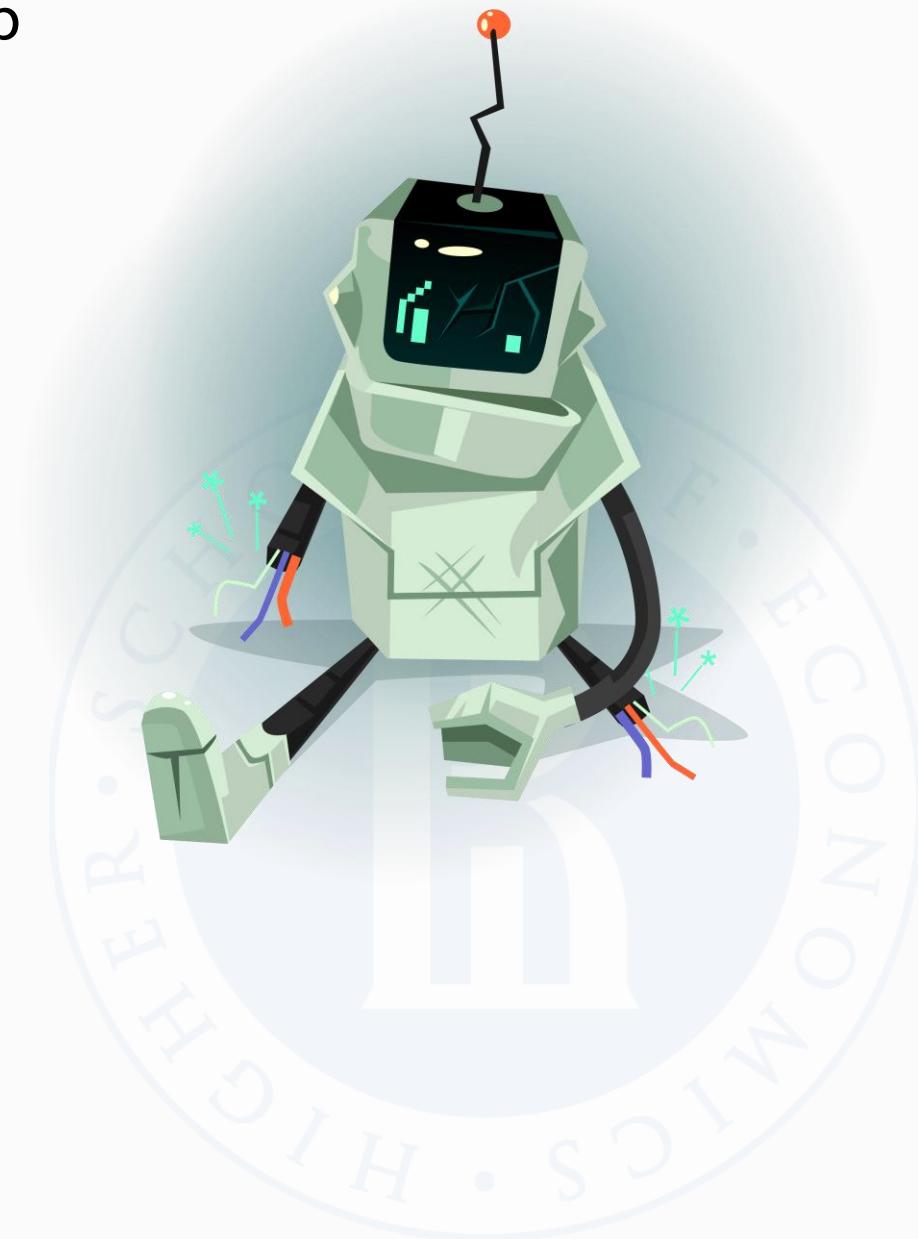
Зависаниям



Отказам

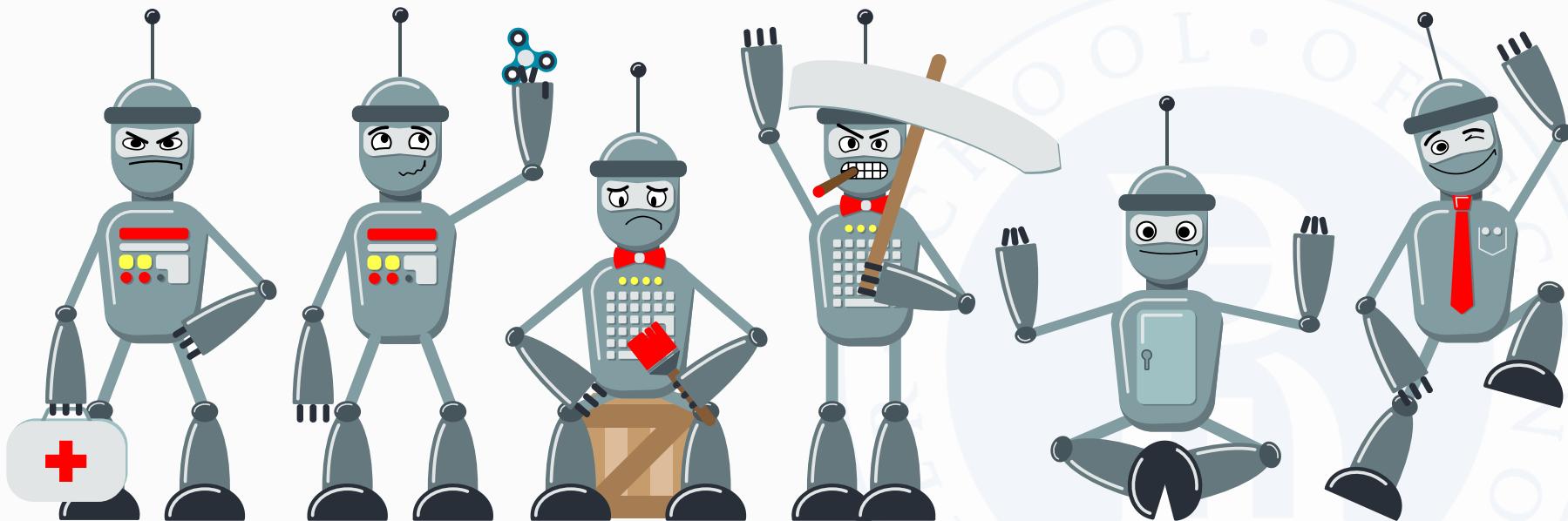


Аварийному завершению работы сервера



# Масштабирование

- Один сервер — это мало для того, чтобы обрабатывать большие нагрузки
- Для более стабильной работы необходимо распределять нагрузку между несколькими машинами



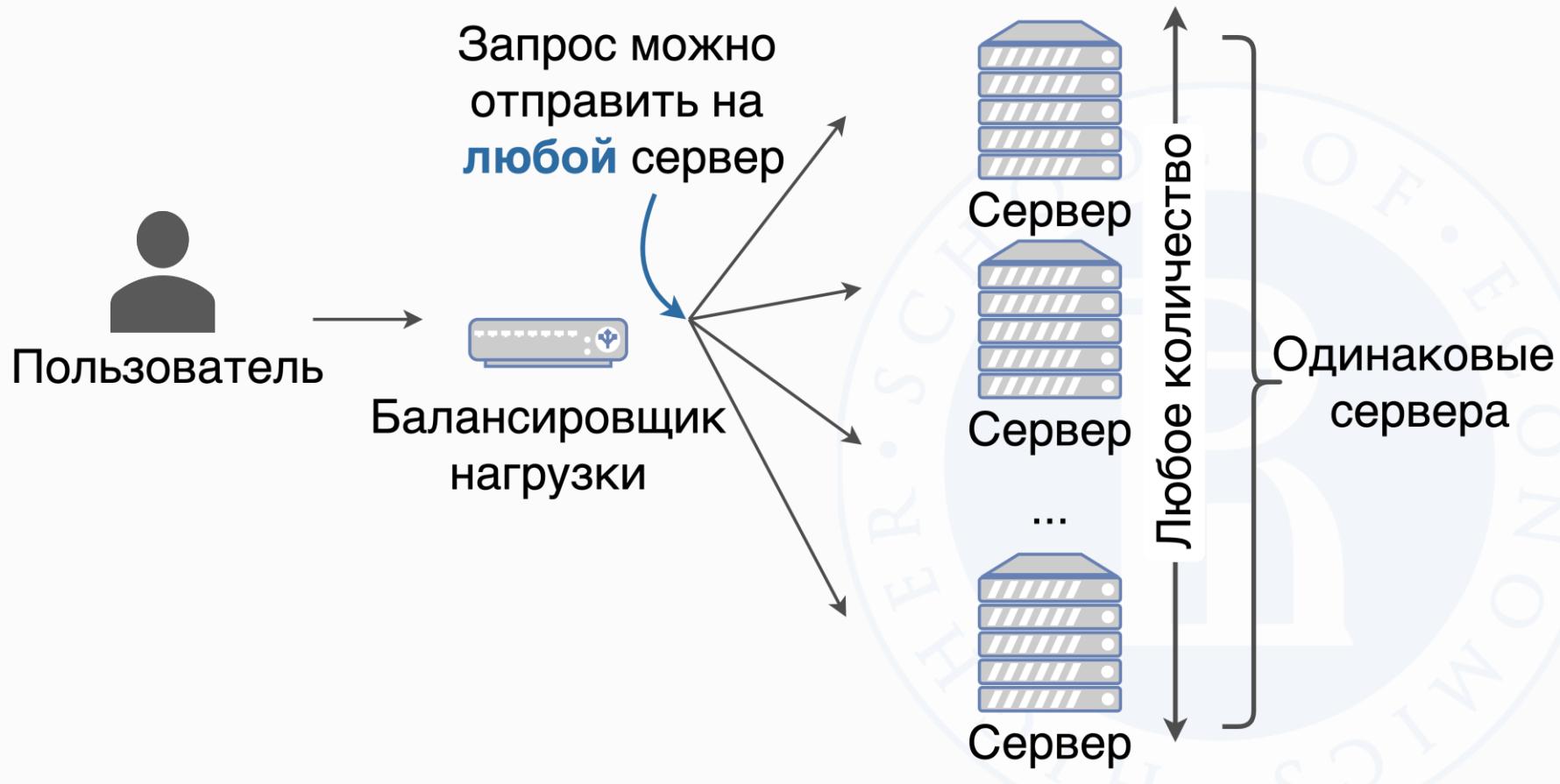
# Масштабирование



Чтобы масштабировать приложение, оно должно быть **самодостаточным и не иметь общего состояния**



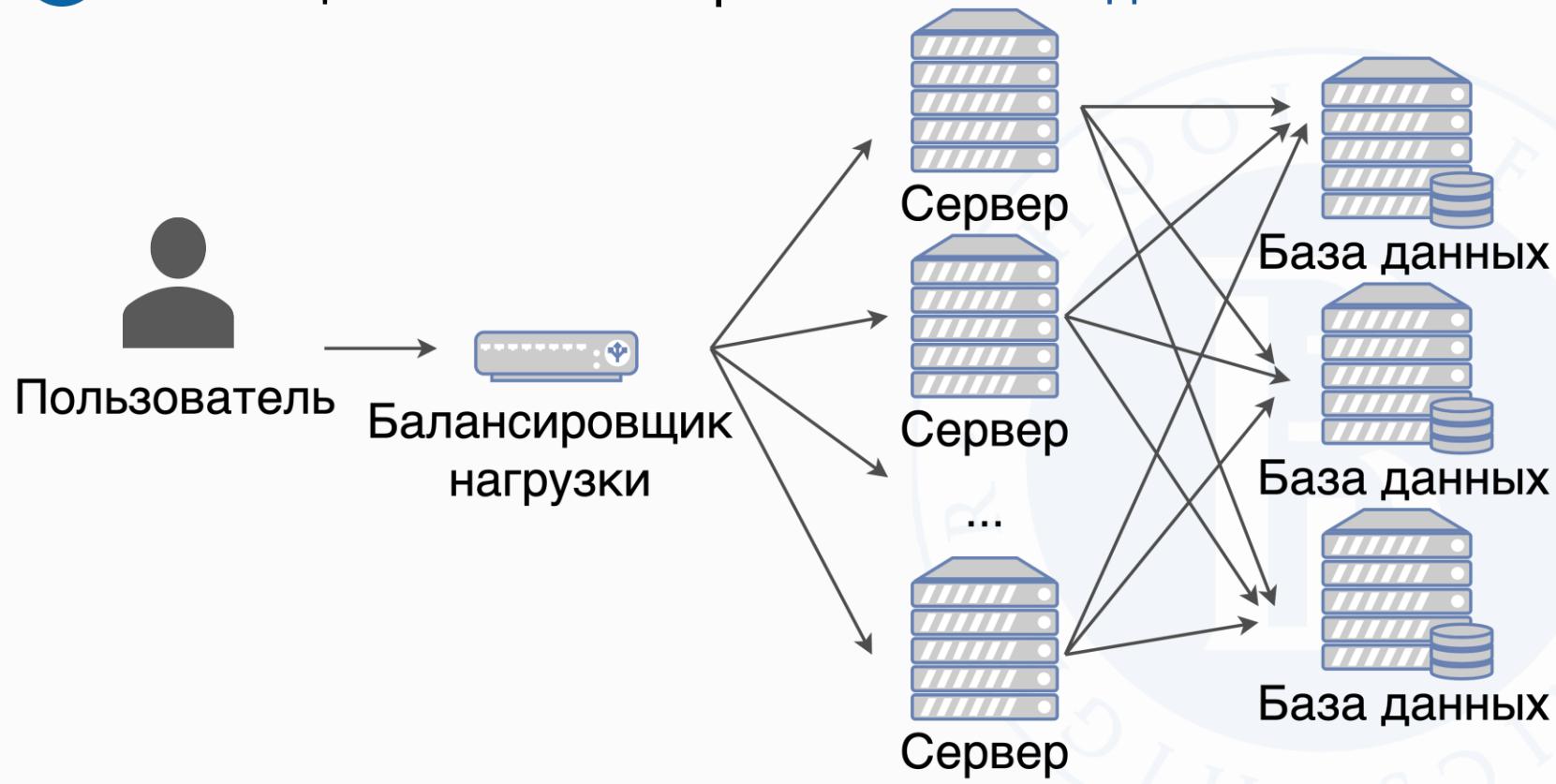
Тогда **любая** копия сможет обработать запрос



# Масштабирование

Это означает, что

- Все необходимое уже загружено в приложение
- Данные не сохраняются локально (в файлы и т.д.)
- Все общее состояние хранится в базе данных



# Кеширование

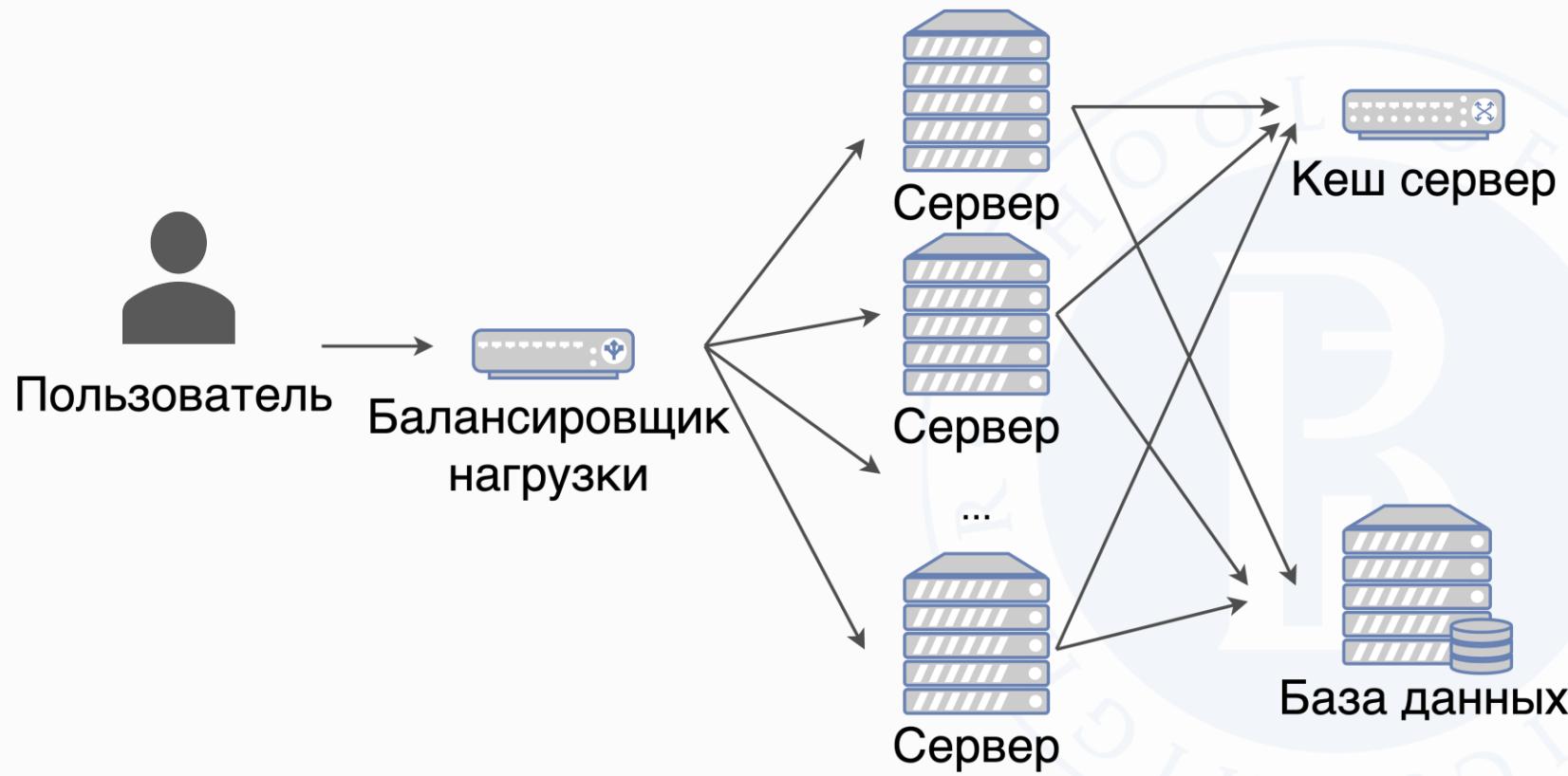
→ Чтобы дополнительно **ускорить** работу приложения и освободить сервер от лишних вычислений, можно использовать **технику кеширования**



**Memcached**

# Кеширование

- Выделяется небольшое хранилище, куда складываются уже **подсчитанные** результаты для частых запросов
- Тогда для ответа на такие же запросы в будущем их **не нужно** лишний раз считать



# Веб-сервер на Flask

→ Минимальный ML-сервис может быть крайне небольшим

```
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True)))
app = Flask(__name__)
@app.route('/predict')
def hello_world():
    request_data = request.get_json()
    data = np.array(request_data['data'])
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Веб-сервер на Flask

→ Подготовим модель. Как загружать уже предобученную модель — расскажем на другой неделе

```
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True))
)
app = Flask(__name__)
@app.route('/predict')
def hello_world():
    request_data = request.get_json()
    data = np.array(request_data['data'])
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Веб-сервер на Flask

→ Создаем специальный объект приложения Flask

```
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True))
)
app = Flask(__name__)
@app.route('/predict')
def hello_world():
    request_data = request.get_json()
    data = np.array(request_data['data'])
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Веб-сервер на Flask

→ Создаем обработчик HTTP на пути /predict

```
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True))
)
app = Flask(__name__)
@app.route('/predict')
def hello_world():
    request_data = request.get_json()
    data = np.array(request_data['data'])
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Веб-сервер на Flask



Предполагаем, что нам пришел POST запрос с JSON.  
Читаем этот JSON

```
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True))
)
app = Flask(__name__)
@app.route('/predict')
def hello_world():
    request_data = request.get_json()
    data = np.array(request_data['data'])
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Веб-сервер на Flask

→ Формируем признаки и считаем предсказание

```
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True))
)
app = Flask(__name__)
@app.route('/predict')
def hello_world():
    request_data = request.get_json()
    data = np.array(request_data['data'])
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Веб-сервер на Flask



Возвращаем ответ также в виде JSON

```
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True))
)
app = Flask(__name__)
@app.route('/predict')
def hello_world():
    request_data = request.get_json()
    data = np.array(request_data['data'])
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Веб-сервер на Flask



Готово! Можно запускать наш сервер и отправлять POST запрос

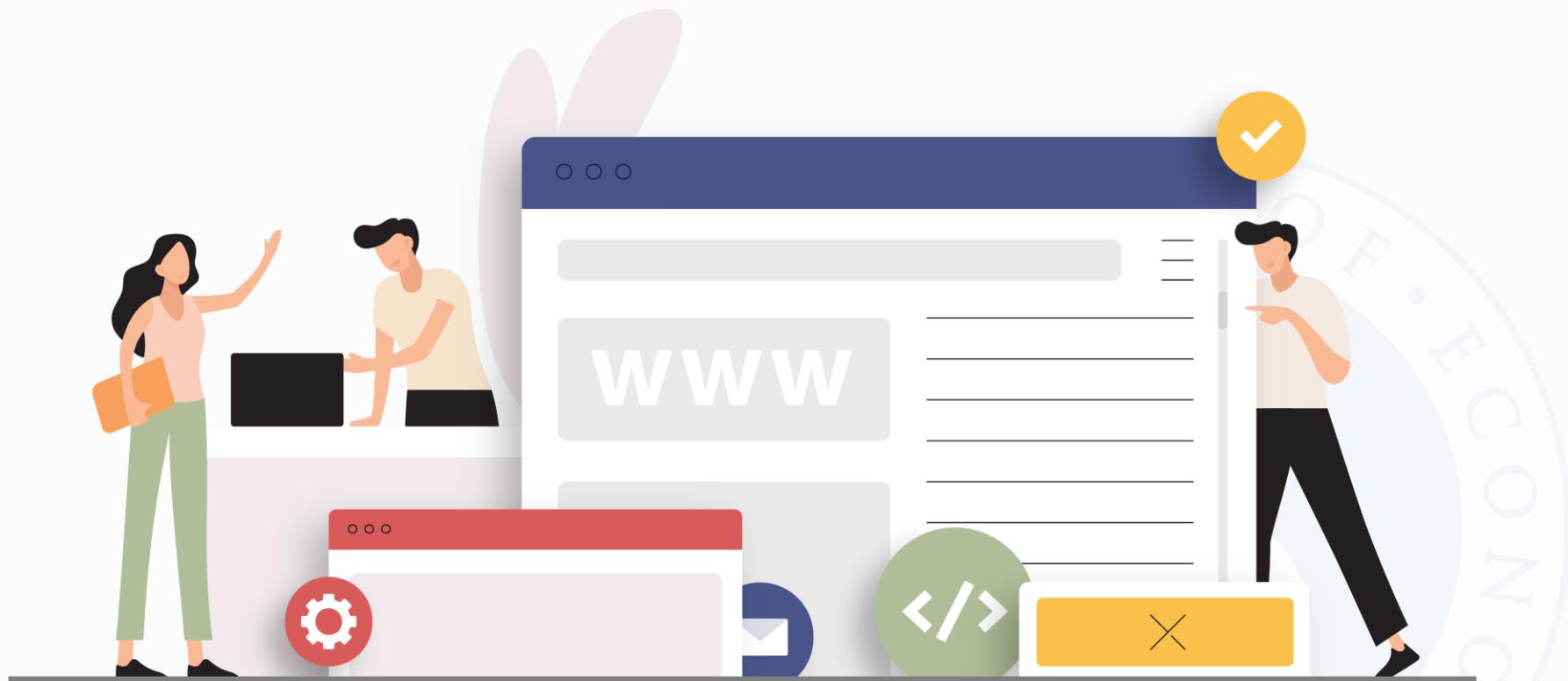
```
requests.post(  
    'http://localhost:8000/predict',  
    json={"data": [  
        [5.1, 3.5, 1.4, 0.2],  
        [2.4, 0.1, 5.3, 4.2]  
    ]}  
)
```

```
{  
    "result": [0, 2]  
}
```

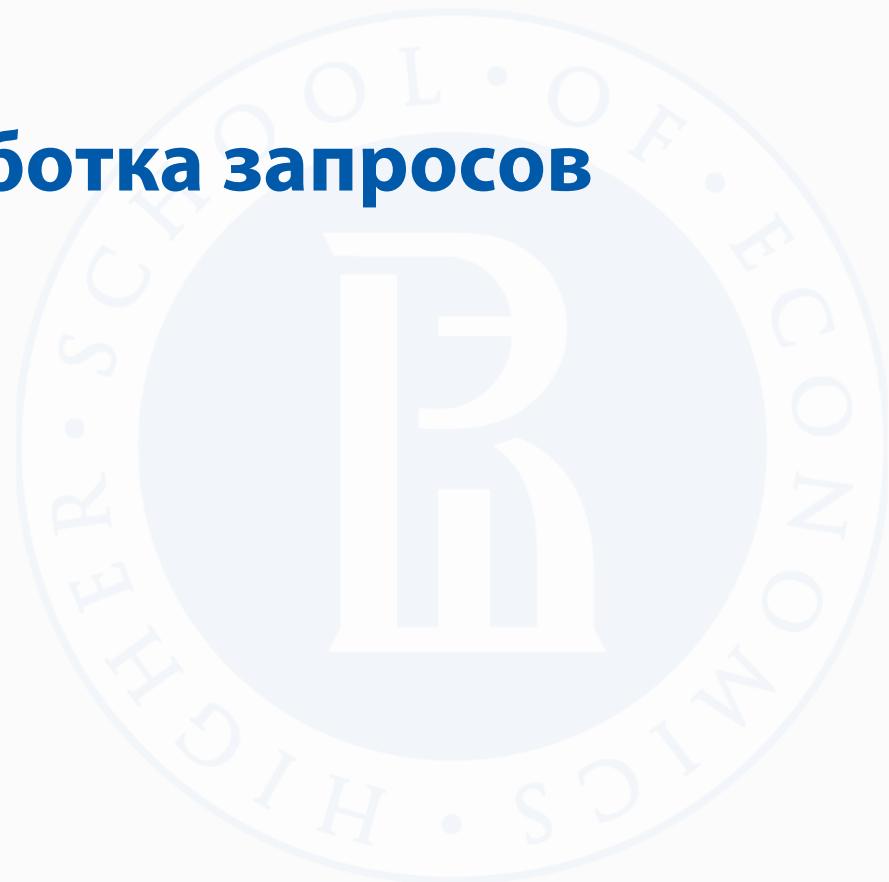
# Практическая сторона вопроса



В следующем видео рассмотрим, как на практике писать и запускать веб-приложения



# **Асинхронная обработка запросов**



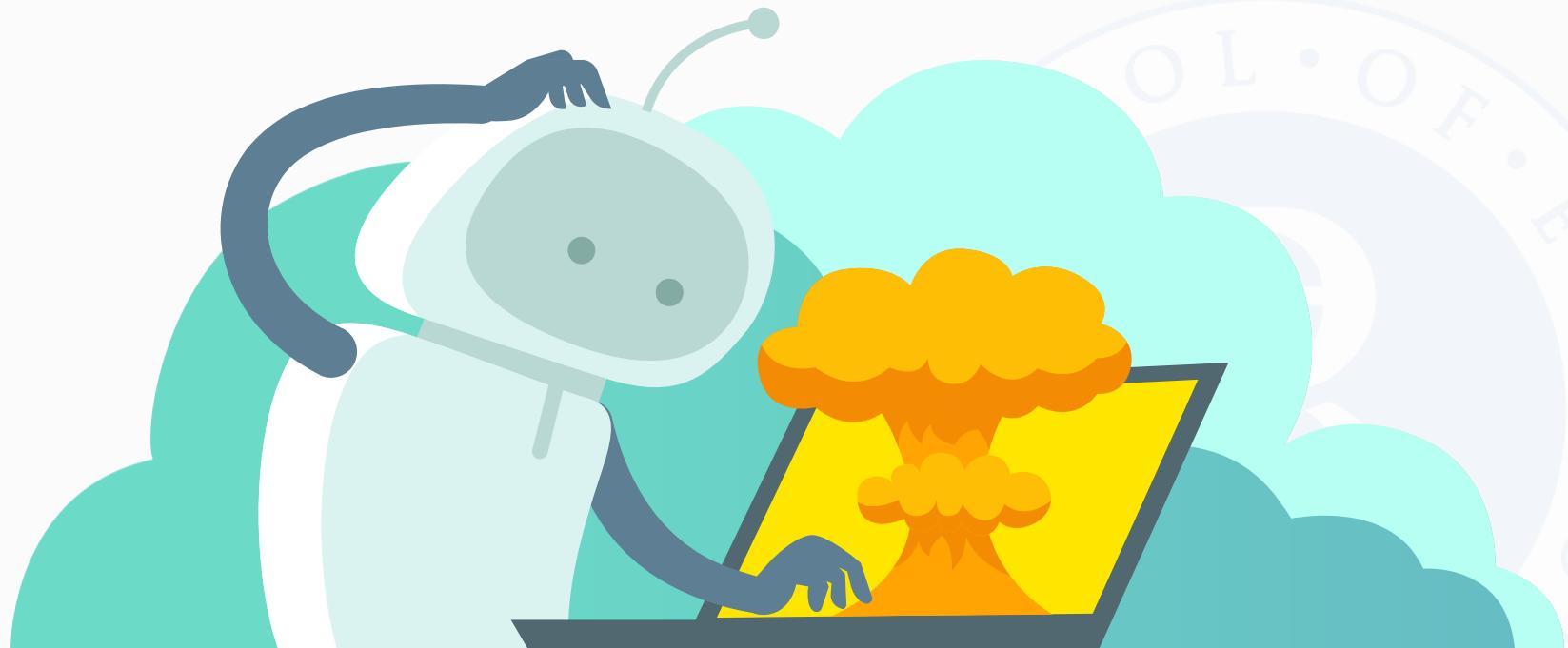
# Невозможность обработки в реальном времени

→ Не всегда реализованный алгоритм можно запустить в режиме реального времени!



# Тяжелые модели

- Если модель очень большая, то она будет потреблять много памяти и долго считать ответ
- Это будет приводить к отказам и аварийному завершению работы



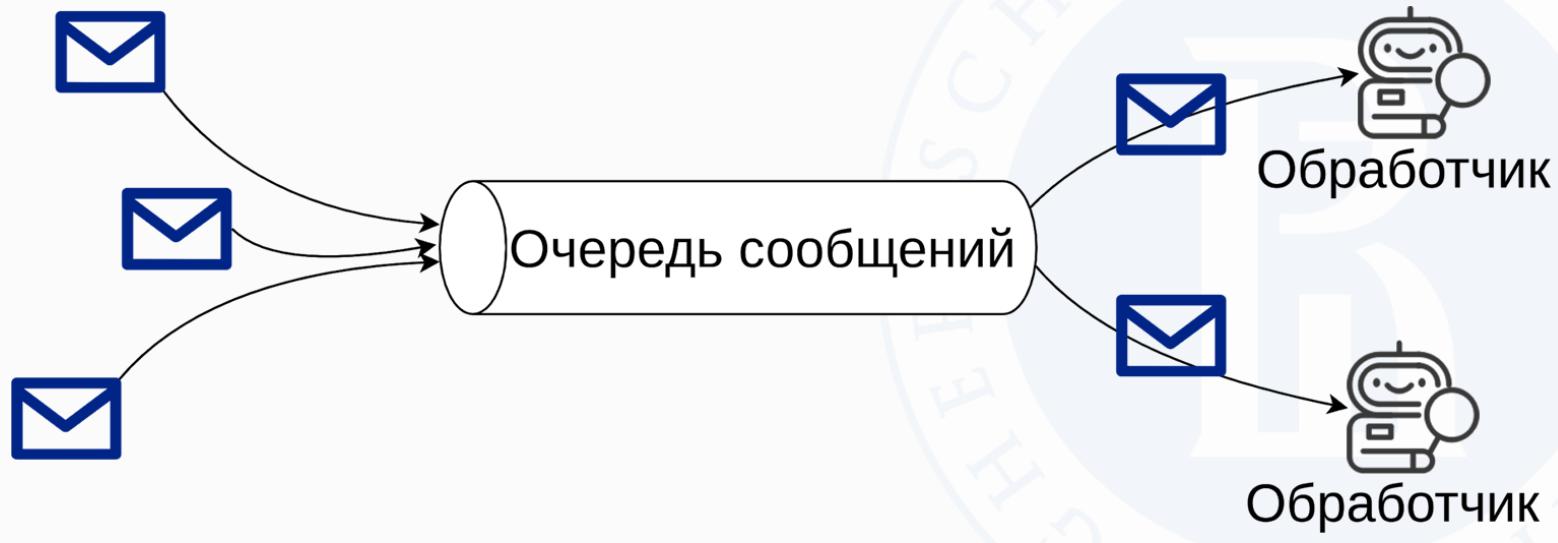
# Большой объем вычислений

- Данных для обработки может быть очень много
- Например, если мы хотим посчитать все рекомендации для пользователя



# Распределенные очереди сообщений

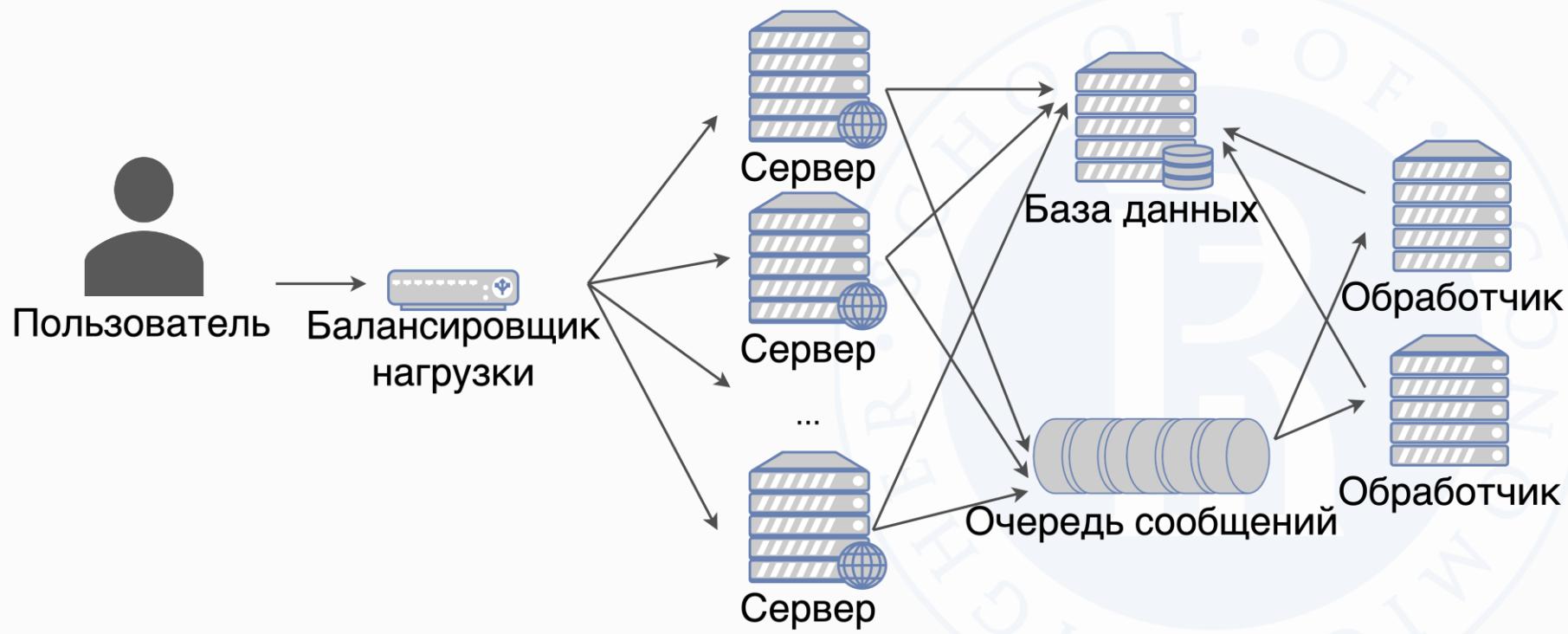
- Распределенная очередь сообщений — это система, собирающая входящие **сообщения**, из которой можно эти сообщения **получать** в изначальном порядке
- Это позволяет выполнять задачи **не сразу же**, а когда появится **возможность**



# Архитектура распределенных очередей



Взаимодействие с клиентом в такой архитектуре становится немного сложнее



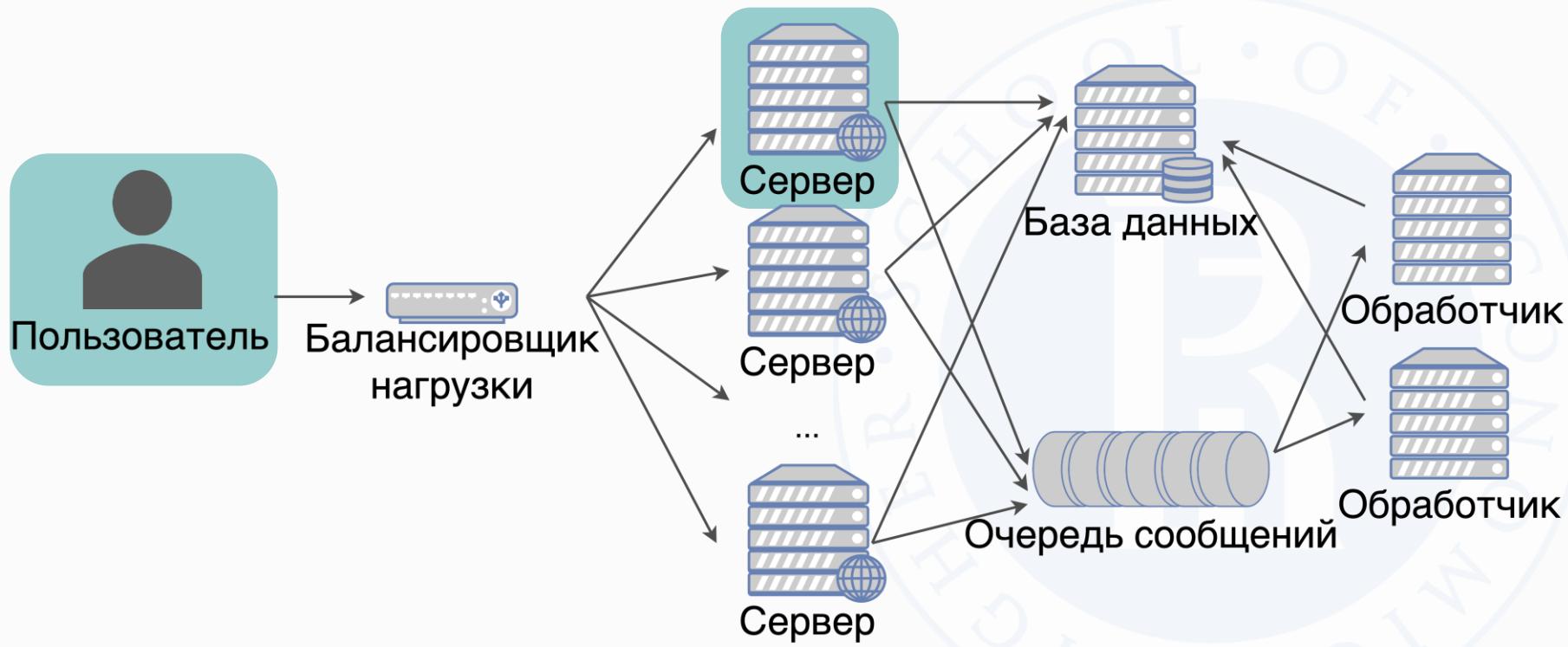
# Архитектура распределенных очередей



Клиент запрашивает предсказание на данных

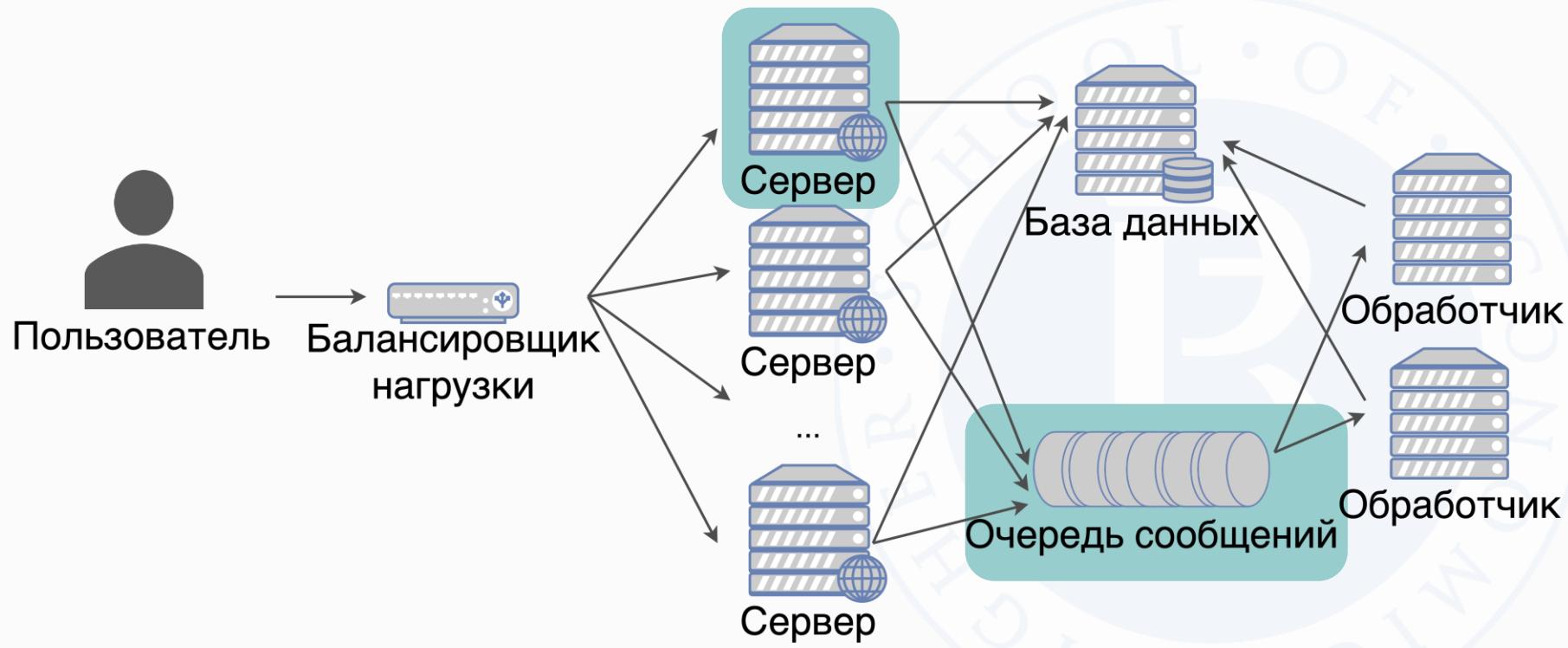


В ответ он получает **не ответ**, а лишь **номер** своего запроса



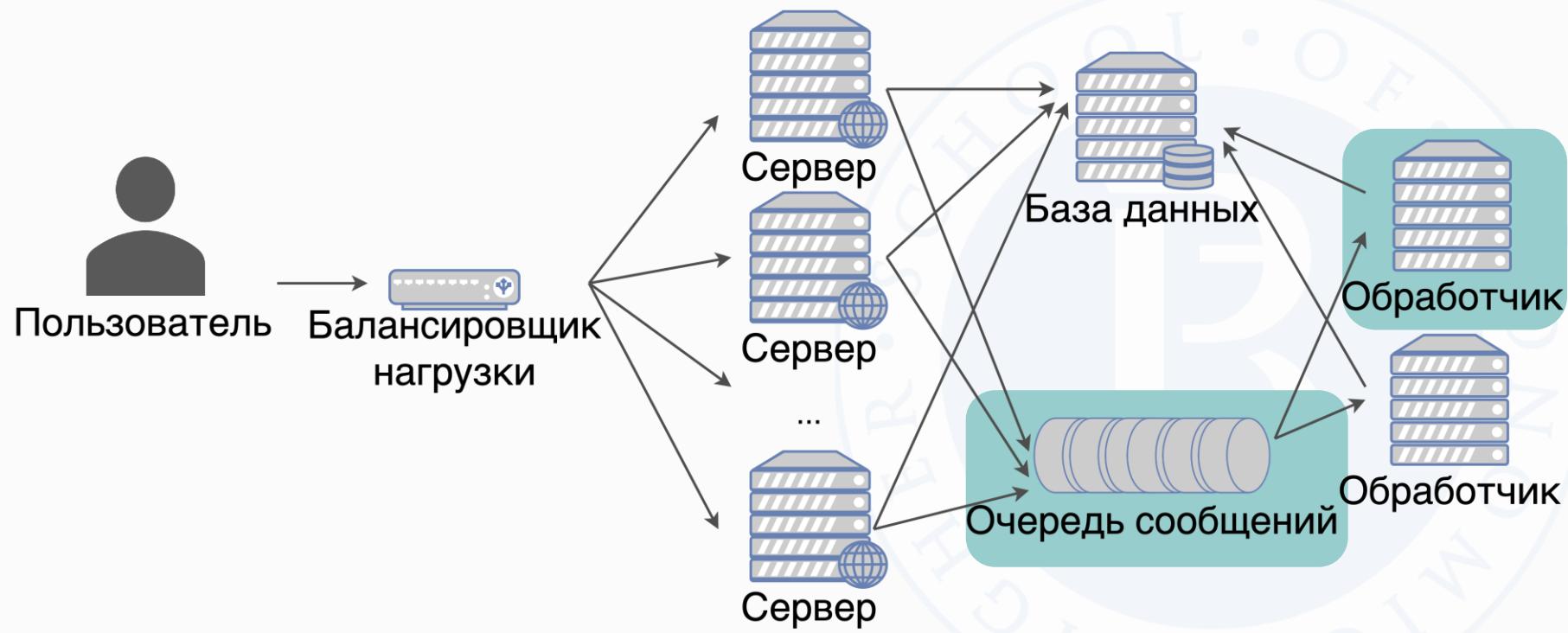
# Архитектура распределенных очередей

- Сервер на запрос не считает ответ сразу, а лишь кладет сообщение с запросом в очередь
- Клиенту он отдает номер его запроса



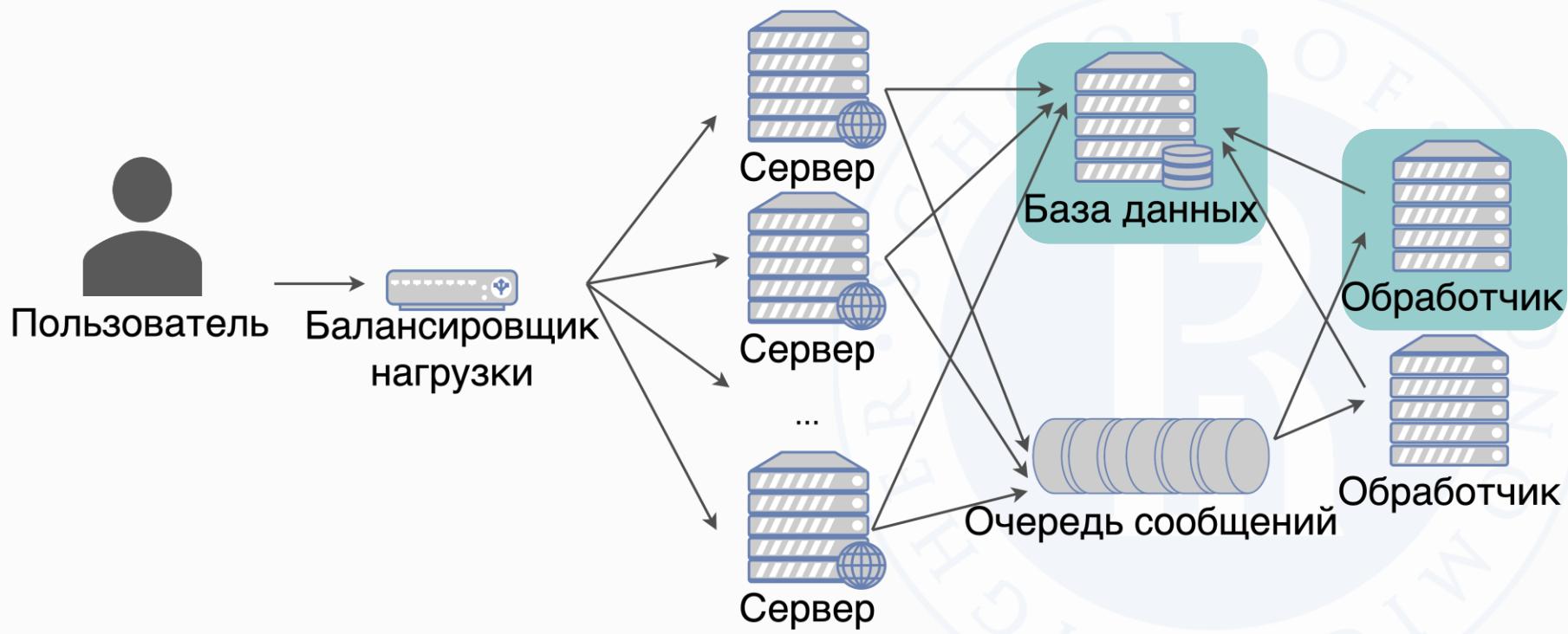
# Архитектура распределенных очередей

→ Сервер-обработчик **следит** за очередью, и как только там появляется сообщение, он берет его и выполняет запрошенный расчет



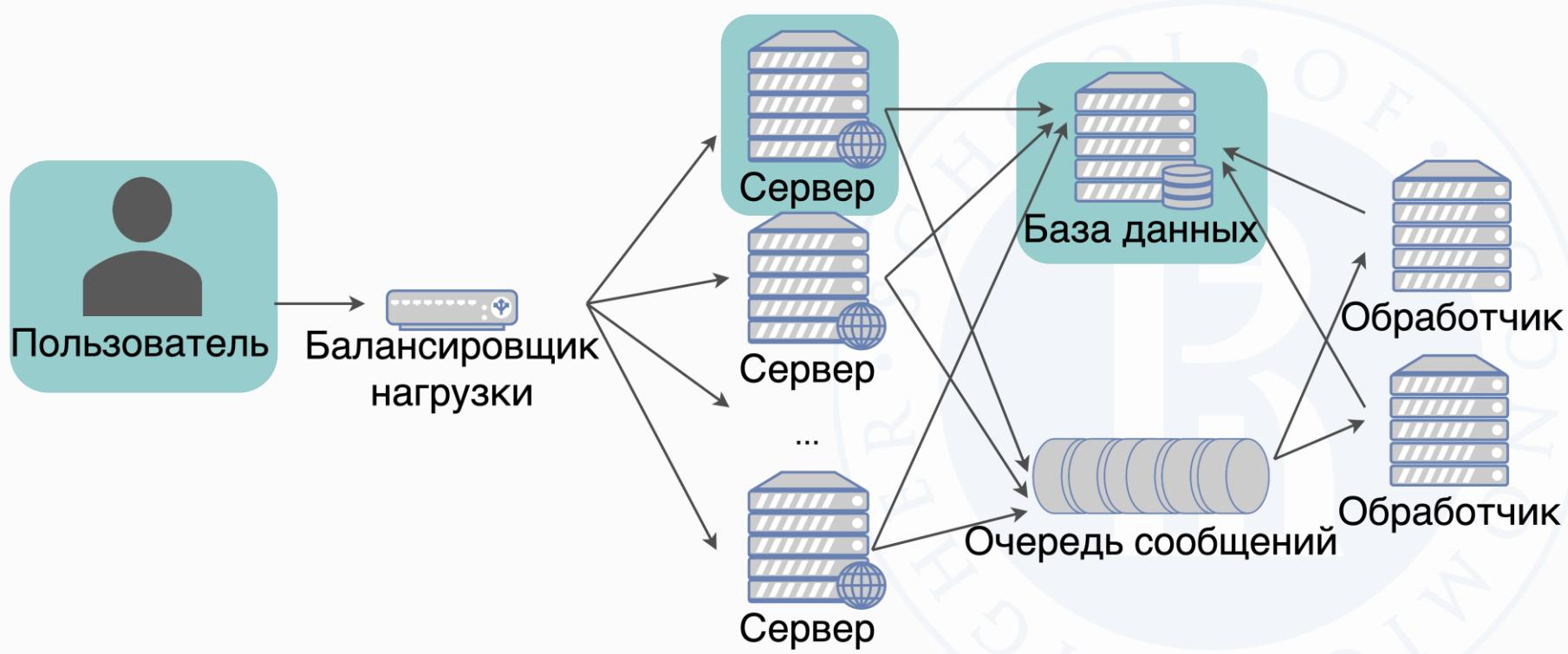
# Архитектура распределенных очередей

- Как только сервер завершает долгий расчет, он кладет результат в базу вместе с номером запроса
- После чего он продолжает следить за очередью



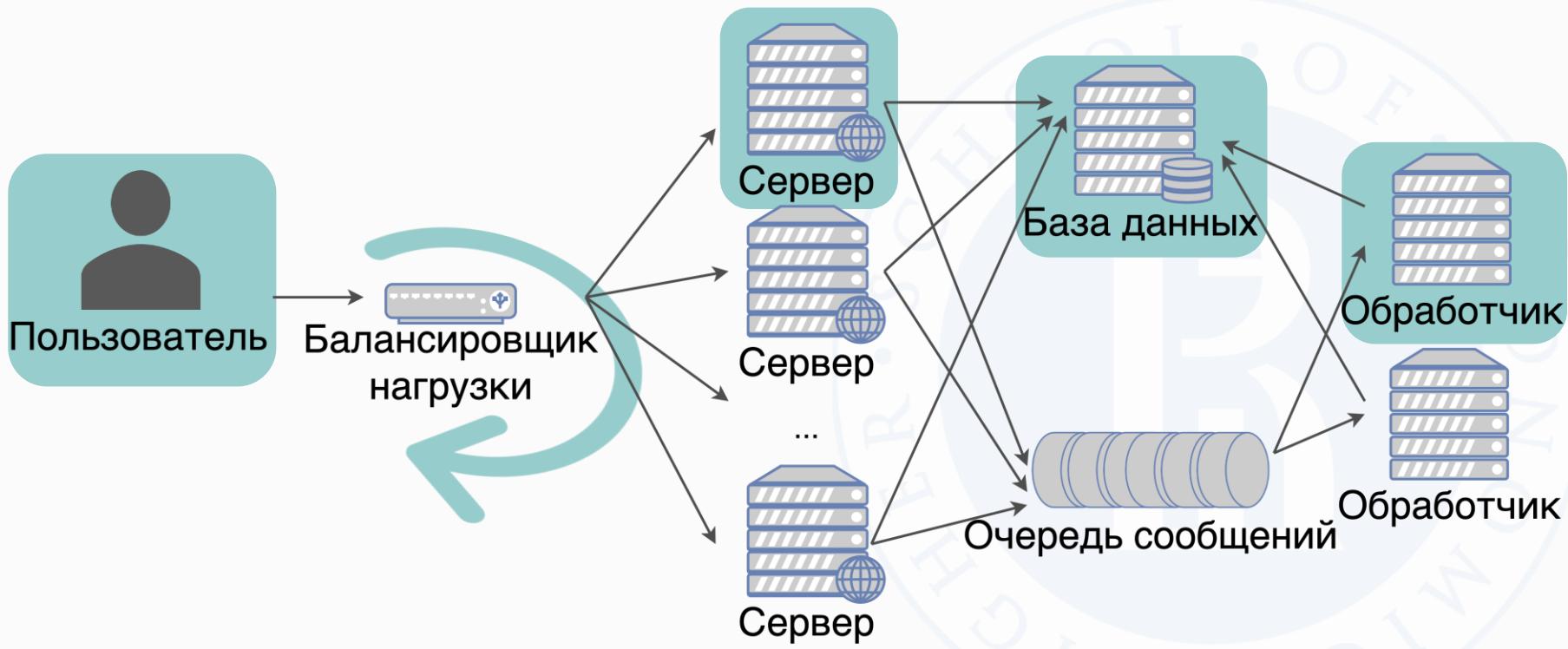
# Архитектура распределенных очередей

→ В какой-то момент клиент спрашивает состояние своего запроса. Сервер ищет результат с этим номером в базе и если находит, возвращает пользователю



# Архитектура распределенных очередей

→ Если сервер не нашел готовый результат, то сообщает, что **расчет в процессе**. Через некоторое время клиент придет еще раз и так, пока расчет не завершится



# Плюсы распределенных очередей

Архитектура с очередями очень гибкая, так как позволяет:

- ✓ Обрабатывать огромный поток запросов

# Плюсы распределенных очередей

Архитектура с очередями очень гибкая, так как позволяет:

- ✓ Обрабатывать огромный поток запросов
- ✓ Не перегружать систему

# Плюсы распределенных очередей

Архитектура с очередями очень гибкая, так как позволяет:

- ✓ Обрабатывать огромный поток запросов
- ✓ Не перегружать систему
- ✓ Все еще легко масштабировать ресурсы

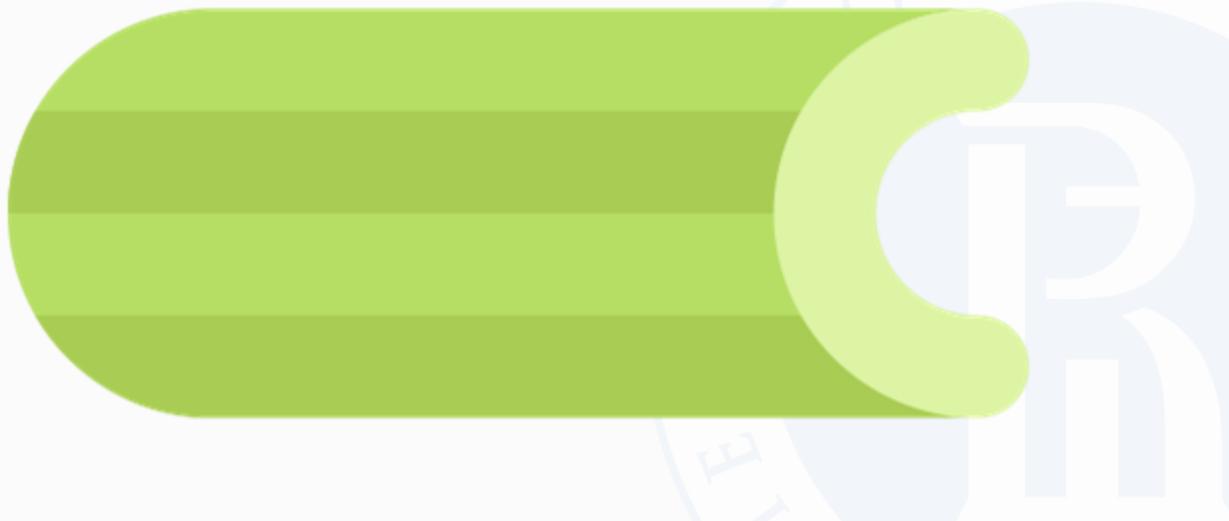
# Примеры очередей

The RabbitMQ logo features a stylized orange icon resembling a rabbit's head on the left, followed by the word "RabbitMQ" in a large, orange and grey sans-serif font. A trademark symbol (TM) is located at the top right of the "M".

redis.io  
rabbitmq.com  
kafka.apache.org

# Celery

→ Celery — библиотека для Python, позволяющая создавать обработчики для очереди сообщений



# Пример

→ Создадим обработчик для расчета предсказаний

```
app = Celery('tasks',
broker='redis://guest@localhost//')
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True)))
@app.task
def predict(request_data):
    data = np.array(request_data)
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Пример

→ Создаем специальный объект Celery, указывая название и сетевой путь до очереди сообщений

```
app = Celery('tasks',
broker='redis://guest@localhost//')
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True)))
@app.task
def predict(request_data):
    data = np.array(request_data)
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Пример



Подготовим модель

```
app = Celery('tasks',
broker='redis://guest@localhost//')
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True)))
@app.task
def predict(request_data):
    data = np.array(request_data)
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Пример

→ Создадим обработчик на события

```
app = Celery('tasks',
broker='redis://guest@localhost//')
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True)))
@app.task
def predict(request_data):
    data = np.array(request_data)
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Пример

→ Подсчитаем предсказание

```
app = Celery('tasks',
broker='redis://guest@localhost//')
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True)))
@app.task
def predict(request_data):
    data = np.array(request_data)
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Пример

→ Redis умеет работать как очередь и как база данных.  
Это позволяет сохранить результат работы в нем же

```
app = Celery('tasks',
broker='redis://guest@localhost//')
model = (
    LogisticRegression(random_state=0)
    .fit(*load_iris(return_X_y=True)))
@app.task
def predict(request_data):
    data = np.array(request_data)
    predictions = model.predict(data)
    return {"result": list(predictions)}
```

# Пример

→ Для запуска обработчика достаточно запустить одну команду

```
celery -A tasks worker
```

→ После чего в соседнем скрипте можно добавить задачу в очередь

```
from tasks import predict  
request = predict.delay([[5.1, 3.5, 1.4, 0.2],  
[2.4, 0.1, 5.3, 4.2]])
```

# Пример



После подсчета результат будет сохранен в Redis



Проверить его готовность можно через функцию ready

```
request.ready()
```

# Периодические задачи

- Иногда требуется проводить расчеты периодически (например, раз в неделю)
- Для этого Celery можно настроить на [периодические задачи](#)

```
sender.add_periodic_task(  
    crontab(hour=7, minute=30, day_of_week=1),  
    predict.s(data),  
)
```

- В этом примере задача predict будет запускаться каждый понедельник в 7:30

# Практическая сторона вопроса



В следующем видео рассмотрим, как на практике реализовать приложение с очередью сообщений

