Эксплуатация веб-проектов :: мониторинг

Николай Сивко

okmeter.io

okmeter.io

- Сервис мониторинга
- Фокус на подробные метрики
- Автоконфигурация
- Встроенные пополняемый набор автотриггеров на типичные проблемы

Основные задачи мониторинга

- Узнавать, что что-то сломалось от мониторинга, а не от клиентов

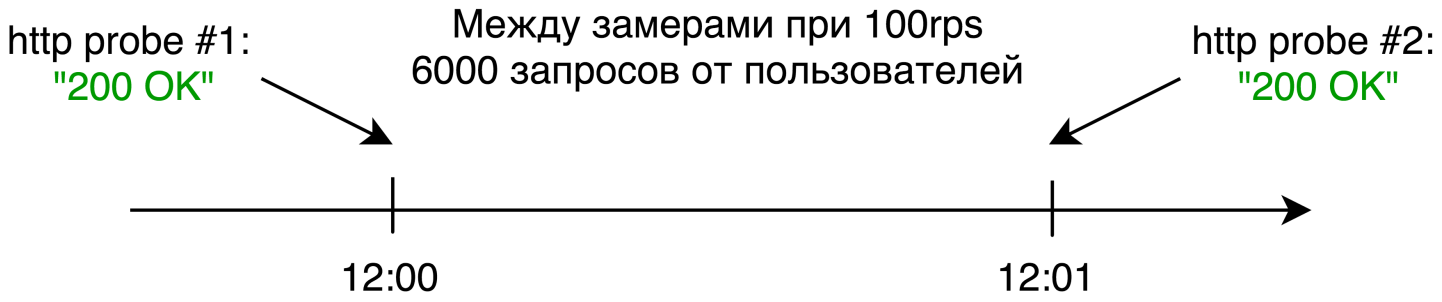
- Понижать MTTR (mean time to recovery)

  TTR = время понимание причины + время исправления

Узнать о проблеме

Регулярные проверки нужных сценариев

+ Легко сделать, есть куча сервисов
+ Дешево/бесплатно
- Большая погрешность
- Покрытие



okmeter

Узнать о проблеме

Логи

++++ Видим все запросы РЕАЛЬНЫХ пользователей

+ Ошибки, тайминги

- Тратим ресурсы на парсинг

- Времена считаем по версии сервера

- Сложно обрабатывать кейс, когда пропали запросы совсем (канал, домен итд)

Парсим логи nginx

Запарились над автоконфигурацией:

- Агент видит процесс nginx
- Читаем конфиг, знаем все access_log’и и их формат
- Вжух! И мы уже парсим лог

Парсим логи nginx

Но у среднего клиента нет в логе таймеров :(

Зажигаем ему алерт, чтобы расширил формат ибо это MUST

Нужно добавить как минимум:

- **$request_time**
- **$upstream_response_time**
- **$upstream_cache_status**

Что напарсили

{

name="nginx.requests.rate"

file="/log_path.log”

plugin=“nginx”

source_hostname=“front1”

method="X"

status="YYY"

cache_status="zzz"

url="/xxx”

}

okmeter

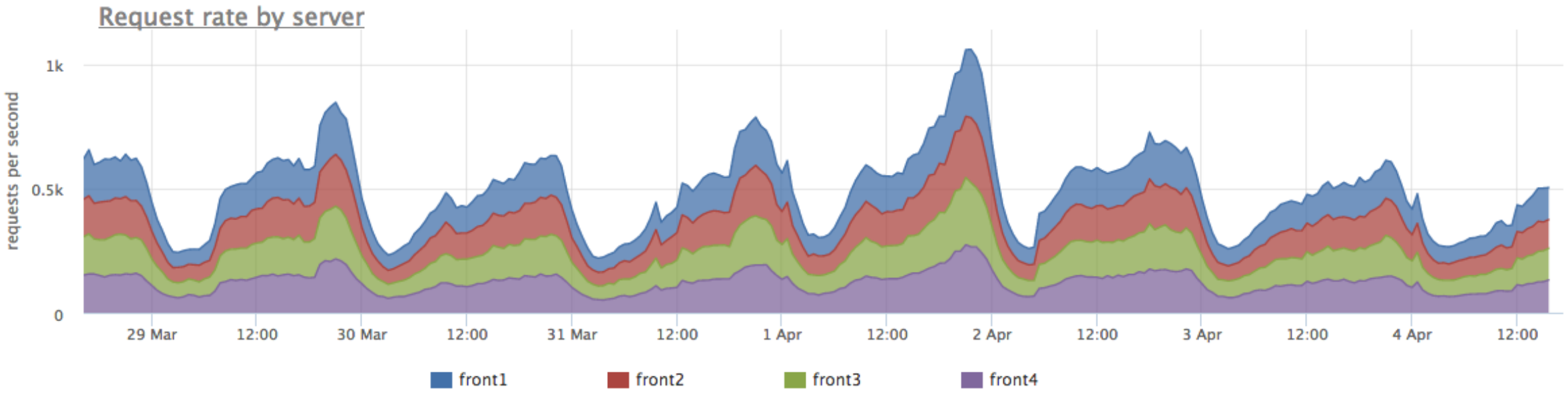
Что напарсили: урлы

URL естественно не полностью

- Выкинули аргументы
- Выкинули все, что похоже на идентификаторы
- Если то, что осталось по количеству запросов > N% - делаем отдельную метрику
- Если нет, суммируем все в url="**~other**"

nginx.requests.rate

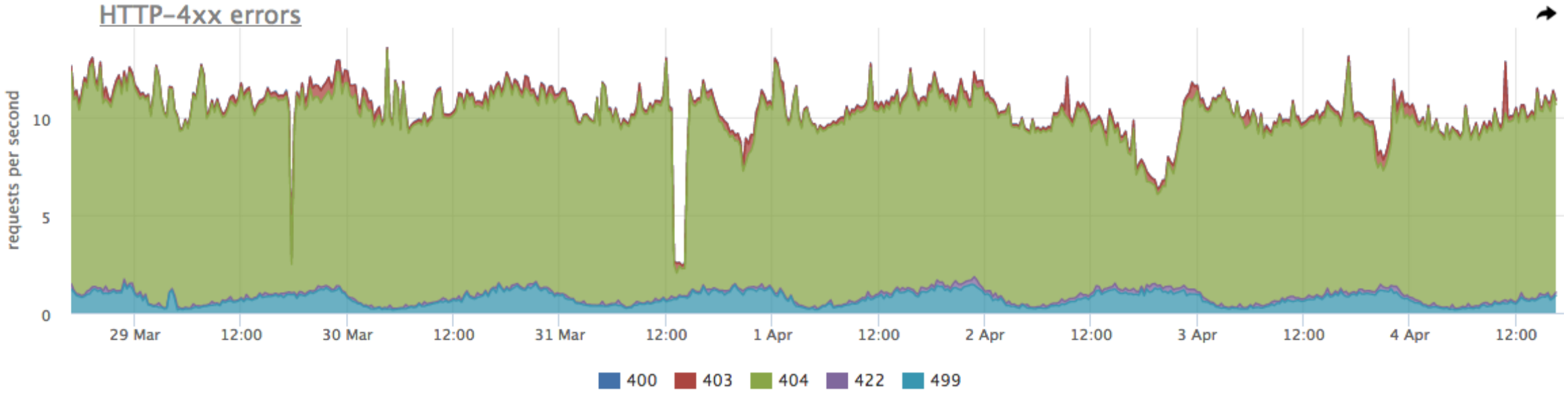
sum_by(**source_hostname**, metric(name="nginx.requests.rate"))



okmeter

nginx.requests.rate

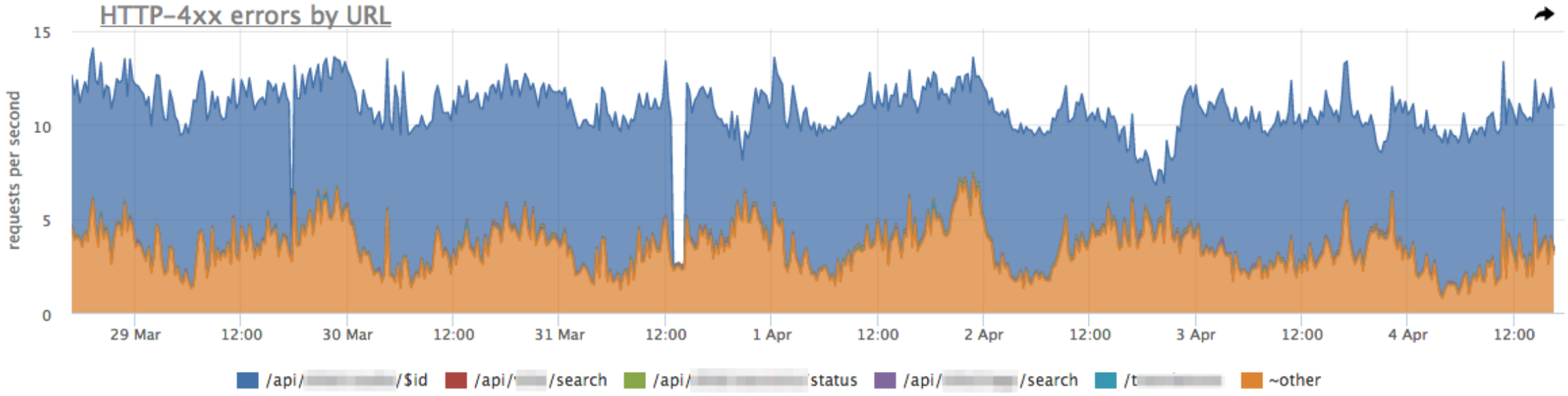
sum_by(**status**, metric(name="nginx.requests.rate", status="**4***"))



okmeter

nginx.requests.rate

top(5, sum_by(**url**, metric(name="nginx.requests.rate", status="**4***")))



okmeter

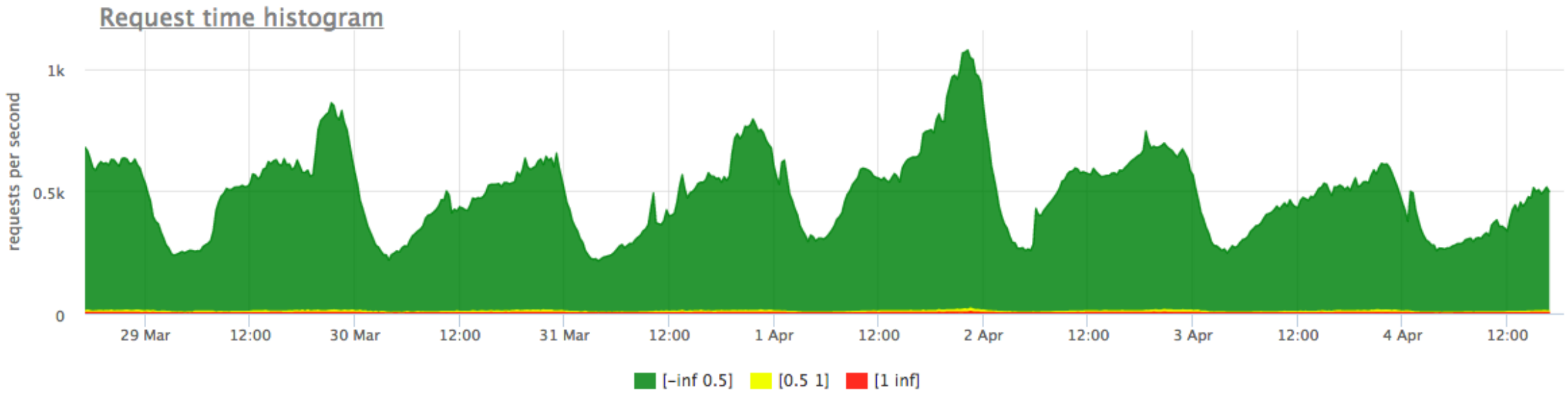
Тайминги: гистограмма

```
{
        name="nginx.response_time.histogram"
        file="/log_path.log”
        plugin=“nginx”
        source_hostname=“front1”
        method="X"
        cache_status="zzz"
        url="/xxx”
        level=”ll”
}
```

okmeter

nginx.response_time.histogram

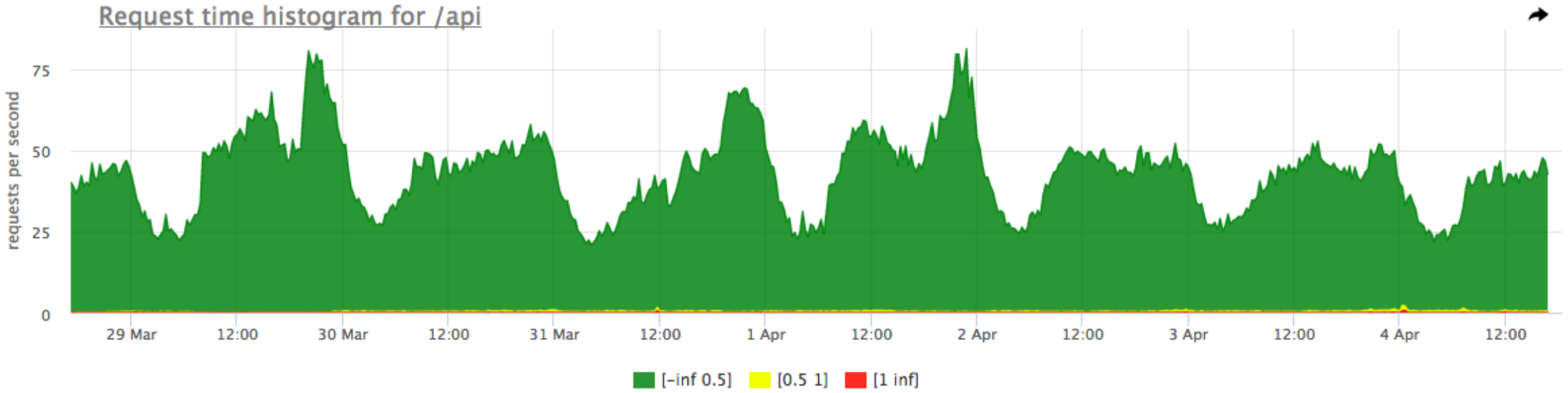
sum_by(level, metric(name="nginx.response_time.histogram"))



okmeter

nginx.response_time.histogram

sum_by(level, metric(name="nginx.response_time.histogram", url="**/api***"))



okmeter

$request_time VS $upstream_response_time

- Нужно смотреть на обе метрики
- На $request_time видно и тупняки бэкенда и тупняки канала до клиента и время бана при достижении limit_req+burst
- На $upstream_response_time видно только тупняки бэкенда (или ложные тупняки бэкенда, если ioloop nginx где-то блокируется)

Общие принципы мониторинга SOA

- Что пользователь думает про фронтенд?
- Что фронтенд думает про бэкенд?
- Что бэкенд думает про соседние сервисы?
- Что бэкенд думает про БД?
- Что БД думает про запросы от бэкенда?
- Как OS видит потребление ресурсов от приложений/БД?

Все врут

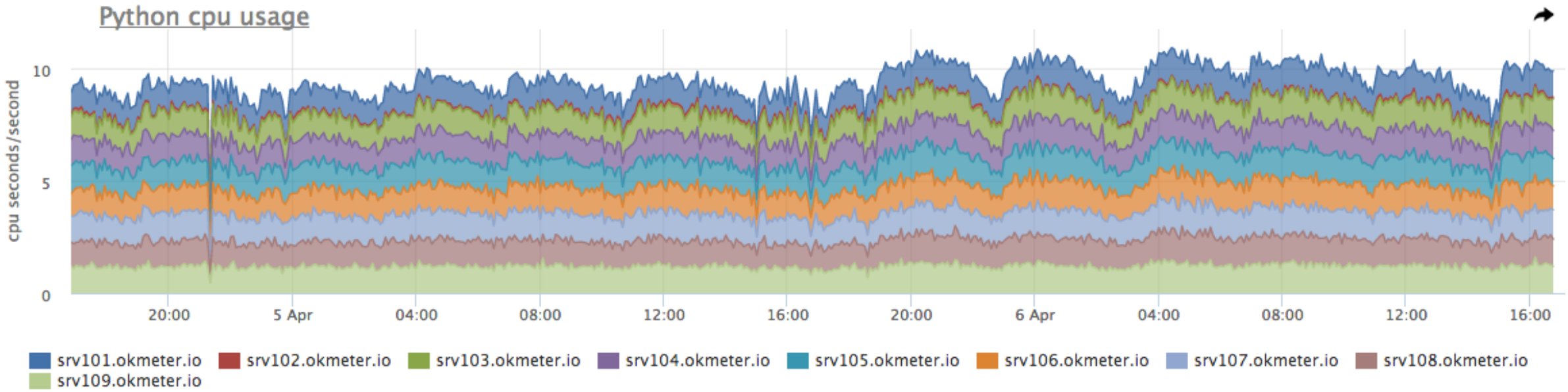
- Никому не верим
- Асинхронный http клиент может заблокироваться и насчитать неправильное время ответа сервера
- Считаем на всех уровнях, сравниваем, исследуем несостыковки

Backend

- Использование ресурсов: cpu/mem/disk io/fd для процессов приложения
- Исчерпание ограниченных ресурсов: open files limit, tcp ack backlog limit итд
- Метрики runtime: gc, memory pools, workers
- Инструментируем код: statsd,*-metrics, pinba

Backend: cpu

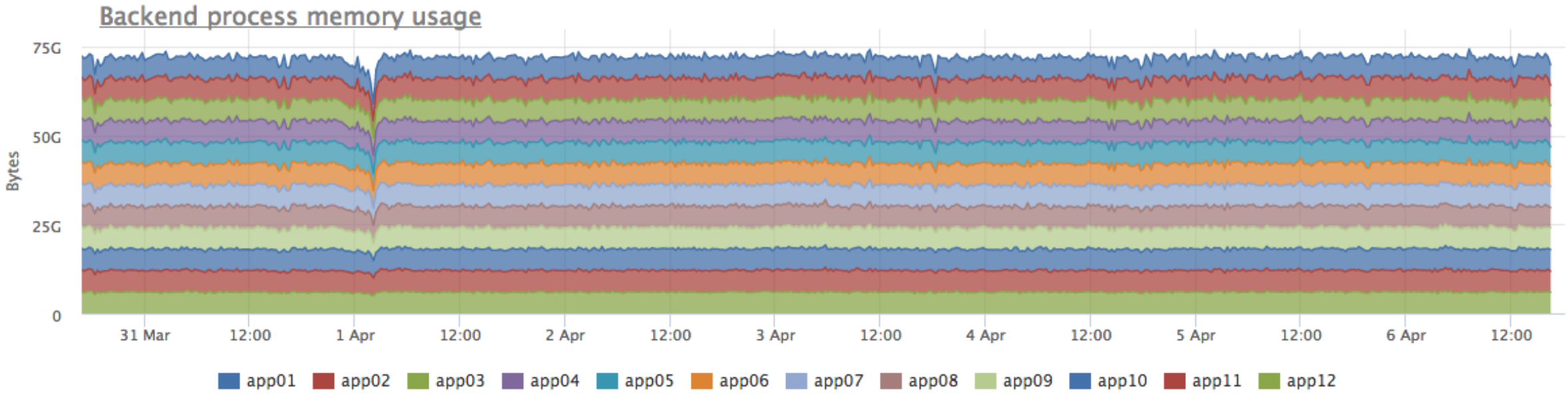
sum_by(source_hostname, metric(name="process.cpu.*", process="**/usr/bin/python**"))



okmeter

Backend: memory

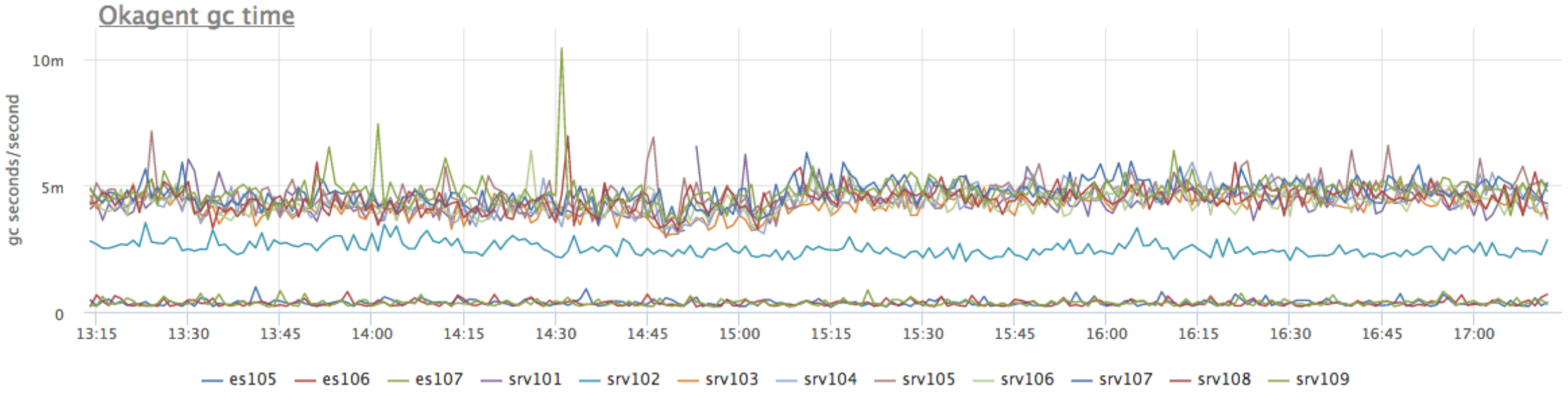
sum_by(source_hostname, metric(name="process.mem.rss", process="**/usr/bin/python**"))



okmeter

Backend: runtime

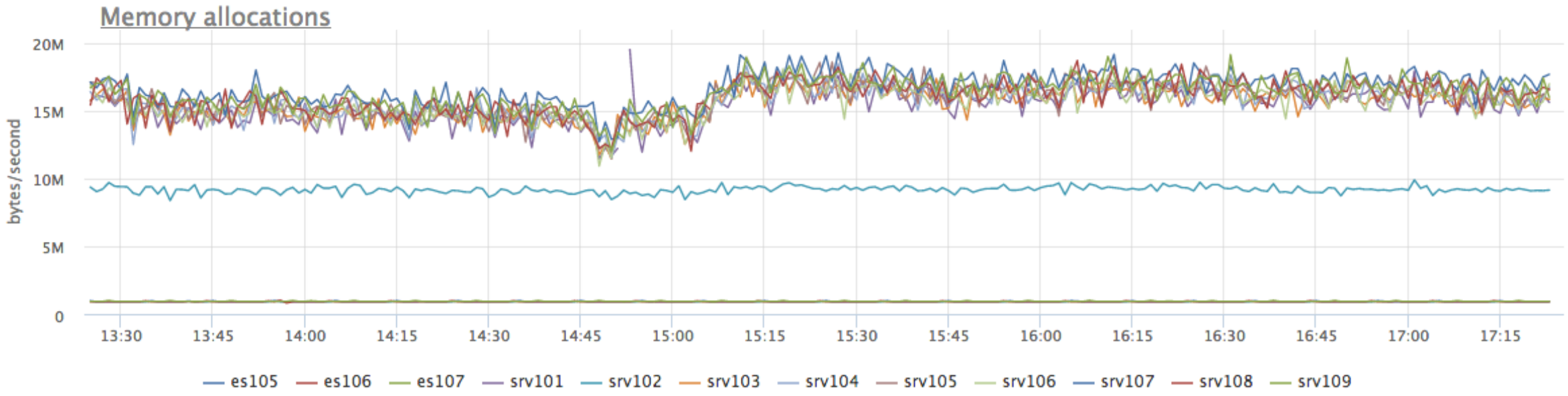
counter_rate(metric(name="okagent.gc.time", source_hostname=["srv10*", "es10*"]))



okmeter

Backend: runtime

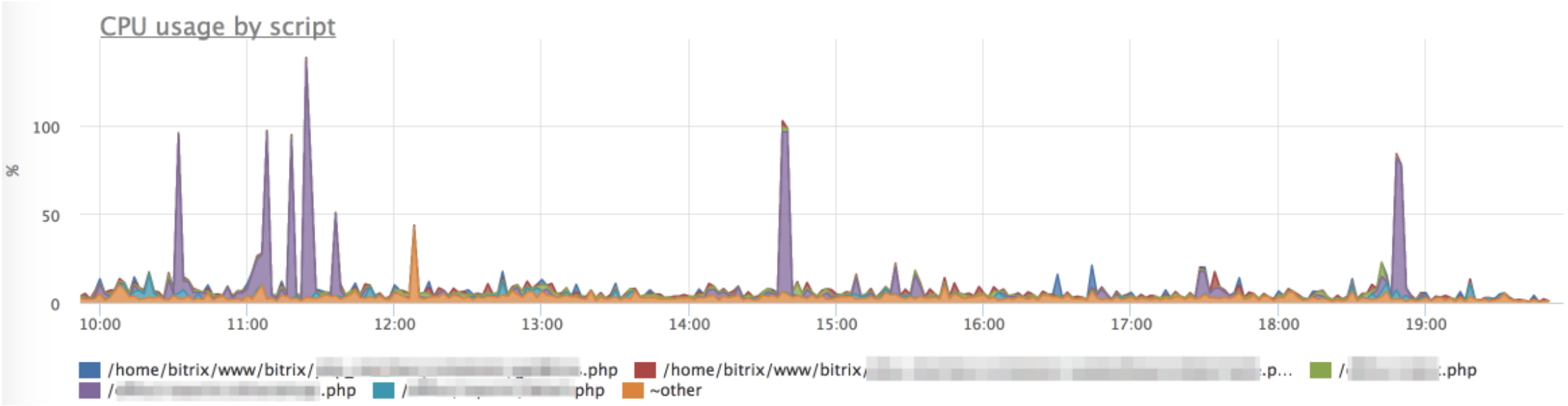
counter_rate(metric(name="okagent.memstats.**total_alloc**", source_hostname=["srv10*", "es10*"]))



okmeter

Backend: инструментируем

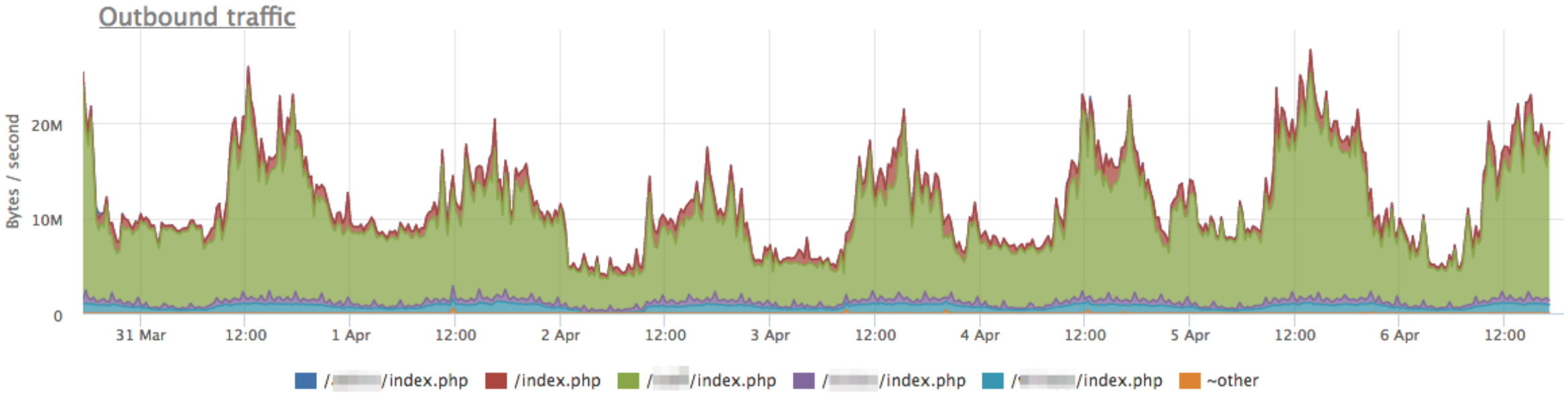
top(5, sum_by(**script**, metric(name="pinba.cpu.time")))



okmeter

Backend: инструментируем

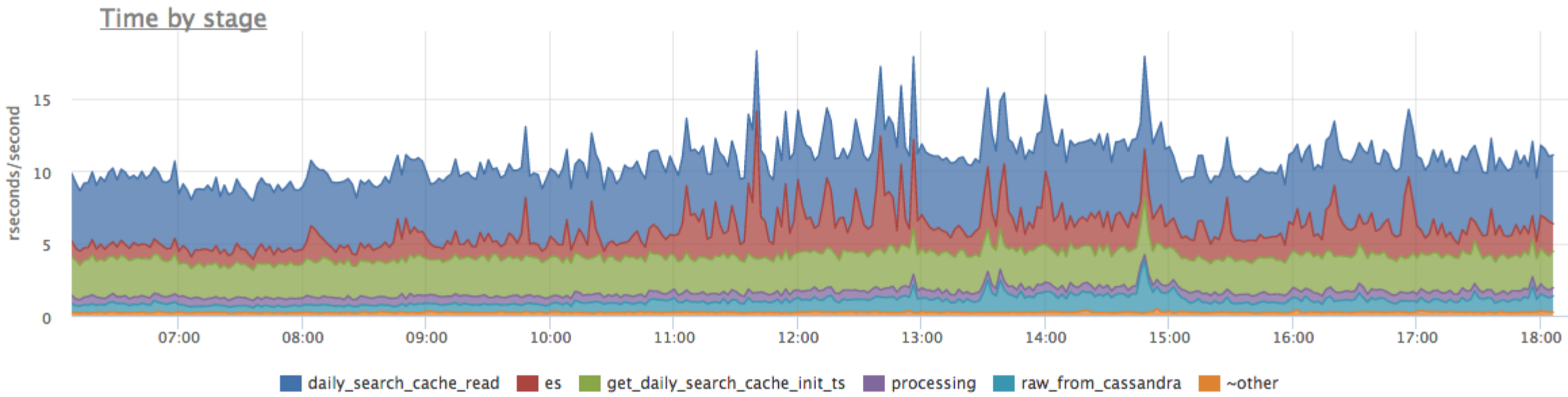
top(5, sum_by(**script**, metric(name="pinba.traffic.rate")))



okmeter

Backend: инструментируем

top(5, sum_by(stage, metric(name="gokserver.handler_stages.sum", handler="**/metric/query**")))



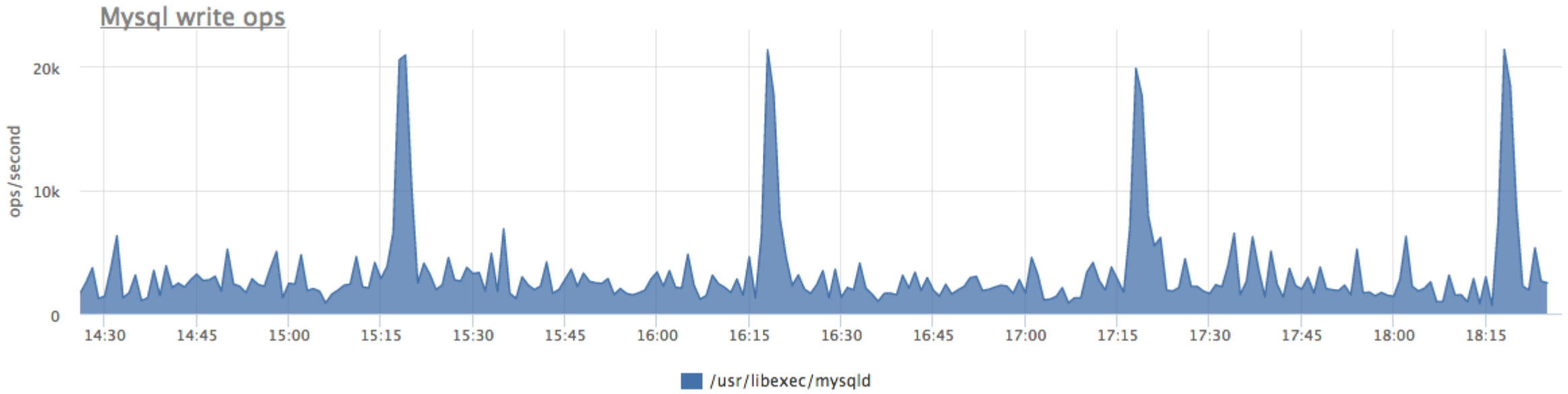
okmeter

БД

- Потребление ресурсов базой с точки зрения OS (cpu/mem/disk io/net,…)
- Про ресурсы хорошо бы проверить, что их не стало меньше
- Идеально разложить это потребление по запросам

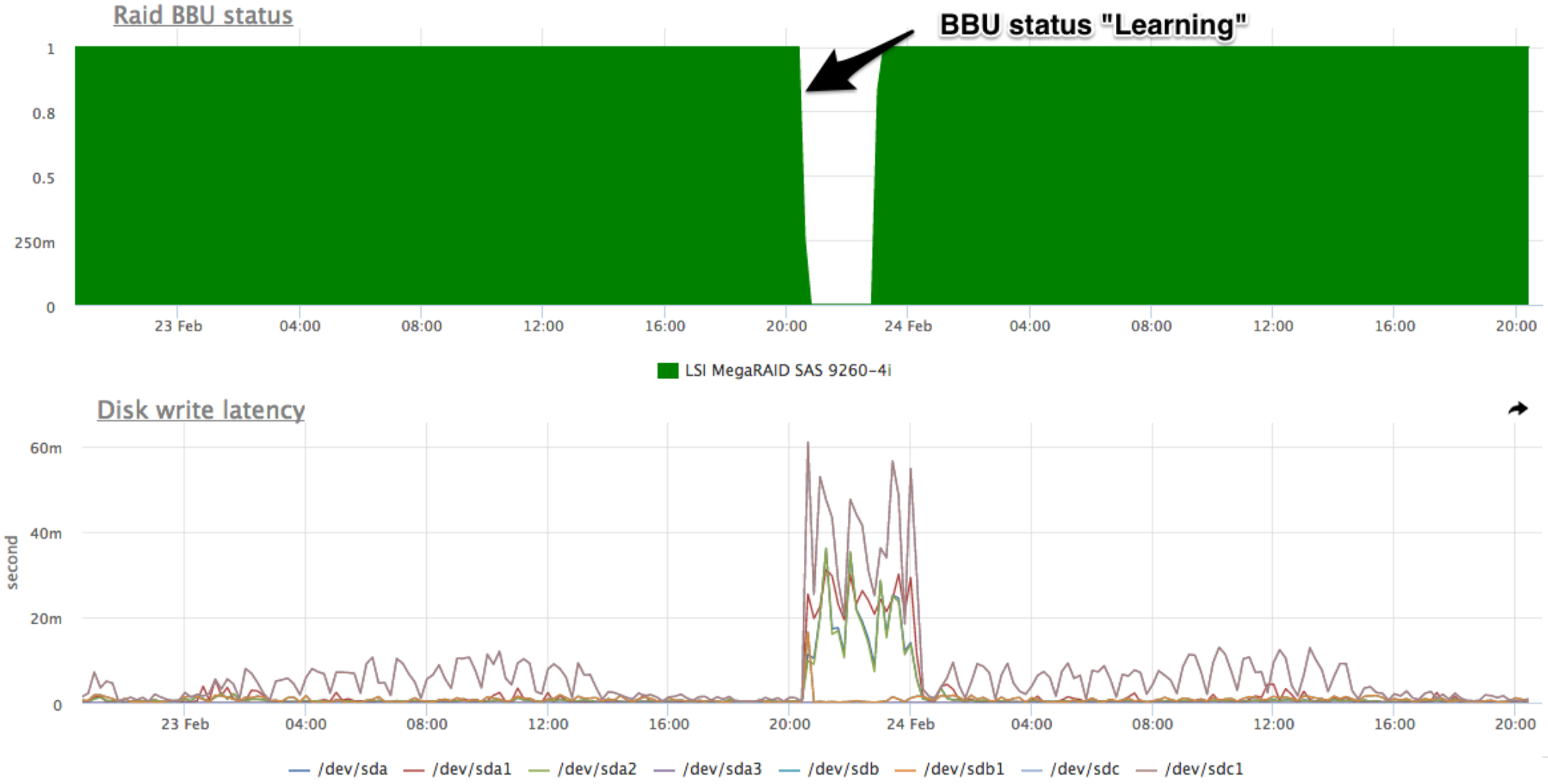
БД: ресурсы

- sum_by(process, metric(name="process.disk.ops.write", process="**/usr/libexec/mysqld**"))



okmeter

БД: деградация по ресурсам



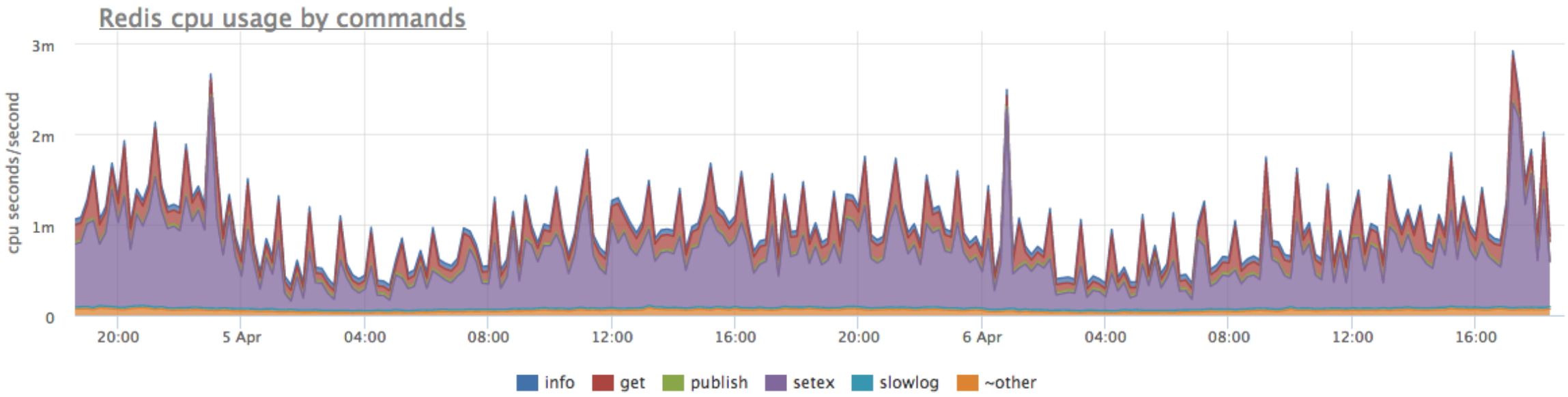
okmeter

БД: ресурсы по запросам

- **PostreSQL**: pg_stat_statements покажет cpu*/disk io*/net* по разным **типам** запросов
- **MySQL**: performance schema, но все сложнее, только свежие версии, иногда дороговатенько
- **Redis**: покажет cpu в разрезе команд
- **Cassandra**: время по запросам к таблицам, а так как 1 таблица = 1 тип запроса, достаточно

Redis: ресурсы по запросам

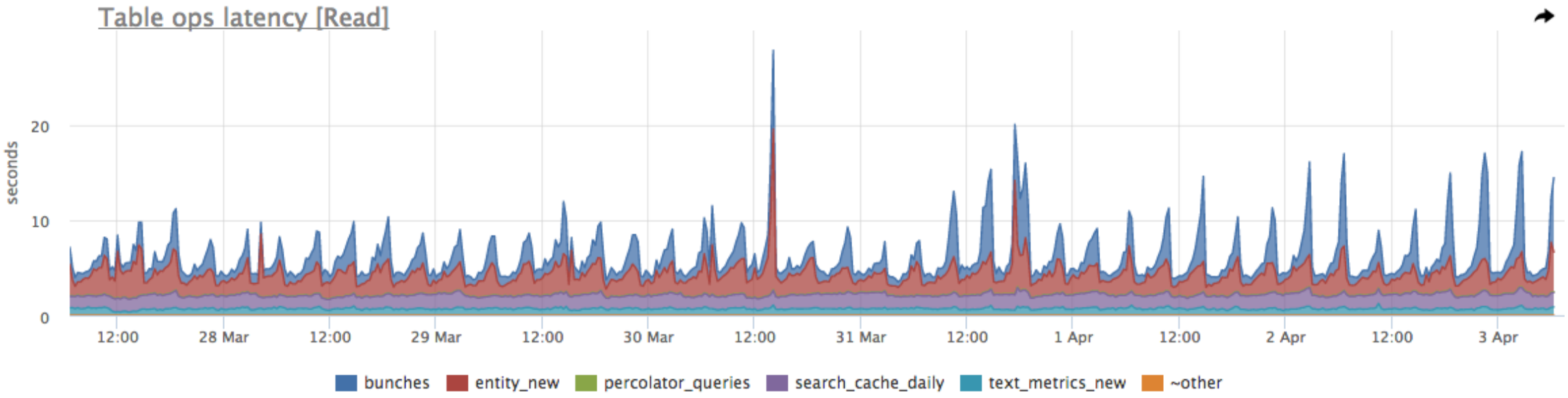
top(5, sum_by(command, counter_rate(metric(name="redis.commands.seconds"))))



okmeter

Cassandra: ресурсы по запросам

top(5, sum_by(**table**, counter_rate(metric(name="cassandra.table.ops.total_time", operation="Read"))))



okmeter

Алерты

- Правильные severity
- Работа с инцидентами

Алерты: правильные severity

- CRITICAL: уведомляем по SMS/IM
- WARNING: можно уведомлять на email или без уведомления
- INFO: без уведомления

Алерты: CRITICAL

- Сайт не работает
- Ошибки приема платежей
- Пропал поток покупок

Алерты: CRITICAL

- Бросаем все и чиним
- Починку нельзя отложить
- Никто никуда не уходит пока есть проблема

Алерты: WARNING

- Диск кончается
- Какой-то сервис недоступен, дает много ошибок или тупит
- Много ошибок на сетевом интерфейсе
- **Сервер недоступен (ДА, это реально warn!!!)**
- Swap IO процесса > 1 Mb/s

Алерты: WARNING

- Желательно закрыть в течении дня
- Если алерт не по делу – добавляем исключение
- Если часто флапает - сглаживаем

Алерты: INFO

- CPU usage > 99%
- Disk IO > 99%
- Использование других ресурсов

Алерты: INFO

- Используем как подсказки во время поиска причин CRITICAL или WARNING

- Если загораются бессмысленные алерты – добавляем исключения

Алерты: общие принципы

- Алерты должны показывать причину проблем
- Зависимости и автомагия не нужны
- Глазами можно быстро просмотреть 100+ алертов и выбрать подходящий
- Алертам, которые не помогают, можно понизить severity

Алерты: работа с инцидентами

- Важно считать продолжительность CRITICAL (он же **downtime**)
- Если critical != downtime (чиним severity)

Алерты: работа с инцидентами

- Круто классифицировать факапы:
  - плохой релиз
  - человеколажа
  - перегруз
  - хостинг/ДЦ
  .. что-то еще по вкусу

Алерты: работа с инцидентами

- Сначала тушим пожар
- Потом докапываемся до причины
- Классифицируем
- Заводим задачи, чтобы не повторялось тоже самое
- PROFIT через N итераций

Алерты: работа с инцидентами

- Смотрим, какой класс проблем доставляет больше всего проблем
- Если хостинг − переезжаем
- Если человеки − автоматизируем стрёмные места, нагоняем страх на человеков, дальше в зависимости от управленческих навыков:)
- Если релизы − нагоняем страх на QA
- Ну и далее по стеку

Итого

- Мониторинг = обнаружение проблем + поиск причины проблемы
- Мониторинг — оптимизация над ручной диагностикой
- Чем точнее мониторинг, тем быстрее понятна причина проблемы
- Алгоритм внедрения: обеспечиваем нужную точность обнаружения, потом оптимизируем диагностику

Спасибо за внимание!

Вопросы?

Николай Сивко

nsv@okmeter.io