

Мониторинг, потоки данных, облачка

Борисенко Глеб
ФТиАД2022

Что было в прошлый раз

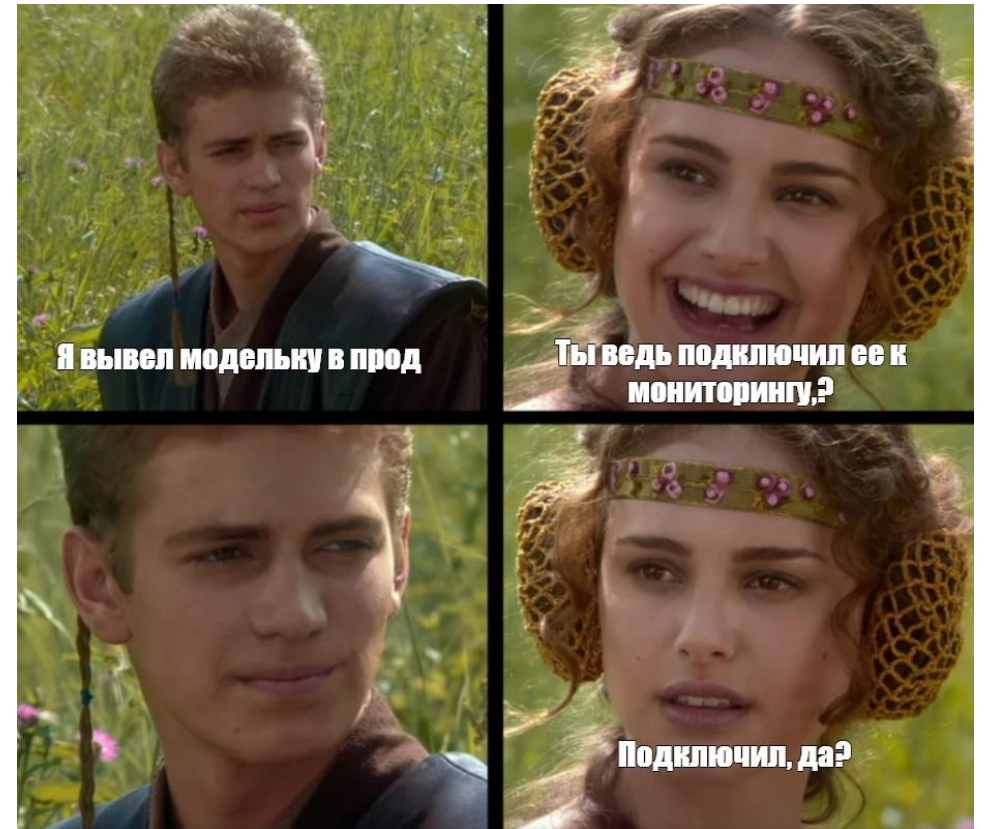
- CI/CD
- Пайплайны

Сегодня на повестке

- Мониторинг
 - Grafana + VictoriaMetrics
 - Evidently
- Поток данных
 - Kafka
 - Flint
- Облачные технологии
 - Пару слов о SberCloud, Heroku, AWS и т.п.
- Архитектура в новом свете

Мониторинг

- Мы вывели модельку в прод, ура!
- А как понять, что где-то появилась проблема?
- Ответ прост: Мониторинг!



Чем отличается мониторинг в ML от классики?

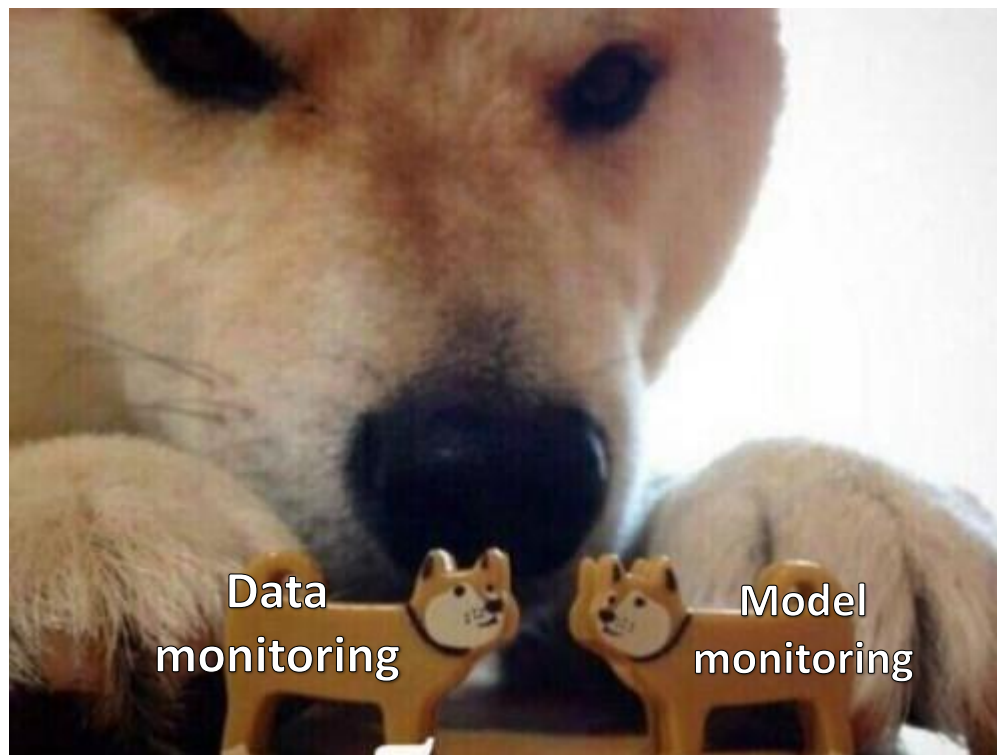
- Service health - совпадает с мониторингом любого ИТ сервиса: проверяем, жив ли сервис, пайплайны, здоровы ли
- Model health - проверка на то, что модель в проде делает полезные предсказания

Методики классического мониторинга

- USE method (Utilization, Saturation и Errors)
- RED method (Rate (Requests), Errors, Duration)
- The Four Golden Signals by Google (Latency, Traffic, Errors, Saturation)
- UCA method (Users, Conversions, Activity)
- Есть еще, вроде как, но менее популярные :)

А что по мониторингу ML Systems?

- Data monitoring
- Model monitoring



Data monitoring

- что-то изменили в схеме данных
- один из источников данных внезапно отвалился
- пришла только часть данных или неправильные данные
- **Что можно проверить?**
 - Совпадают ли типы данных?
 - Не изменилась ли доля пропусков в данных?
 - Значения признаков остались в “нормальном” диапазоне:
 - Появились ли новые категории?
 - Непрерывные признаки принимают значения, которые модель раньше не видела?
 - Изменились ли статистики по признакам, распределение признаков?

Model monitoring

- «Смещения» в окружении
- Изменения в поведении пользователей
- **Что можно проверить?**
 - Контролировать качество ML моделей
 - Сравнивать распределения предсказаний
 - Базовые статистики
 - Статистические тесты

Мини инсайты

- даже базовый мониторинг лучше чем его отсутствие
- думайте о своей задаче
- следите за блогами стартапов

Инструменты

- Grafana – дашборды; лидер рынка
- Prometheus – классика, старичок, и самый популярный
- VictoriaMetrics – изначально замена стандартному хранилищу метрик в Prometheus; сейчас уже типа прометеуса, только лучше; относительно новый
- Бот в вашей сети или почтовая рассылка для алертов

Grafana

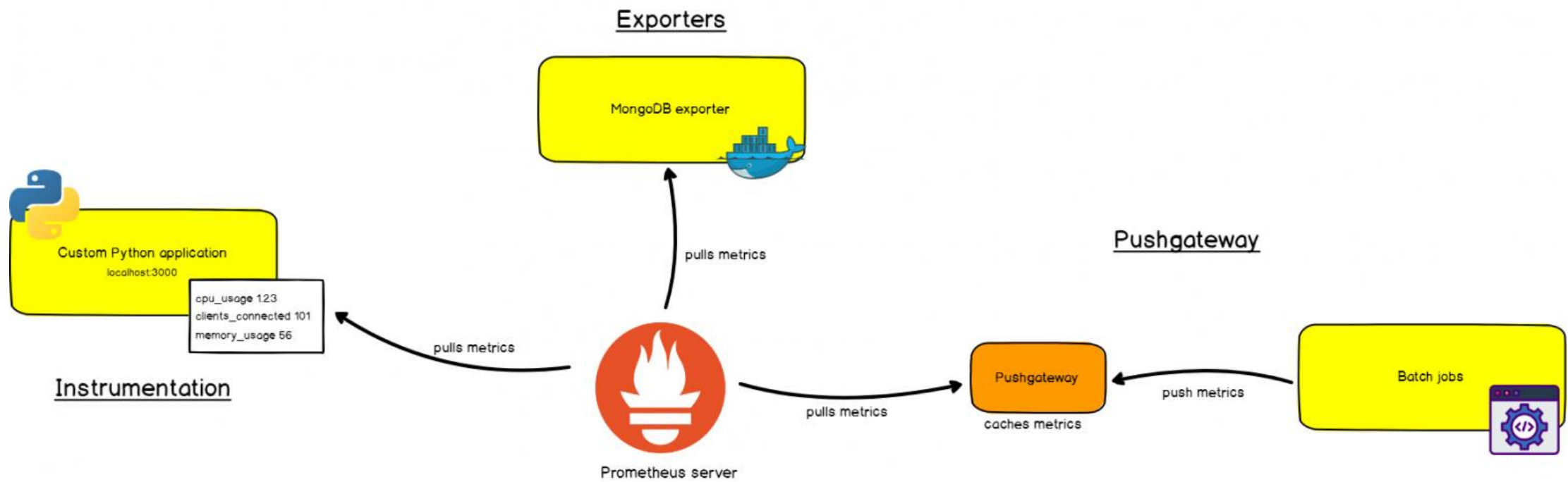
- позволяет легко делать дашборды
- комбинировать на них данные из разных источников (можно на графиках по метрикам из Prometheus нанести события из логов, собранных в Loki)
- совместима с множеством источников
- позволяет настраивать alerts

Grafana

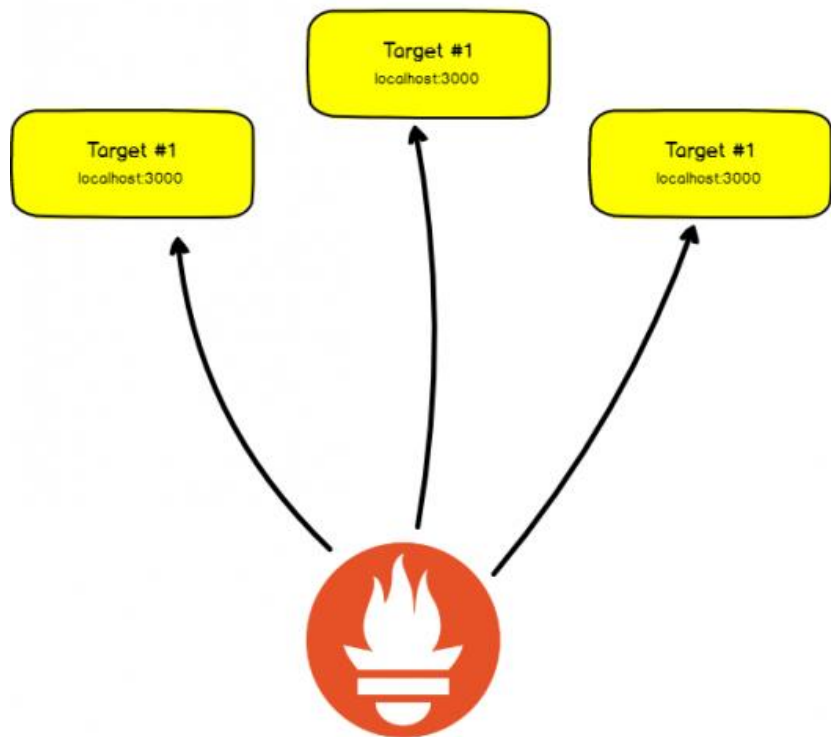


Prometheus (это база)

Ways to gather metrics in Prometheus

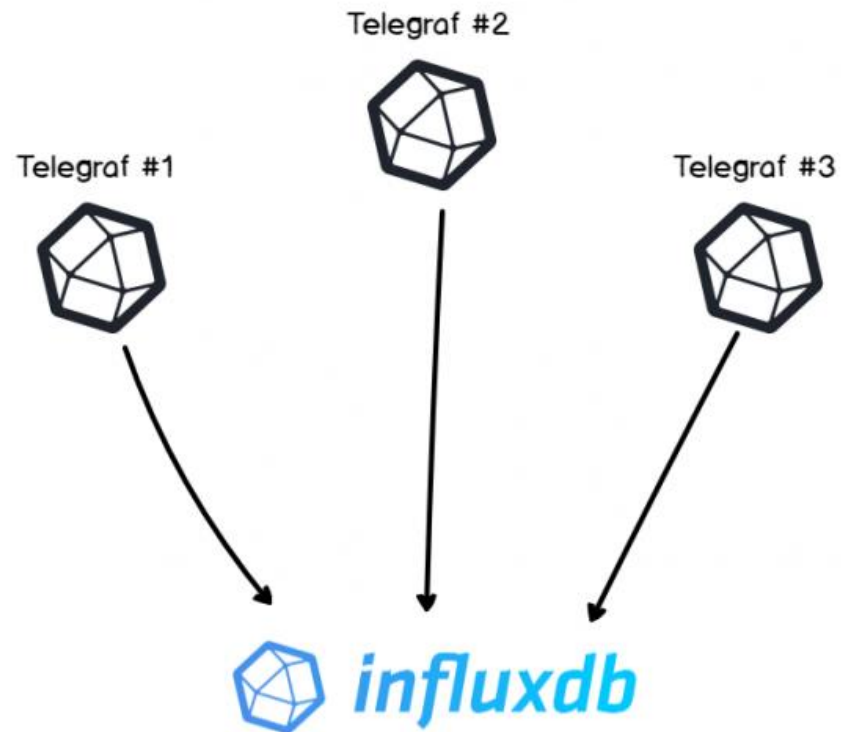


Push vs Pull



Prometheus pull data every 10 seconds

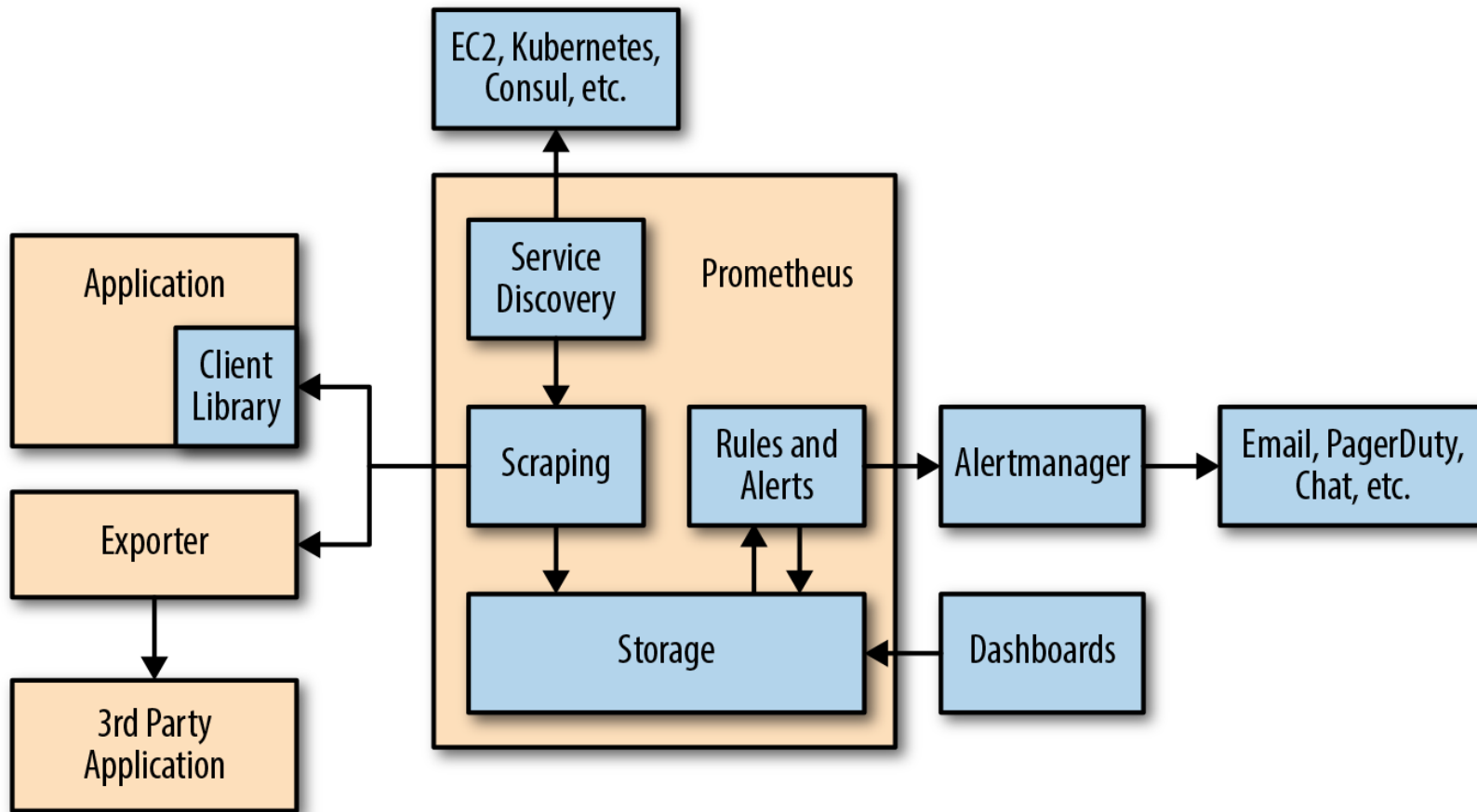
Prometheus is **active**



Individual instances push data every 10 seconds

InfluxDB is **passive**

Экосистема Prometheus



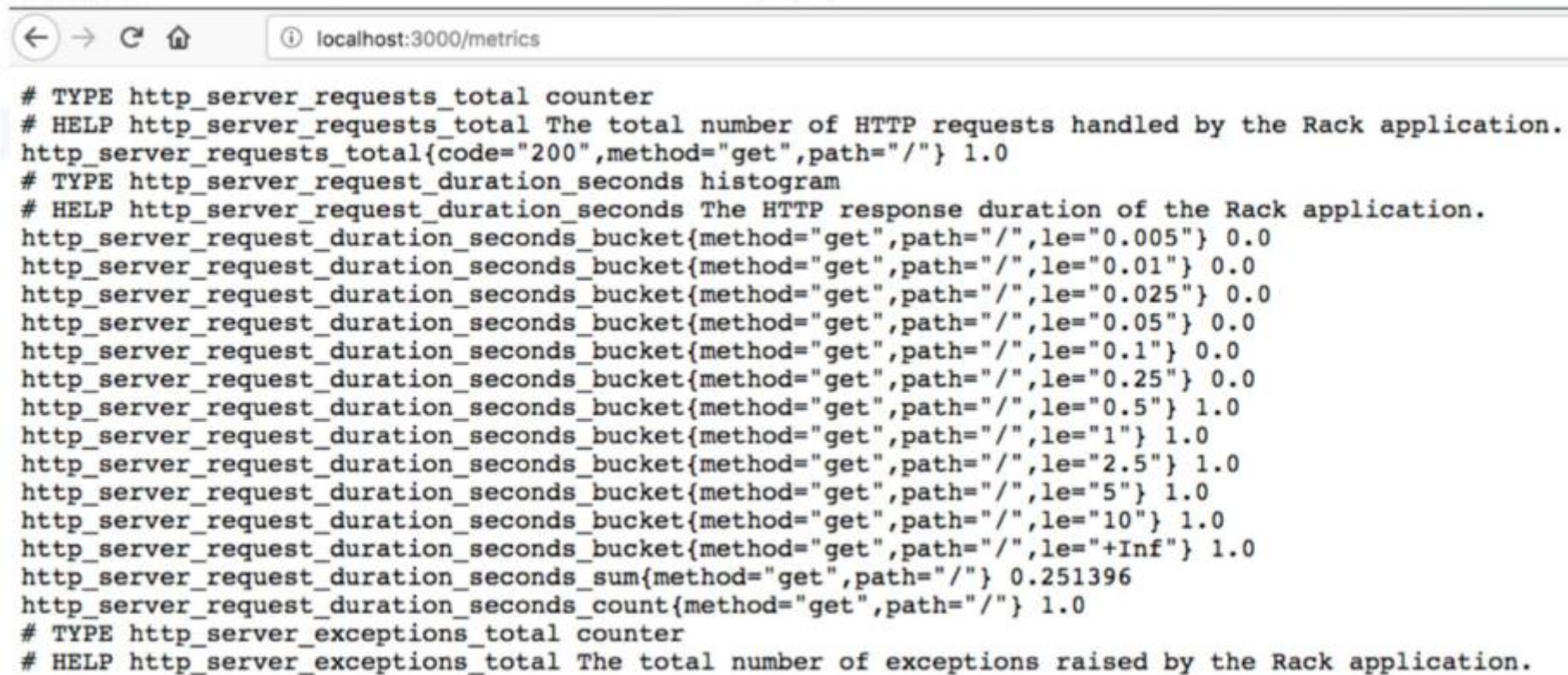
Push mechanism

- На каждый объект, который нужно отслеживать (target) необходимо установить ПО которое будет отправлять push запросы
- Если необходимо отслеживать много микросервисов и каждый из них отправляет данные в систему мониторинга, может возникнуть высоконагруженный трафик и система мониторинга становится узким местом. (инфраструктура перегружена постоянными push запросами)
- Если мы перестали получать метрики от сервиса, это не может однозначно трактоваться как остановка сервиса (может что-то с сетью, или потерялся пакет, итд)

Pull mechanism

- нет неопределенности, которая возникает в push mechanism системах, если запрос не пришел
- мы можем регулировать нагрузку на сеть
- микросервисы для работы с Prometheus должны иметь endpoint, кроме того для многих решений есть готовые exporters
- если нам критична точность и мы хотим учитывать каждое значение метрики, pull mechanism не лучший выбор

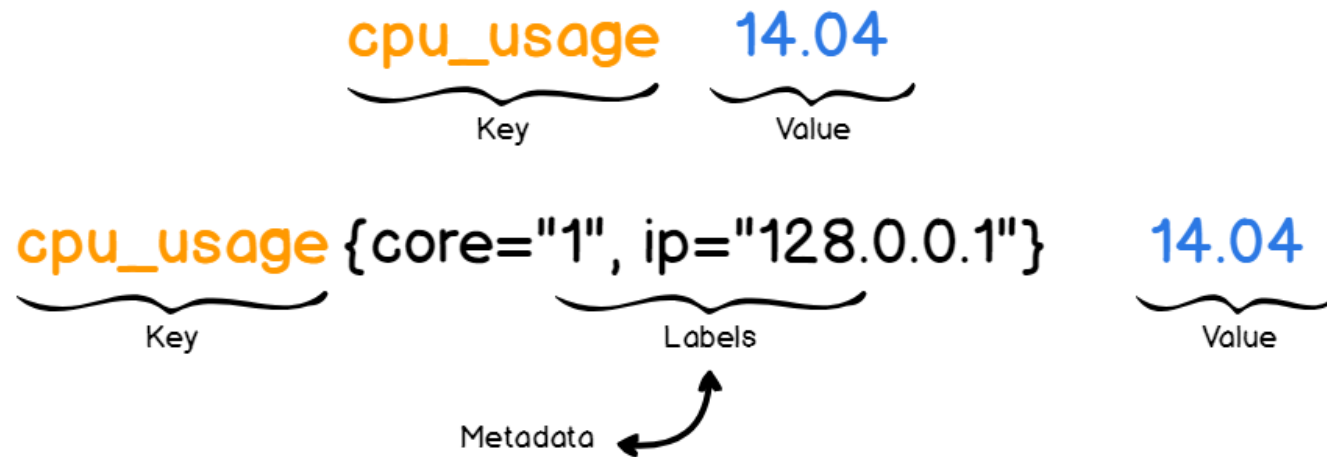
Как это выглядит



```
# TYPE http_server_requests_total counter
# HELP http_server_requests_total The total number of HTTP requests handled by the Rack application.
http_server_requests_total{code="200",method="get",path="/"} 1.0
# TYPE http_server_request_duration_seconds histogram
# HELP http_server_request_duration_seconds The HTTP response duration of the Rack application.
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.005"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.01"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.025"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.05"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.1"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.25"} 0.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="0.5"} 1.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="1"} 1.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="2.5"} 1.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="5"} 1.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="10"} 1.0
http_server_request_duration_seconds_bucket{method="get",path="/",le="+Inf"} 1.0
http_server_request_duration_seconds_sum{method="get",path="/"} 0.251396
http_server_request_duration_seconds_count{method="get",path="/"} 1.0
# TYPE http_server_exceptions_total counter
# HELP http_server_exceptions_total The total number of exceptions raised by the Rack application.
```

А можно подробнее, что это?

① Prometheus Data Model



Типы метрик

- counter - счетчики (the number of requests served, tasks completed, or errors)
- gauge - число которое может увеличиваться или уменьшаться (temperatures, current memory usage, but also “counts” that can go up and down, like the number of concurrent requests)
- Histogram - histograms, quantiles are calculated on the Prometheus server(request durations or response sizes)
- Summary (сводка, хех) – extended histograms, quantiles are calculated on the application server

Как это может выглядеть в питоне

```
from prometheus_client import Counter
c = Counter('my_failures', 'Description of counter')
c.inc()      # Increment by 1
c.inc(1.6)   # Increment by given value
```

```
@c.count_exceptions()
def f():
    pass

with c.count_exceptions():
    pass

# Count only one type of exception
with c.count_exceptions(ValueError):
    pass
```

Как это может выглядеть в фаст апи

- Есть `prometheus_fastapi_instrumentator`

```
from prometheus_fastapi_instrumentator import Instrumentator
```

Instrument your app with default metrics and expose the metrics:

```
Instrumentator().instrument(app).expose(app)
```

Depending on your code you might have to use the following instead:

```
instrumentator = Instrumentator().instrument(app)

@app.on_event("startup")
async def _startup():
    instrumentator.expose(app)
```

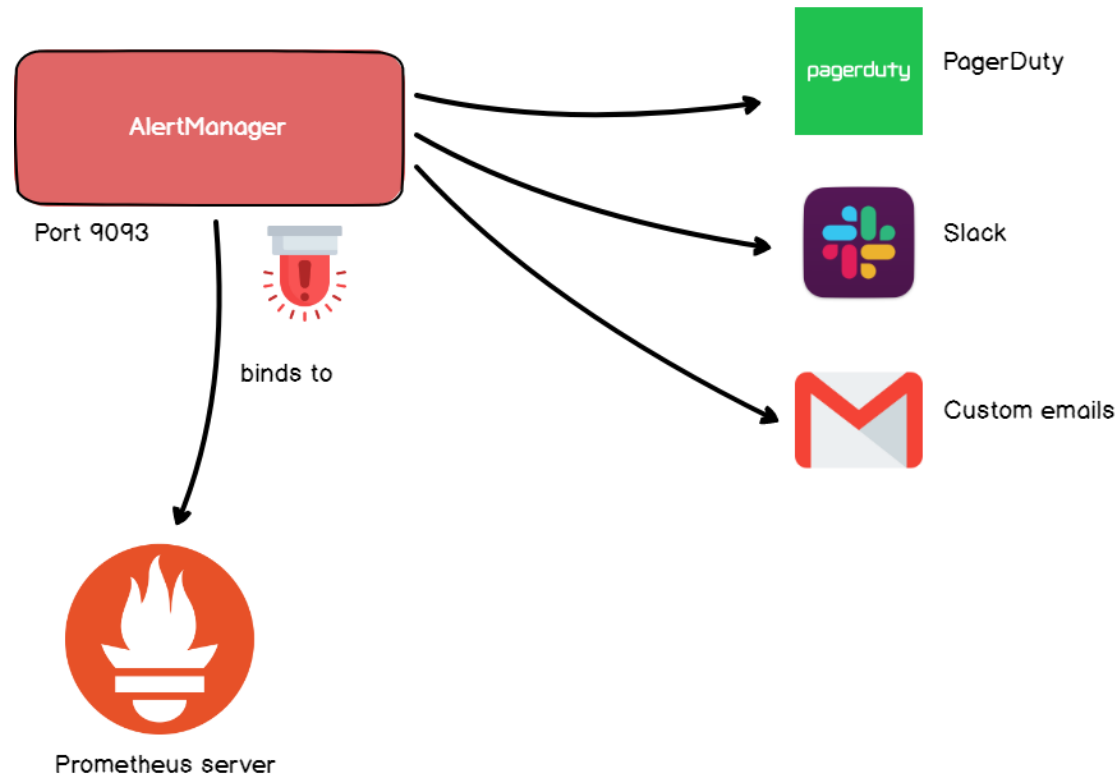
А как получать данные из Prometheus?

Есть PromQL!

- Это свой язык для получения метрик из Prometheus
- Выглядит так же, как метрики, например:
- `http_requests_total{host="10.2.0.4", path="/api"} offset 1d`
- Есть в том числе получение метрик за определенный период, агрегации, свои функции

А как же алерты? Есть свой Alertmanager!

Alerting with Prometheus



Важный моментик

- Записи в Prometheus хранятся (по умолчанию) два месяца
- Да и в принципе Prometheus – не long-term хранилище

Так, отлично, а при чем тут вообще VictoriaMetrics?

- Почти то же самое, но быстрее и требует меньше оперативки
- И есть еще свои фишки

Есть свои:

- vmagent (на клиентах)
- vmalert (замена Alertmanager)
- vmanomaly (для детекции аномалий)
- vmgateway (Pushgateway)
- И еще есть свои сервисы

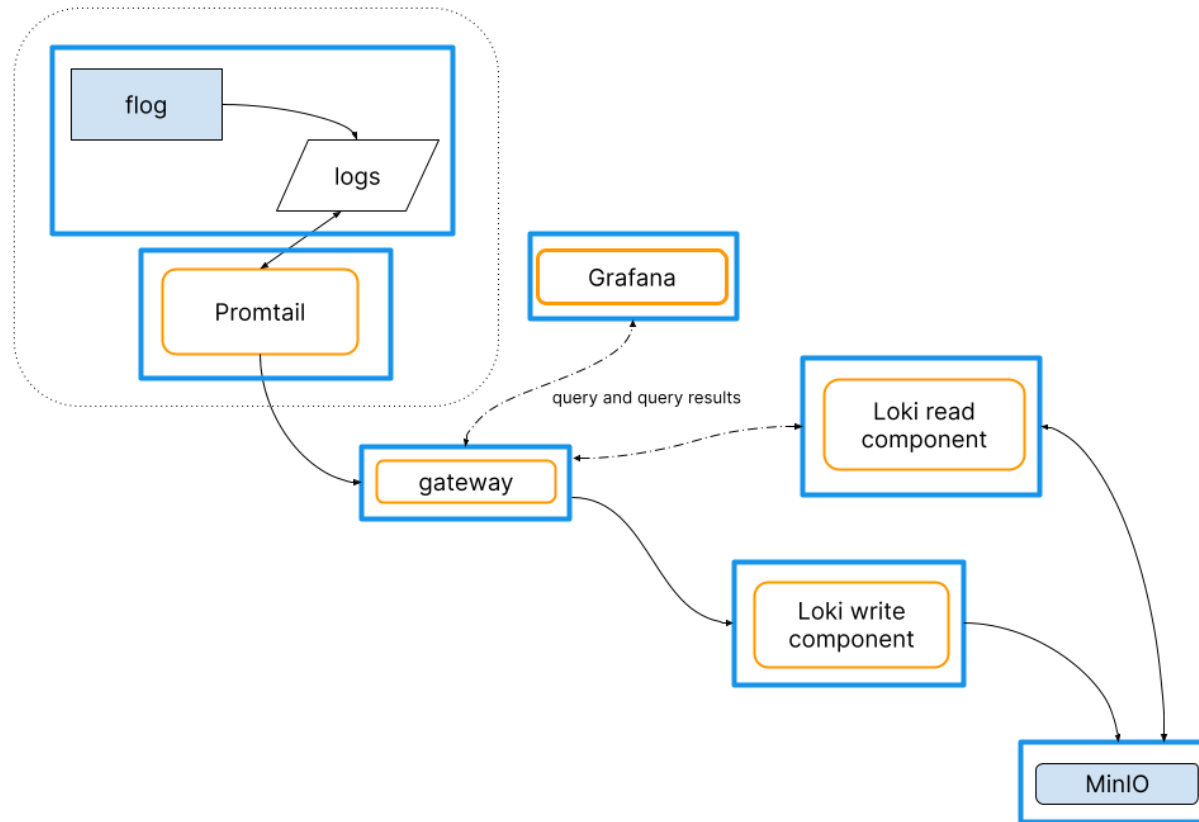


VictoriaMetrics



Prometheus

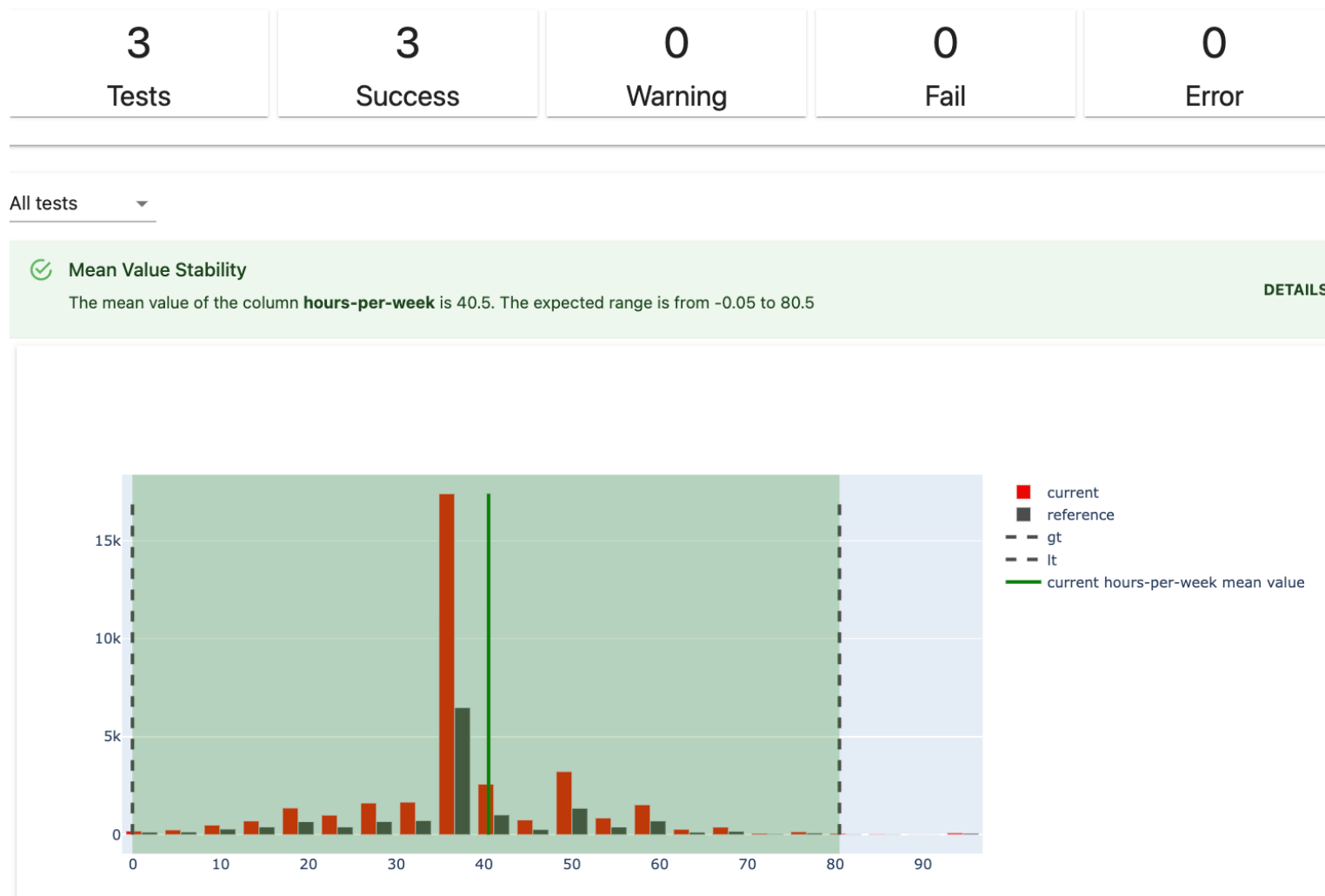
Grafana Loki



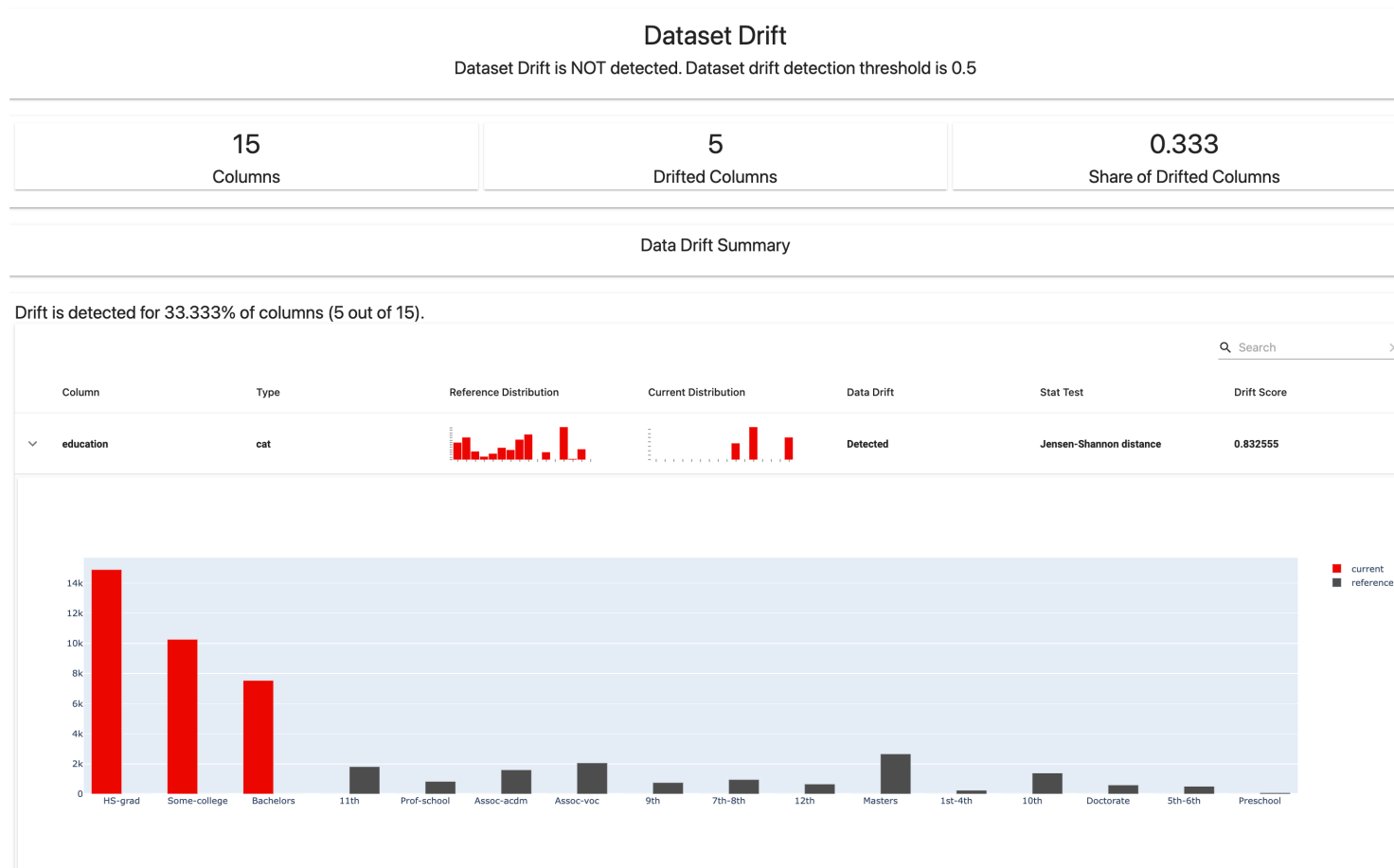
Evidently

- Это фреймворк для вычисления различных метрик из мира ML
- Просто питоновский пакет с набором функций для вычисления метрик
- Есть UI-шка, в которой можно читать репорты их или смотреть дашборд мониторинга

UI (тест)




UI (репорт)



Как это выглядит в коде?

```
data_stability= TestSuite(tests=[
    DataStabilityTestPreset(),
])
data_stability.run(current_data=iris_frame.iloc[:60], reference_data=iris_frame.iloc[60:], column_mapping=None)
data_stability
```

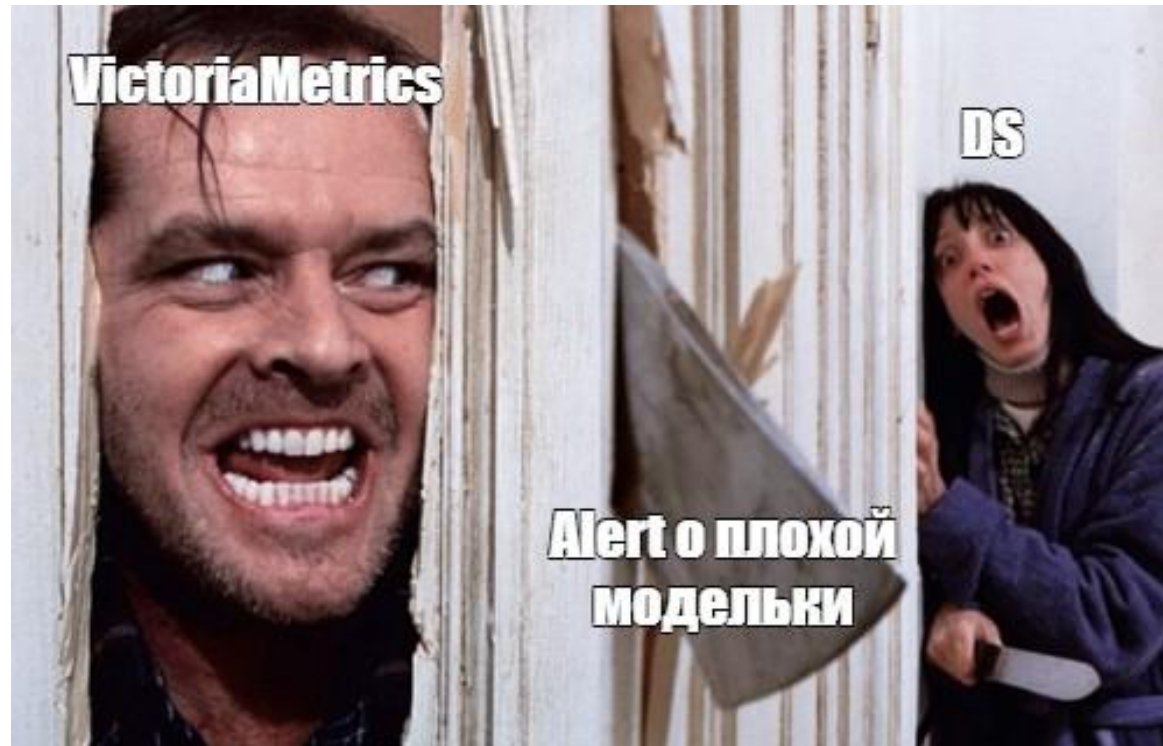


```
data_drift_report = Report(metrics=[
    DataDriftPreset(),
])
```

```
data_drift_report.save_html("file.html")
```


Итак

- Теперь мы знаем, как работает мониторинг
- Можем (с гайдами, конечно) прикрутить его сами
- Мы молодцы



Потоки данных

- В случае online моделей и преобразований данных нам нужно уметь потоково работать с данными
- А как?

Вот тут мы узнаем:

- Что такое брокер сообщений и конкретно kafka
- Потоки данных в Apache Flink
- Есть еще SparkStreaming, тот же спарк, только все время работающий с «потоком» (там микробатчи на самом деле), но возможно там есть еще свои нюансы



Брокер сообщений

- Это та штука, которая обеспечивает доставку сообщений от одного узла до другого (или группы узлов)
- Зачем?
 - Для организации связи между отдельными службами, даже если какая-то из них не работает в данный момент
 - За счёт асинхронной обработки задач можно увеличить производительность системы в целом
 - Для обеспечения надёжности доставки сообщений
 - Все, в общем то
- Самые популярные:
 - Kafka
 - RabbitMQ

Сообщение отправляется напрямую от отправителя к получателю

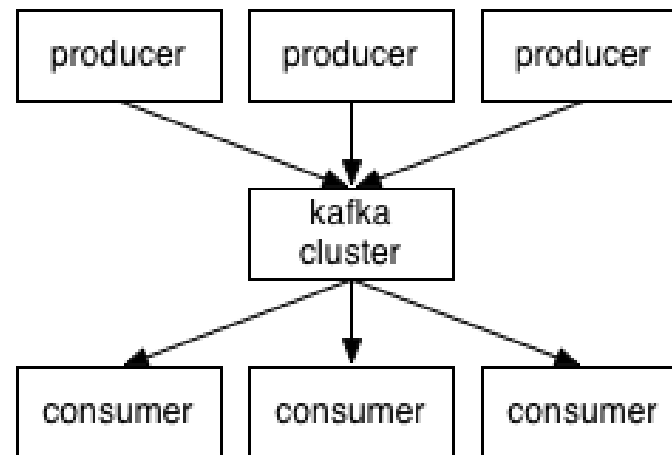


Схема публикации/подписки



Kafka

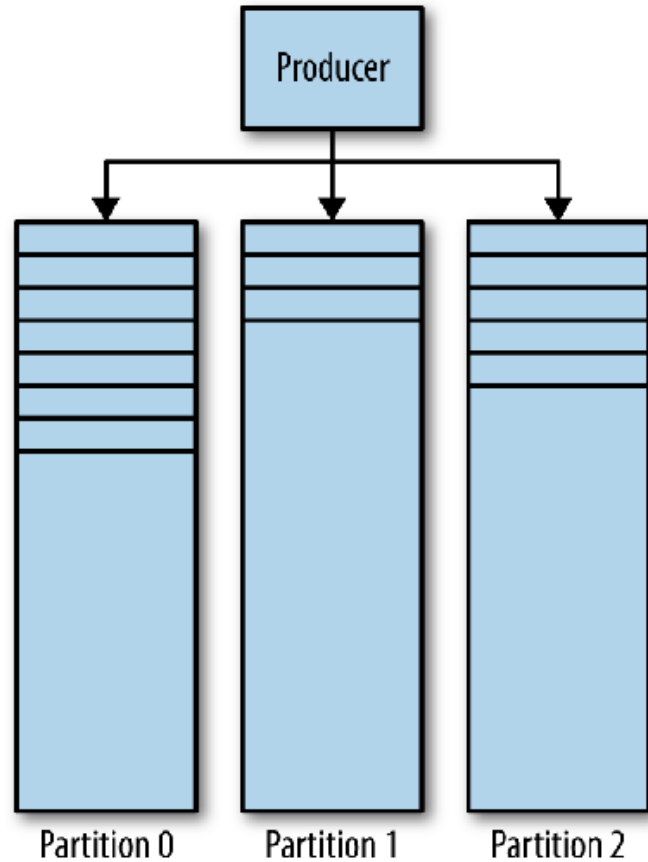
- Приложения (продюсеры, producers) посылают сообщения на узел Kafka (брокер), и указанные сообщения обрабатываются другими приложениями, так называемыми консьюмерами (consumer).
- Указанные сообщения сохраняются в топике , а потребители подписываются на тему для получения новых сообщений.



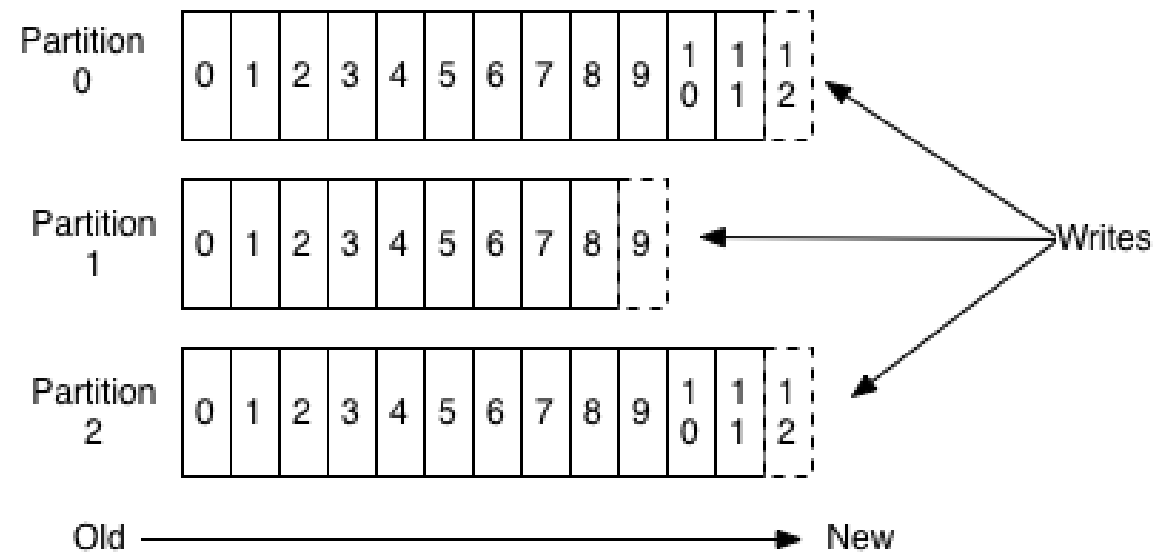
Журнал

- У каждого топика в Kafka есть свой журнал.
- Продюсеры, отправляющие сообщения в Kafka, дописывают в этот журнал, а консюмеры читают из журнала с помощью указателей, которые постоянно перемещаются вперед
- Периодически Kafka удаляет самые старые части журнала
- Брокер не заботится о том, прочитаны ли сообщения или нет — это ответственность клиента.

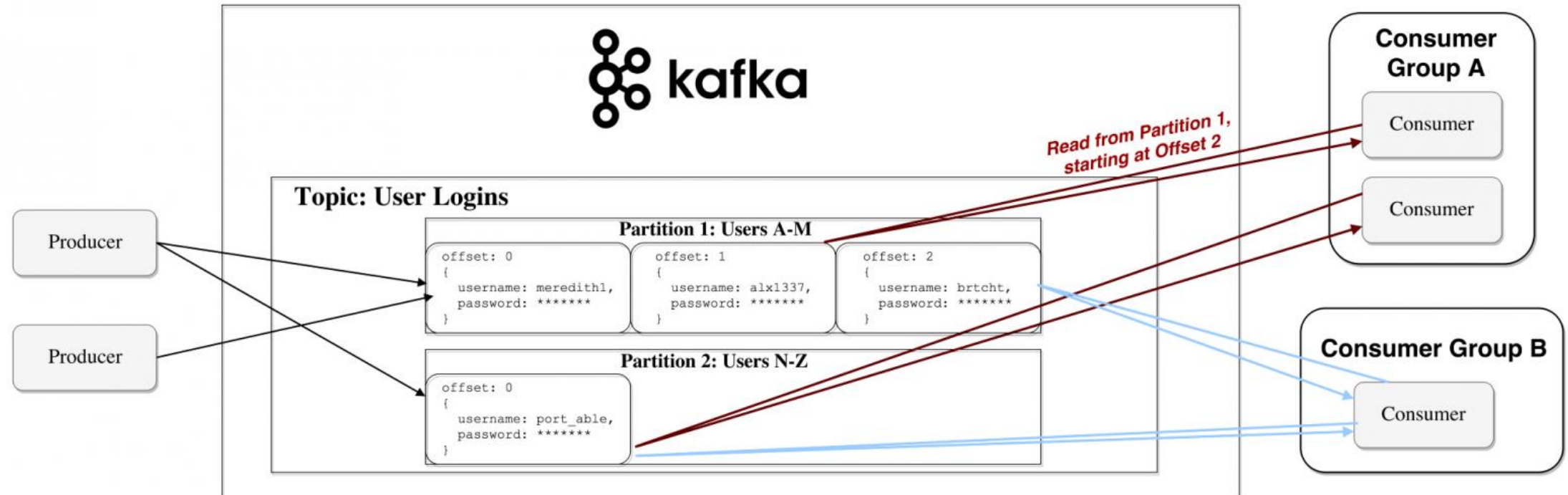
Partitions



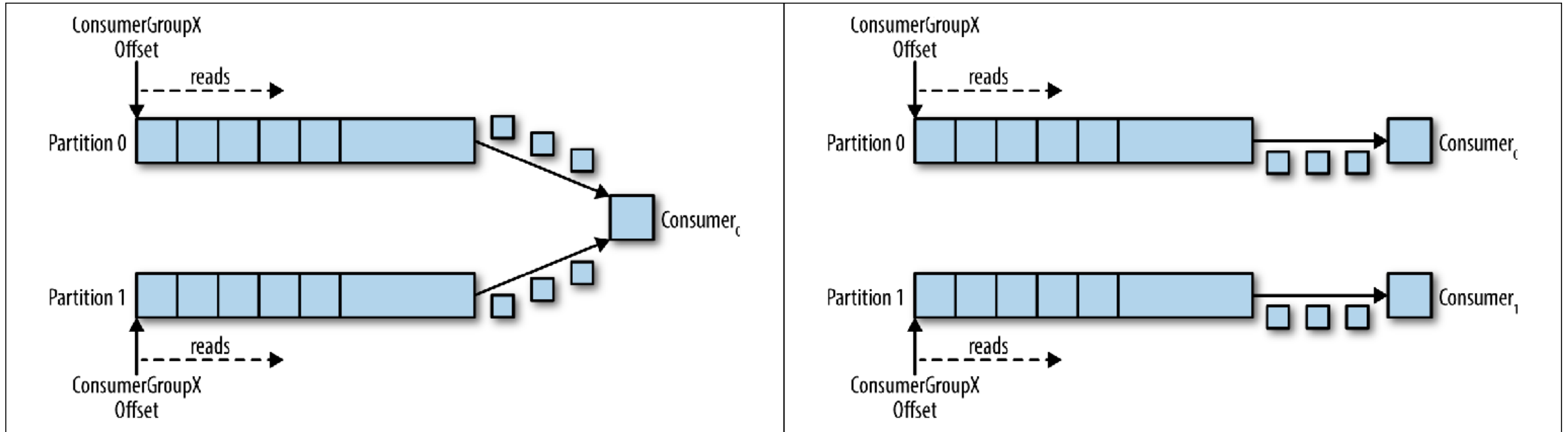
Anatomy of a Topic



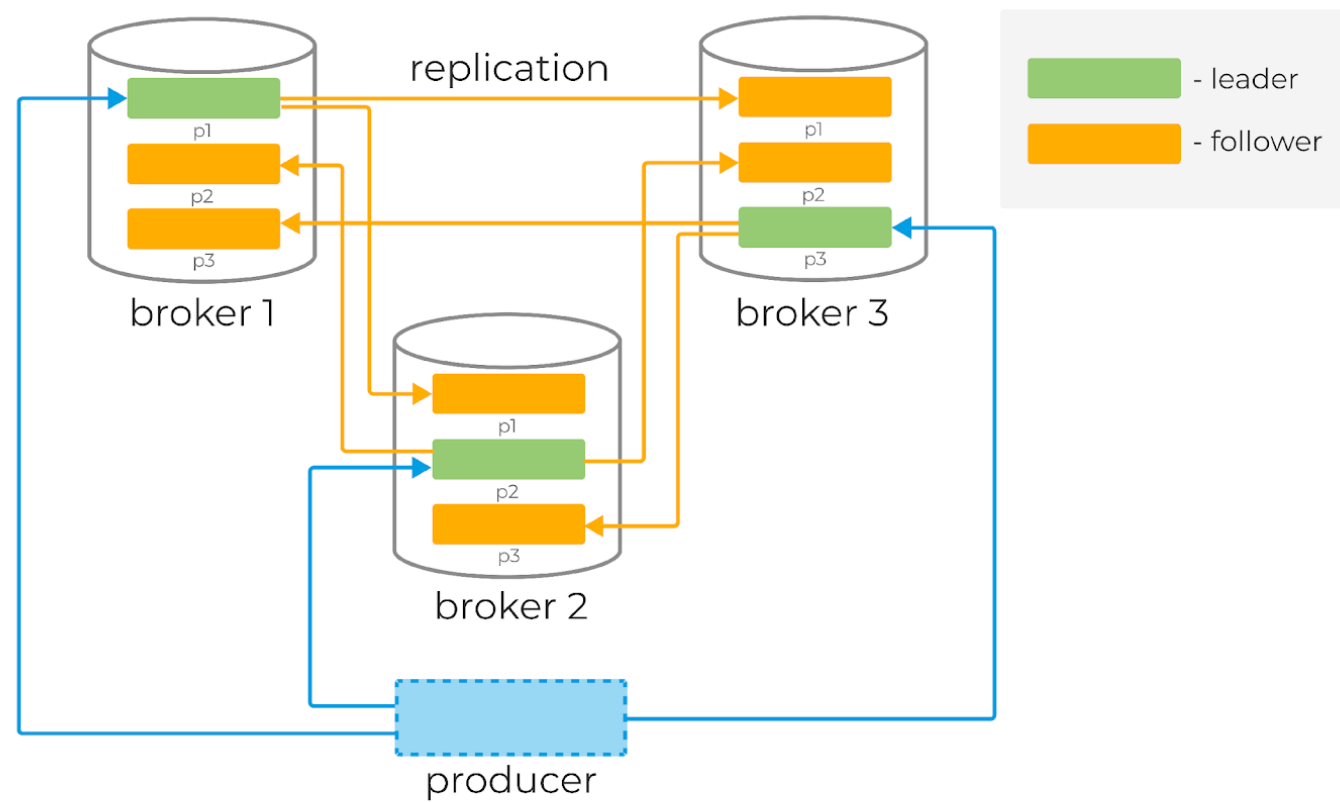
Partitions



Partitions



Лидер и репликация



Zookeeper

- Zookeeper – это распределенное хранилище ключей и значений. Оно сильно оптимизировано для считывания, но записи в нем происходят медленнее. Чаще всего Zookeeper применяется для хранения метаданных и обработки механизмов кластеризации

Kafka Streams и KSQL

- Есть особая штука для работы с потоковыми данными – kafka streams.
- По сути, дают возможность данные из топика обрабатывать потоково через KSQL – SQL язык для преобразования данных
- В мире ML (да и вообще) такое особо не видел, но решил, что как минимум сказать точно стоит.

Flink

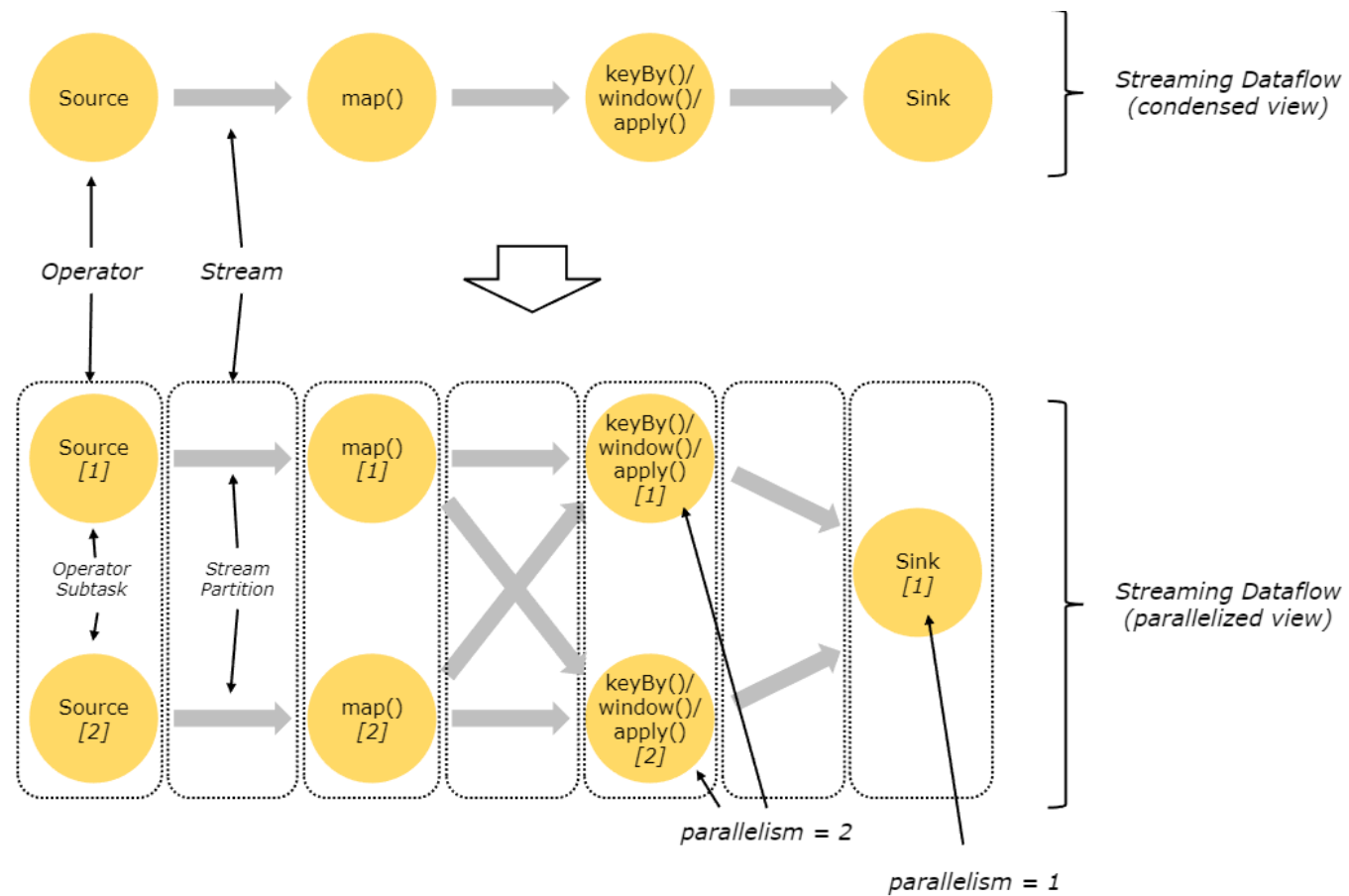
- Kafka дает нам передавать потоково сообщения, а Flink – их потоково (и эффективно) обрабатывать
- Ультра сложная и прикольная штука



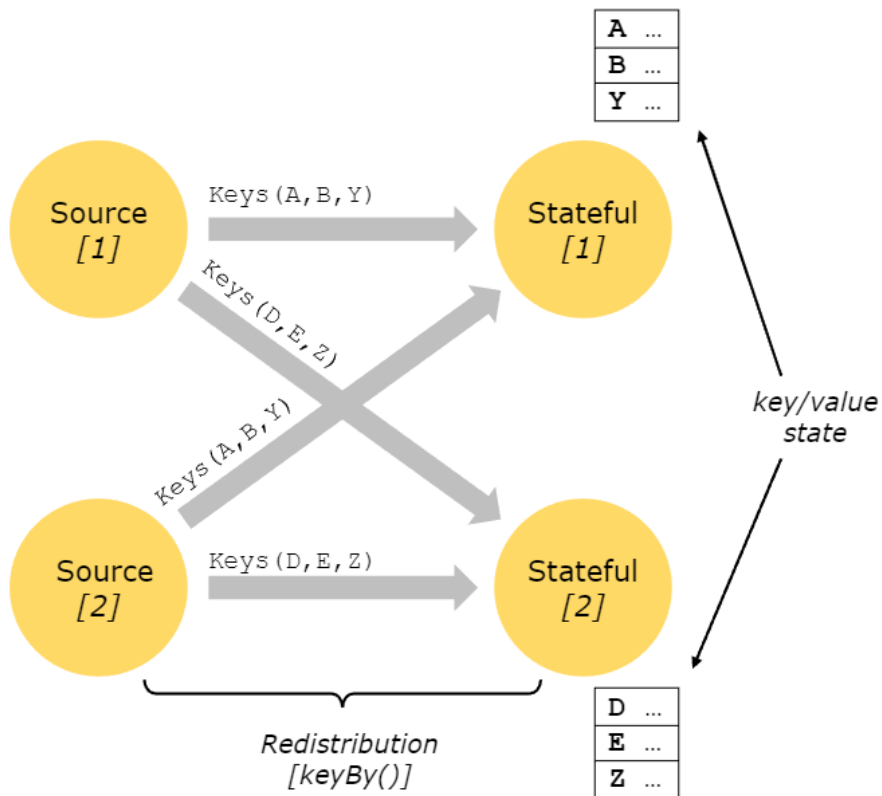
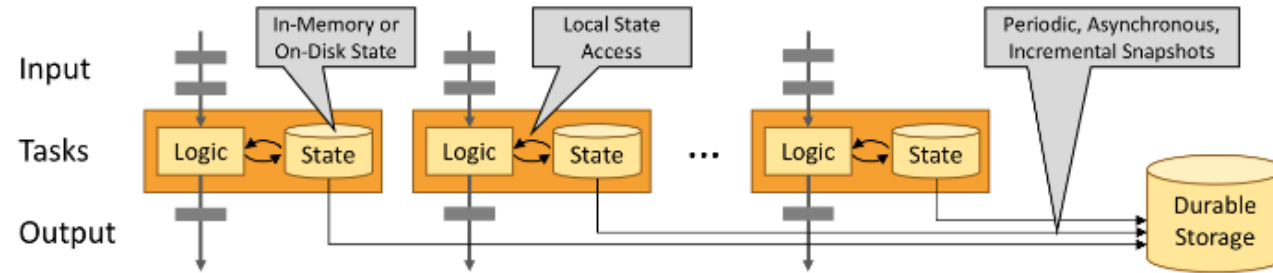
ОСНОВЫ

- Здесь используется трушная стриминговая обработка по сообщениям
- Обработка состоит из операторов, которые легко параллелизируются
- У оператора может быть стейт – по сути, его локальное хранилище, которые контролируется Flink-ом
- Стримы могут распространяться по ключам для эффективности
- Есть таймеры и обработка событий по таймерам

Параллелизация



Стейт по ключу и потоки по ключам



Чекпоинты

- Есть механизм чекпоинтов
- Это когда каждые N секунд или сообщений происходит снапшот — запись всех стейтов в базу
- Есть механизмы синхронизации чекпоинтов по операторам
- Хранение самих стейтов конфигурируется. Есть два стула:
 - Hashmap в памяти
 - RocksDB

Таймеры

- Есть два таймера:
 - Process time – текущее время по часам на стене
 - Event time – время события.
- Wattermark(t) – флаг, показывающий, что все сообщения со временем события меньше t уже обработаны. Используется для синхронизации потоков по event time. Оч важная штука для системы
- Можно регистрировать свои таймеры и по их окончании выполнять какое-то действие
- Есть механизмы обработки окон

Time To Live

- Для сте́йта можно определять ttl – время после определенного события, после которого нужно что-то сделать
- Чаще всего, это используют для очистки сте́йта от опоздавших ивентов после определенной обработки

Process function

- По сути, главный строительный блок потоковой обработки
- Process function – функция, которая применяется для каждого сообщения в потоке
- Есть вариант этой функции на двух потоках (для их совместной обработки)
-

Python API

- Есть PythonAPI для работы как со стримами, так и с таблицами (батчами стримов)
- Сама работа с данными (преобразования) очень похожа на спарк
- Все понятия, что были до этого, так же определяются наследованием от нужных классов или используя объекты из пакета
- Есть некоторые штуки, которые еще не реализованы для PythonAPI

Пример

```
from pyflink.common.typeinfo import Types
from pyflink.datastream import StreamExecutionEnvironment, FlatMapFunction, RuntimeContext
from pyflink.datastream.state import ValueStateDescriptor

class CountWindowAverage(FlatMapFunction):

    def __init__(self):
        self.sum = None

    def open(self, runtime_context: RuntimeContext):
        descriptor = ValueStateDescriptor(
            "average", # the state name
            Types.PICKLED_BYTE_ARRAY() # type information
        )
        self.sum = runtime_context.get_state(descriptor)

    def flat_map(self, value):
        # access the state value
        current_sum = self.sum.value()
        if current_sum is None:
            current_sum = (0, 0)

        # update the count
        current_sum = (current_sum[0] + 1, current_sum[1] + value[1])

        # update the state
        self.sum.update(current_sum)

        # if the count reaches 2, emit the average and clear the state
        if current_sum[0] >= 2:
            self.sum.clear()
            yield value[0], int(current_sum[1] / current_sum[0])

env = StreamExecutionEnvironment.get_execution_environment()
env.from_collection([(1, 3), (1, 5), (1, 7), (1, 4), (1, 2)]) \
    .key_by(lambda row: row[0]) \
    .flat_map(CountWindowAverage()) \
    .print()

env.execute()

# the printed output will be (1,4) and (1,5)
```

Еще пример

```
class CountWithTimeoutFunction(KeyedProcessFunction):

    def __init__(self):
        self.state = None

    def open(self, runtime_context: RuntimeContext):
        self.state = runtime_context.get_state(ValueStateDescriptor(
            "my_state", Types.PICKLED_BYTE_ARRAY()))

    def process_element(self, value, ctx: 'KeyedProcessFunction.Context'):
        # retrieve the current count
        current = self.state.value()
        if current is None:
            current = Row(value.f1, 0, 0)

        # update the state's count
        current[1] += 1

        # set the state's timestamp to the record's assigned event time timestamp
        current[2] = ctx.timestamp()

        # write the state back
        self.state.update(current)

        # schedule the next timer 60 seconds from the current event time
        ctx.timer_service().register_event_time_timer(current[2] + 60000)

    def on_timer(self, timestamp: int, ctx: 'KeyedProcessFunction.OnTimerContext'):
        # get the state for the key that scheduled the timer
        result = self.state.value()

        # check if this is an outdated timer or the latest timer
        if timestamp == result[2] + 60000:
            # emit the state on timeout
            yield result[0], result[1]
```


Flink: и так далее

- Основные концепты я показал
- Штука очень эффективная для потоковой обработки
- Работает (архитектурно) практически как спарк
- Есть больше нюансов, моментов в доке, но смысла в этом для нас всех мало, пока не столкнемся в плотную
- Знание концептов позволит быстро разобраться подробнее
- Но очень маловероятно, что придется писать сложный код на Flink самим

Feature Store

- Feature Store – по сути API для удобной работы с фичами, их преобразованиями, их каталогизацией.
- Короче говоря, удобный инструмент для работы с фичами командно в компании
- Есть разные готовые варианты, одни из лучших бесплатных (open source) – это Horowsworks и Feast

Подробнее (оч круто написано, советую) по ссылке ниже:

- <https://habr.com/ru/company/glowbyte/blog/581458/>

Картиночка с хабра

Широкие витрины

“Одноразовые”
датасеты и фичи

vs.

Feature Store

Общий каталог
фичей-преагрегатов

Сложное ведение
документации

vs.

Работа с данными и
документирование в
одном окне

Нет четкого разделения
ролей участников

vs.

DE и DS по разные
стороны “прилавка”:
первые готовят фичи и
поддерживают работу “магазина”;
вторые — выступают в роли
потребителей

Нерациональное
использование ресурсов

vs.

Благодаря преагрегатам,
процесс перестроения
датасетов занимает меньше
ресурсов

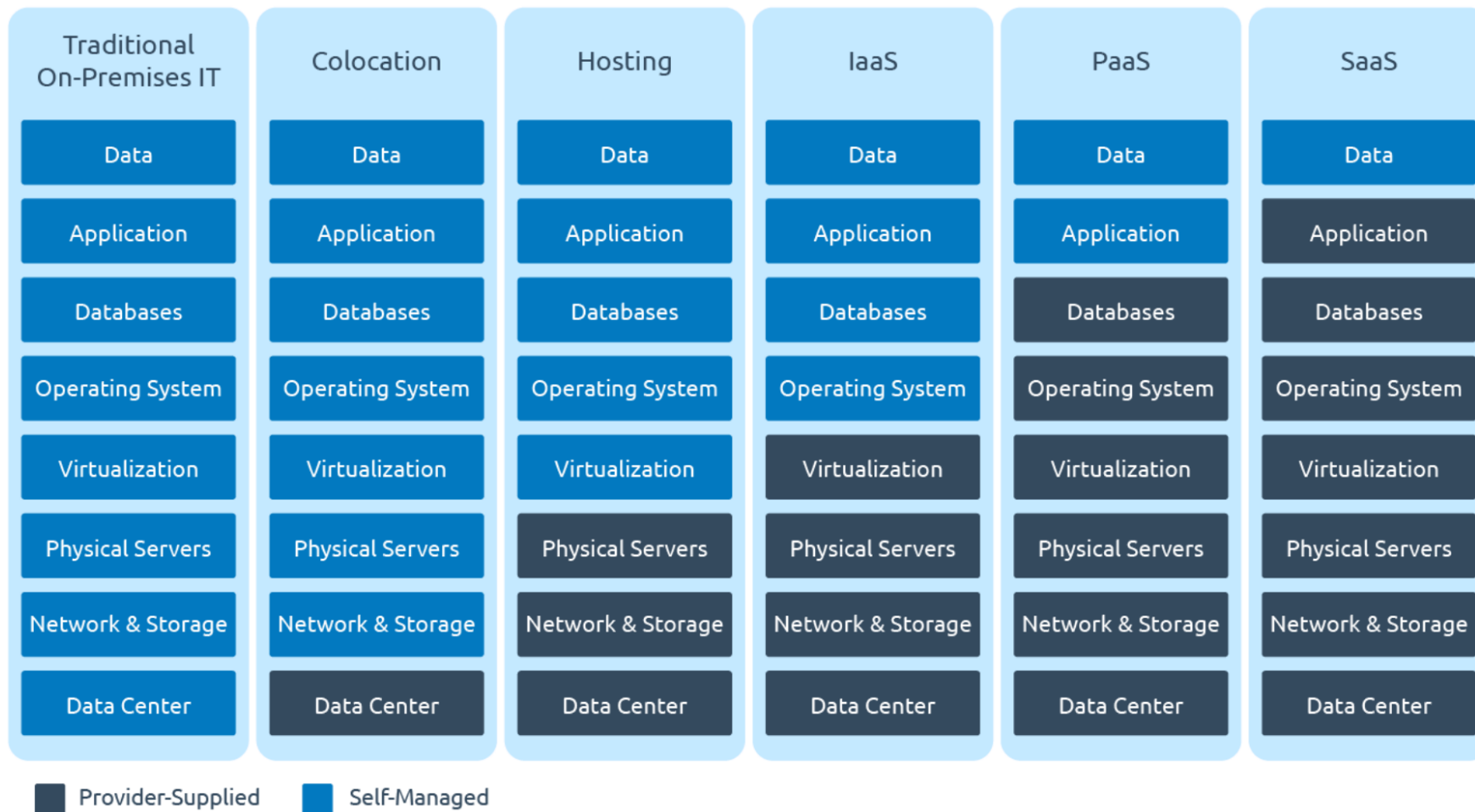
Подробнее расскажу про FS на след паре



Облачка и облачные сервисы

- Для деплоя ваших приложений могут использоваться облачные сервисы
- Они бывают очень разные, от предоставления тупого железа до автоматического выделения мощностей для ваших функций

В чем разница одной картинкой



Traditional On-Premises IT, Collocation

- Плюсы
 - Управляем всей инфраструктурой (или большей частью)
 - Можем оптимизировать стоимость, если наши запросы очень велики (нам требуются сотни машин, купить GPU для исследований может быть дешевле, чем арендовать)
- Минусы
 - Требуется отдельный штат специалистов, занимающихся поддержкой инфраструктуры (сеть, питание, вентиляция, обслуживание помещений)
 - Слишком сложно и дорого для небольших компаний

Hosting - аренда физических серверов

- Плюсы
 - Полноценная удаленная машина
 - Обслуживанием железа занимается хостер
 - Отсутствие виртуализации
 - необходимость для некоторого софта
 - нулевой оверхед
- Минусы
 - Все ещё сложно поддерживать (с программной стороны) - требуется DevOps
 - При небольшой нагрузке возникает соблазн поддерживать самому, что приводит к росту технического долга.

Infrastructure-as-a-Service (IaaS) - AWS EC2, GCE

- Плюсы
 - Виртуализированные машинки
 - Можно быстро “накатывать” требуемые образы
 - Более гибко, чем Hosting, - часть машин можно временно отключить и сэкономить
 - Можно легко масштабировать мощности
 - Часто дефолтный вариант для компаний
- Минусы
 - Машины все еще остаются независимыми машинами
 - Требуется выстраивания процессов деплоя и мониторинга

PS: виртуализацию полезно представлять как некоторую абстракцию на реальном железе

Platform-as-a-Service (PaaS)

- Плюсы
 - Не нужно думать о инфраструктуре, обычно задача лишь в том, чтобы поместить приложение в контейнер,
 - Легче масштабировать, адаптируясь к нагрузке
- Минусы
 - Обычно дороже, чем аренда серверов

Serverless (AWS Lambda, Google Cloud Function)

- Плюсы
 - Не нужно думать о инфраструктуре,
 - На лету адаптируются к меняющейся нагрузке (autoscaling)
- Минусы
 - Нужно подстраиваться под ограничения на среду исполнения
 - Неудобно делать inference на GPU - прогоняются по одному примеру вместо батча.
 - Оплата не за время использования вычислительных мощностей, а за количество вызовов
 - Становится очень дорого на масштабе

Интересные облачные сервисы

- Heroku (paas)
- AWS (всё)
- Google Cloud (всё)
- Azure (всё)
- Selectel (сервер)
- Hetzner (сервер)
- VastAI (гпу)

РО ССИ ЯЯЯ:

- Sbercloud (всё)
- Yandexcloud (всё)
- VK Cloud (всё)