Ростелеком
Информационные Технологии

Reactive для CRUD: фантазии и реальность

спикер

Антон Котов

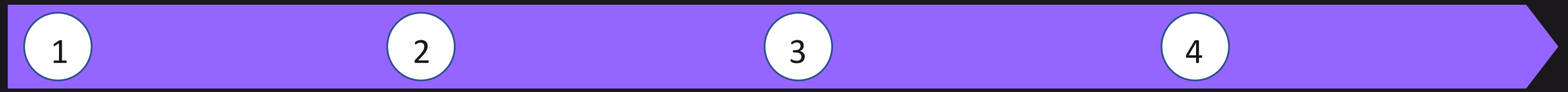
Эксперт – разработчик

Об авторе

- Deutsche Bank
- Яндекс Маркет
- SberDevices
- Ростелеком ИТ

- Выступаю на конференциях
- Разрабатываю свой сайт и мобильное приложение
- Играю на гитаре

R2DBC: история докладов



JPoint 2021

R2DBC. Стоит ли игра свеч?

JPoint 2022

Почему мы решили переходить на R2DBC и чем это закончилось

Joker 2023

Reactive для CRUD: фантазии и реальность

Недостатки предыдущего доклада

- Неудобные графики

  Удобней было бы смотреть, если бы графики по сравнению производительности различных подходов были бы наложены друг на друга, чтобы их легче было сравнивать.

- Дезориентирующие графики

  Графики с RPS несколько дезориентируют. В конце концов понятно, что это RPS для какой-то константной нагрузки. Интересно было бы увидеть как меняется время выполнения запроса при плавном увеличении нагрузки (RPS). Интересно посмотреть на сравнение пропускной способности того или иного подхода.

- Project Loom

  Кажется, в сравнение производительности разных вариантов можно было бы добавить "блокирующий" подход + JDBC поверх Project Loom. Благо он уже где-то в Java 19. Было бы интересно посмотреть на результаты.

- Мало тестов

  Хотелось бы побольше тестов производительности.

- Неубедительные тесты

  Когда ошибки имеются только в начале тестирования, а к концу замера пропадают, это делает тесты в целом неубедительными.

- Недостаточный анализ

  Не хватило анализа в освещении вопроса.

Что будет в докладе

- Разбор различных способов реализации CRUD
- Обзор технологий
- SQL и kotlin код
- Результаты нагрузочного тестирования

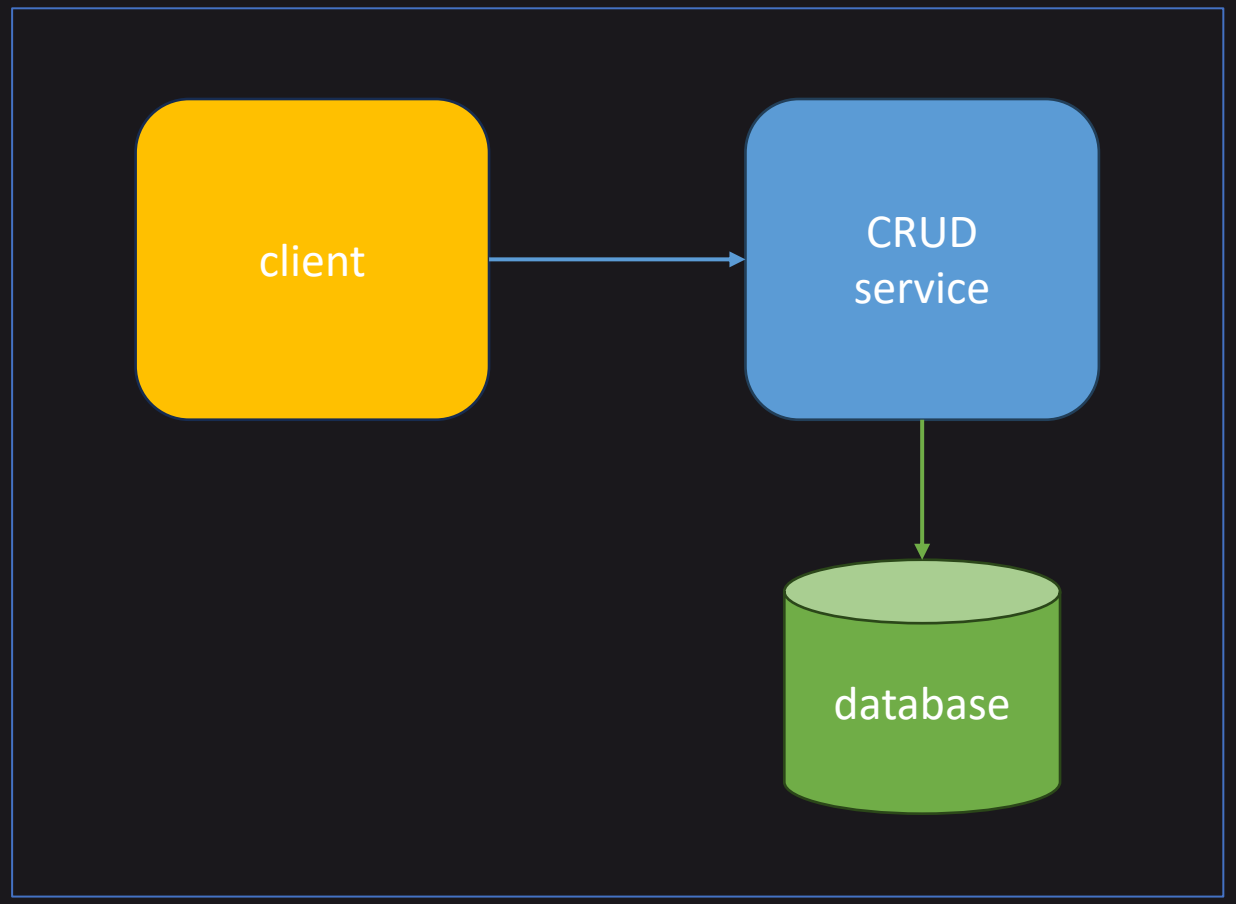
Чего не будет в докладе

- Детальное погружение в технологии
- Демо и лайвкодинг

CRUD

Способы реализации:

1. Blocking + Blocking: webmvc-jdbc
2. Reactive + Blocking: webflux-jdbc
3. Reactive + Reactive: webflux-r2dbc



CRUD: способы реализации

1

webmvc-jdbc

1. webmvc-jdbc
2. webmvc-jdbc-loom

2

webflux-jdbc

1. webflux-jdbc
2. webflux-jdbc-loom
3. webflux-jdbc-coroutine

3

webflux-r2dbc

1. webflux-r2dbc
2. webflux-r2dbc-coroutine

CRUD: способы реализации

Profile				
webmvc-jdbc				
webmvc-jdbc-loom				
webflux-jdbc				
webflux-jdbc-loom				
webflux-jdbc-coroutine				
webflux-r2dbc				
webflux-r2dbc-coroutine				

CRUD: способы реализации

Profile \ Server	tomcat	jetty	undertow	netty
webmvc-jdbc				
webmvc-jdbc-loom				
webflux-jdbc				
webflux-jdbc-loom				
webflux-jdbc-coroutine				
webflux-r2dbc				
webflux-r2dbc-coroutine				

CRUD: способы реализации

Profile \ Server	tomcat	jetty	undertow	netty
webmvc-jdbc				❌
webmvc-jdbc-loom				❌
webflux-jdbc				
webflux-jdbc-loom				
webflux-jdbc-coroutine				
webflux-r2dbc				
webflux-r2dbc-coroutine				

CRUD: способы реализации

Profile \ Server	tomcat	jetty	undertow	netty
webmvc-jdbc				✗
webmvc-jdbc-loom			✗	✗
webflux-jdbc				
webflux-jdbc-loom				
webflux-jdbc-coroutine				
webflux-r2dbc				
webflux-r2dbc-coroutine				

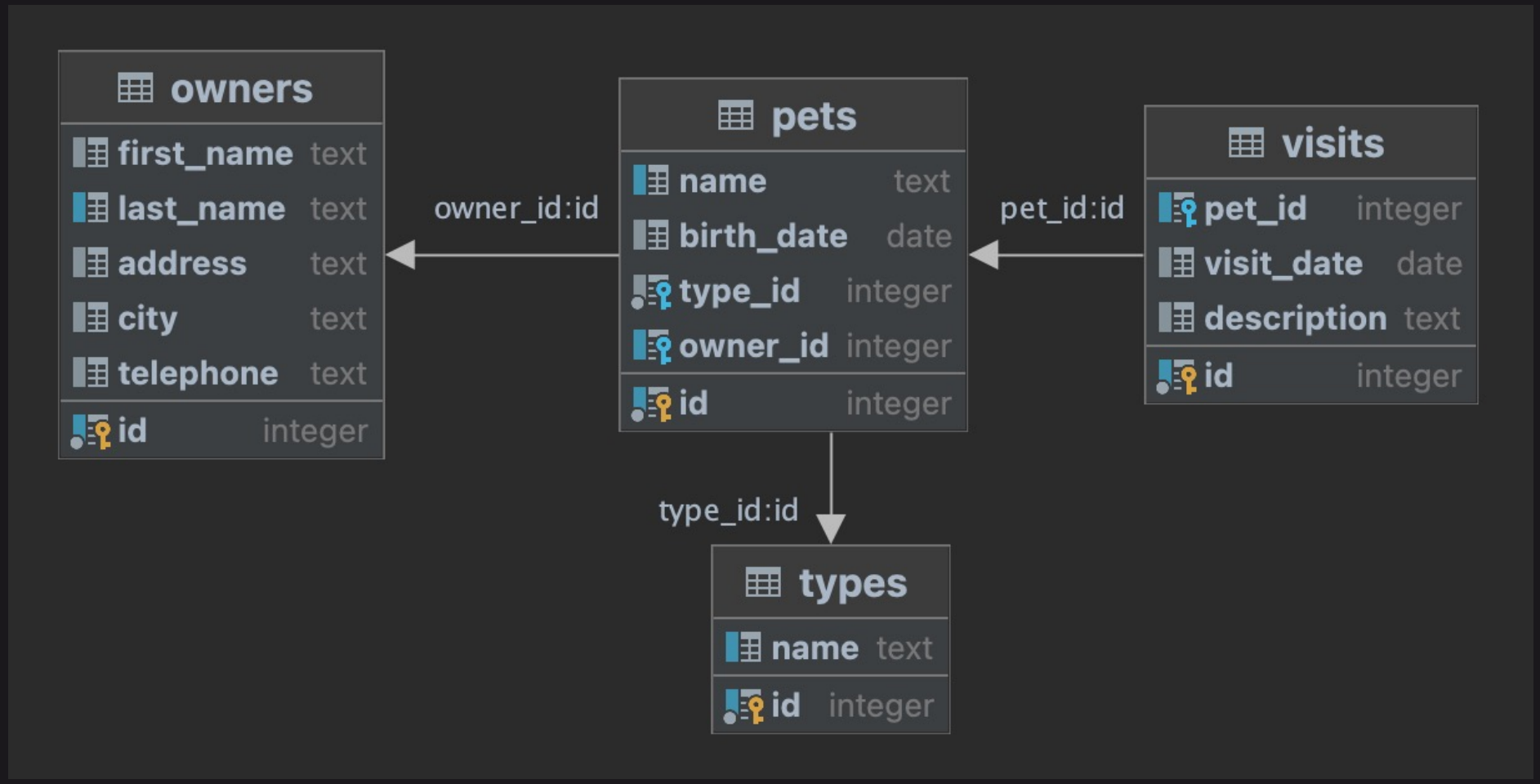
CRUD: способы реализации

Profile \ Server	tomcat	jetty	undertow	netty
webmvc-jdbc	✓	✓	✓	✗
webmvc-jdbc-loom	✓	✓	✗	✗
webflux-jdbc	✓	✓	✓	✓
webflux-jdbc-loom	✓	✓	✓	✓
webflux-jdbc-coroutine	✓	✓	✓	✓
webflux-r2dbc	✓	✓	✓	✓
webflux-r2dbc-coroutine	✓	✓	✓	✓

Тестовый проект

https://github.com/kotoant/spring-petclinic-rest

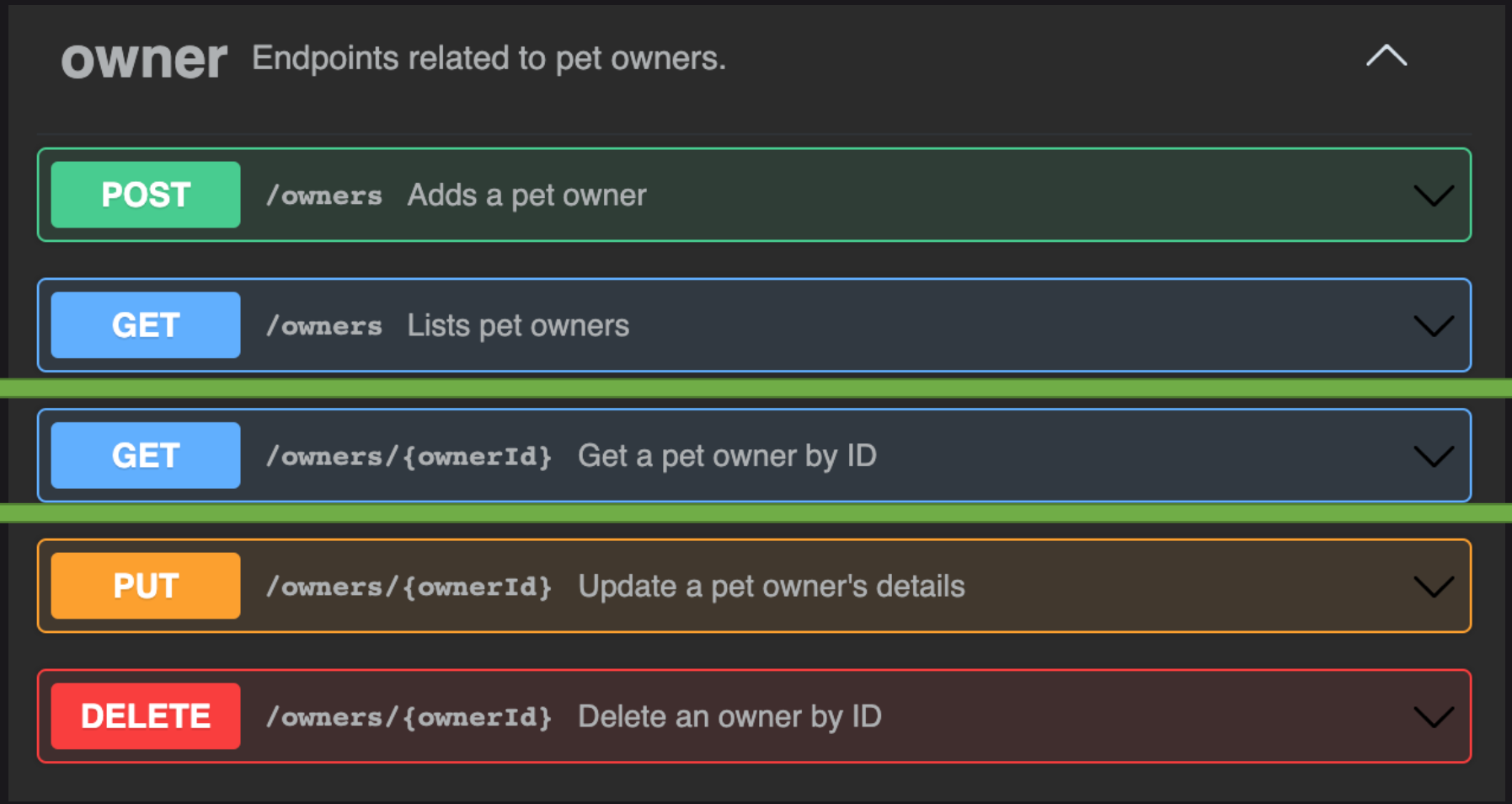
SCAN ME



Тестовый проект: API

https://springpetclinicrest.ru/

SCAN ME



Контроллер


```kotlin
// webmvc: OwnerController.kt
override fun getOwner(ownerId: Int): ResponseEntity<OwnerDto> {
    val owner = clinicService.findOwnerById(ownerId)
    return ResponseEntity(ownerMapper.toOwnerDto(owner), HttpStatus.OK)
}

// webflux-reactive: ReactiveOwnerController.kt
override fun getOwner(ownerId: Int): Mono<ResponseEntity<OwnerDto>> {
    return clinicService.findOwnerById(ownerId).map { owner ->
        ResponseEntity(ownerMapper.toOwnerDto(owner), HttpStatus.OK)
    }
}

// webflux-coroutine: CoroutineOwnerController.kt
override suspend fun getOwner(ownerId: Int): ResponseEntity<OwnerDto> {
    val owner = clinicService.findOwnerById(ownerId)
    return ResponseEntity(ownerMapper.toOwnerDto(owner), HttpStatus.OK)
}
```

Сервис


```kotlin
// JdbcClinicServiceImpl.kt
@Transactional(transactionManager = "transactionManager", readOnly = true)
override fun findOwnerById(id: Int): Owner {
    return ownerRepository.fetchOneById(id) ?: throw OwnerNotFoundException(id)
}

// R2dbcReactiveClinicService.kt
@Transactional(transactionManager = "connectionFactoryTransactionManager", readOnly = true)
override fun findOwnerById(id: Int): Mono<Owner> {
    return ownerRepository.fetchOneById(id).switchIfEmpty {
        Mono.error(OwnerNotFoundException(id))
    }
}

// R2dbcCoroutineClinicService.kt
@Transactional(transactionManager = "connectionFactoryTransactionManager", readOnly = true)
override suspend fun findOwnerById(id: Int): Owner {
    return ownerRepository.fetchOneById(id) ?: throw OwnerNotFoundException(id)
}
```

Сервис: webflux-jdbc


```
// webflux-jdbc-reactive
// JdbcReactiveClinicService.kt
override fun findOwnerById(id: Int): Mono<Owner> = wrapBlockingCall {
    clinicService.findOwnerById(id)
}
private fun <T> wrapBlockingCall(block: Callable<T?>): Mono<T> =
    block.toMono().subscribeOn(scheduler)


// webflux-jdbc-coroutine
// JdbcCoroutineClinicService.kt
override suspend fun findOwnerById(id: Int): Owner = wrapBlockingCall {
    clinicService.findOwnerById(id)
}
private suspend fun <T> wrapBlockingCall(block: suspend CoroutineScope.() -> T): T =
    withContext(coroutineDispatcher, block)
```

Loom


```
@Bean
fun reactiveJdbcServiceScheduler(): Scheduler =
    Schedulers.fromExecutorService(Executors.newVirtualThreadPerTaskExecutor())

@Bean
fun loomTomcatProtocolHandlerVirtualThreadExecutorCustomizer():
        TomcatProtocolHandlerCustomizer<ProtocolHandler> =
    TomcatProtocolHandlerCustomizer { protocolHandler ->
        protocolHandler.executor = Executors.newVirtualThreadPerTaskExecutor()
    }

@Bean
fun loomJettyWebServerFactoryCustomizer():
        WebServerFactoryCustomizer<ConfigurableJettyWebServerFactory> =
    WebServerFactoryCustomizer { factory ->
        factory.setThreadPool(object : ThreadPool {
            private val executor = Executors.newVirtualThreadPerTaskExecutor()
            override fun execute(command: Runnable) = executor.execute(command)
            ...
```

Репозиторий


```kotlin
// JdbcOwnerRepository.kt
override fun deleteById(id: Int): Boolean {
    ctx.deleteVisitsByOwnerId(id).execute()
    ctx.deletePetsByOwnerId(id).execute()
    return ctx.deleteOwnerById(id).execute() > 0
}

// R2dbcOwnerRepository.kt
fun deleteById(id: Int): Mono<Boolean> = client.inContext { ctx ->
    ctx.deleteVisitsByOwnerId(id)
        .then(ctx.deletePetsByOwnerId(id))
        .then(ctx.deleteOwnerById(id)).map { it > 0 }
}

// CoroutineOwnerRepository.kt
suspend fun deleteById(id: Int): Boolean = client.inContextCoroutine { ctx ->
    ctx.deleteVisitsByOwnerId(id).awaitSingle()
    ctx.deletePetsByOwnerId(id).awaitSingle()
    return@inContextCoroutine ctx.deleteOwnerById(id).awaitSingle() > 0
}
```

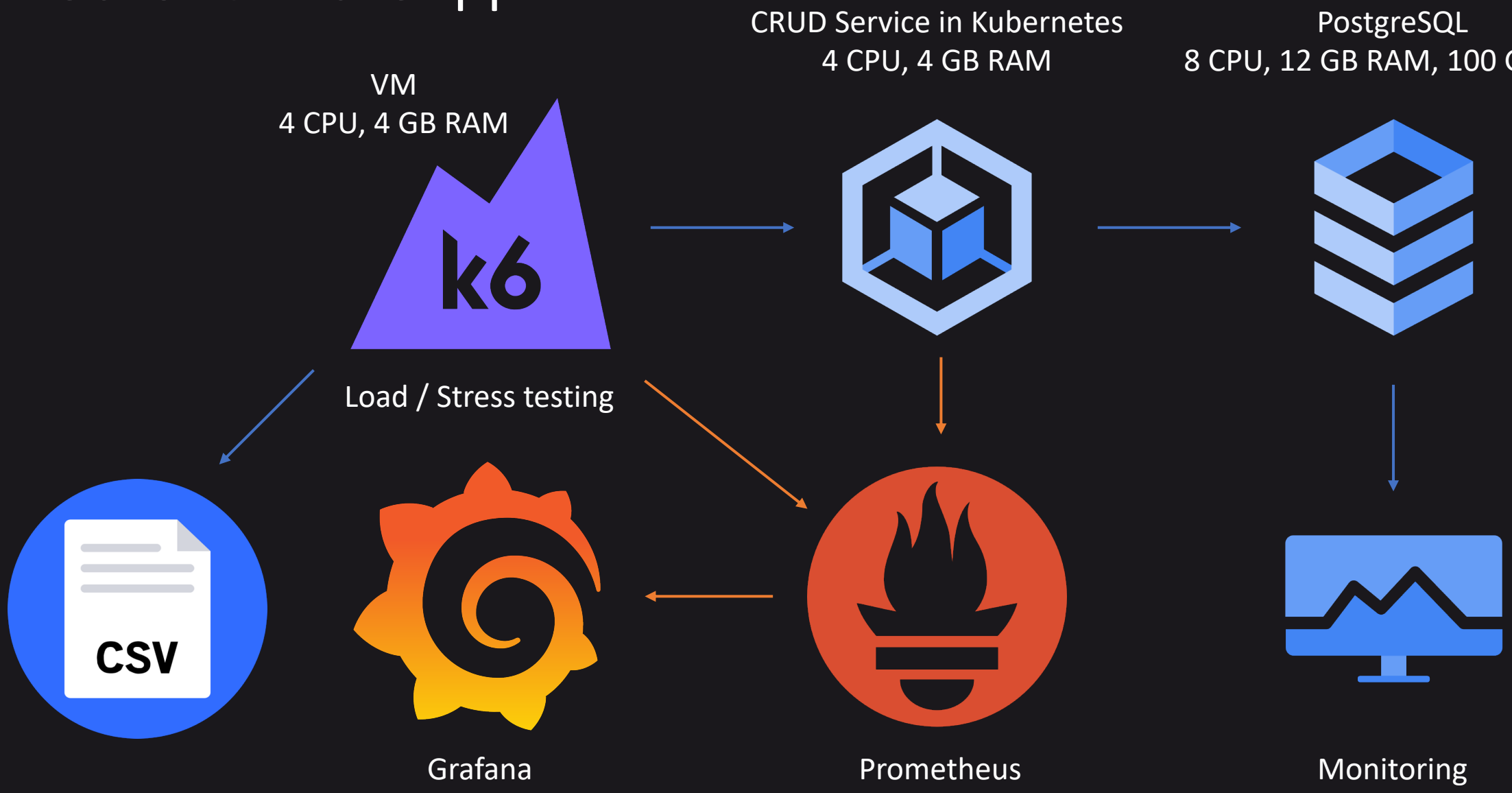
Репозиторий: Jooq


```
// ClinicRepository.kt
fun DSLContext.fetchOneOwnerById(id: Int): Owner? =
    selectOneOwnerById(id).fetchOne(ownerMapper())


fun DSLContext.fetchOneOwnerByIdReactive(id: Int): Mono<Owner> =
    selectOneOwnerById(id).fetchOneReactive(ownerMapper())

fun <R : Record, E : Any> Publisher<R>.fetchOneReactive(mapper: RecordMapper<in
R, E>) = Mono.from(this).map(mapper)

private fun ownerMapper() = Records.mapping(::Owner)

private fun DSLContext.selectOneOwnerById(id: Int) =
    selectOwners().where(OWNERS.ID.equal(id))

private fun DSLContext.selectOwners() =
    select(OWNERS.ID, OWNERS.FIRST_NAME, OWNERS.LAST_NAME, ...
```

Промежуточные выводы по коду

1. Нереактивный код – самый простой
2. Реактивный код – достаточно сложный
3. Неблокирующую webflux-логику можно писать нереактивным кодом, используя Kotlin Coroutines

Тестовый стенд



Тестовый сценарий

1. Create owner
2. Create pet
3. Create visit
4. Read owner
5. Read pet
6. Read visit
7. Find owners by last name
8. Update pet
9. Update visit
10. Update owner

Тестируем правильно

- Идемпотентность

  Сложность постоянно повторяющейся тестируемой операции не должна зависеть от времени

- Равные внешние условия для всех тестов

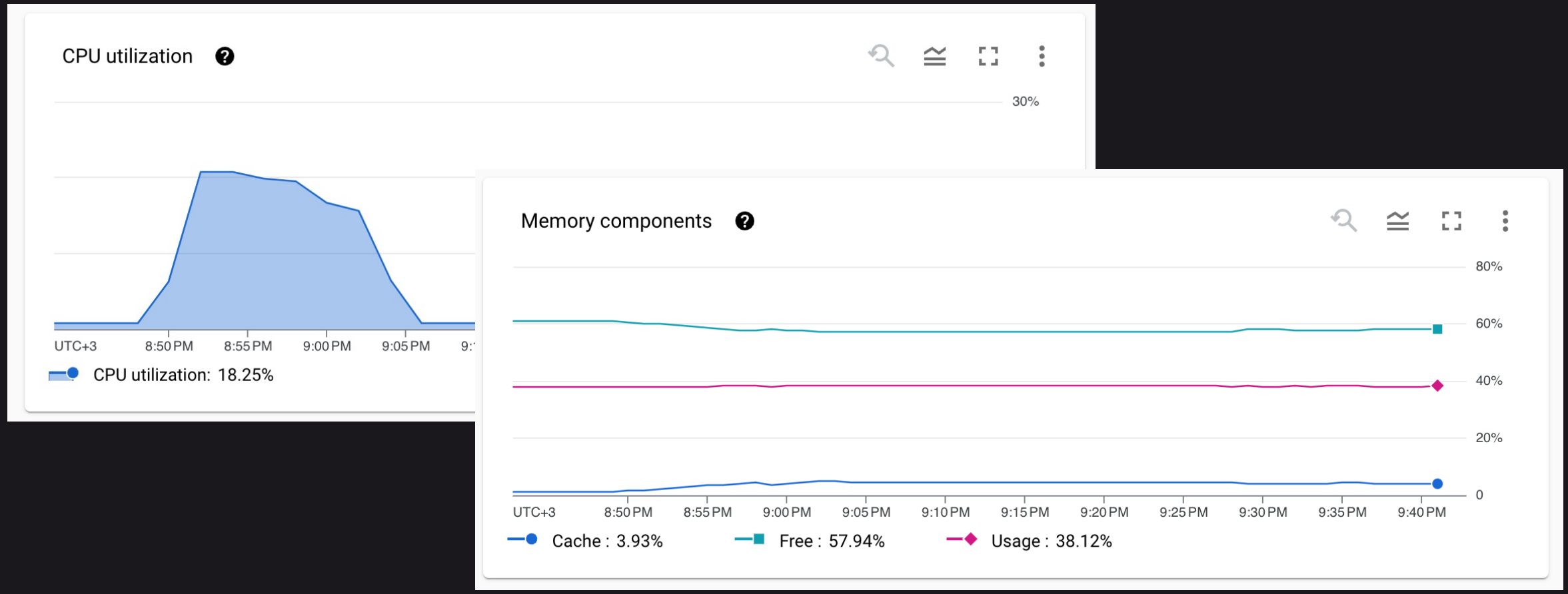
  Кластер кубера, база данных, нагрузочный сценарий

- Тестируем именно наш сервис

  Важно не перегрузить базу данных

Подбор параметров для тестирования

База данных: 8 CPU, 12 GB RAM, 100 GB SSD - при пуле соединений размеров 8 расходуется порядка 20% CPU и 40% RAM



Типы генераторов нагрузки

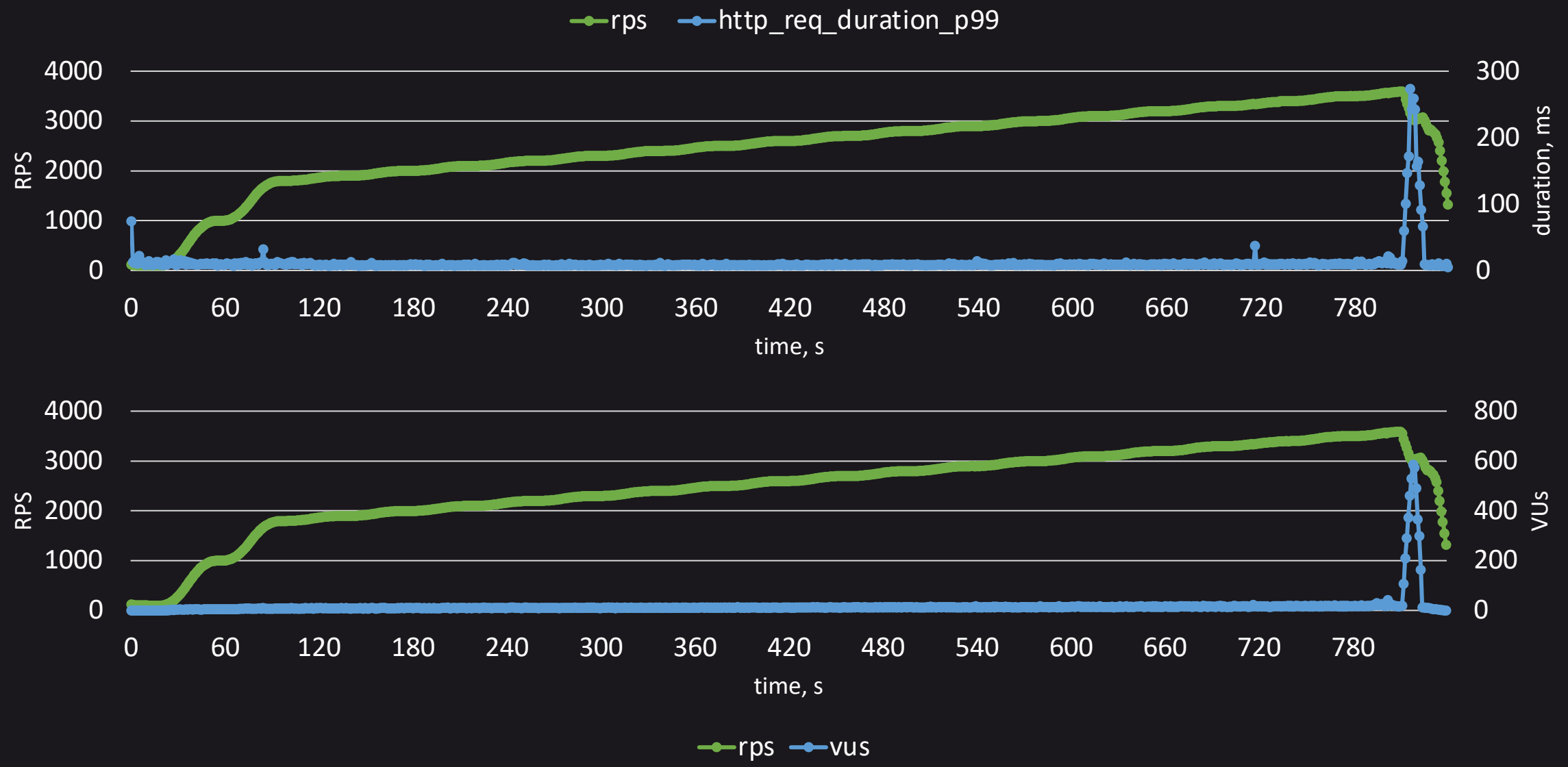
- Ramping arrival rate

  Позволяет понять, какой максимальный RPS система может обслужить до того, как закончатся ресурсы (CPU)

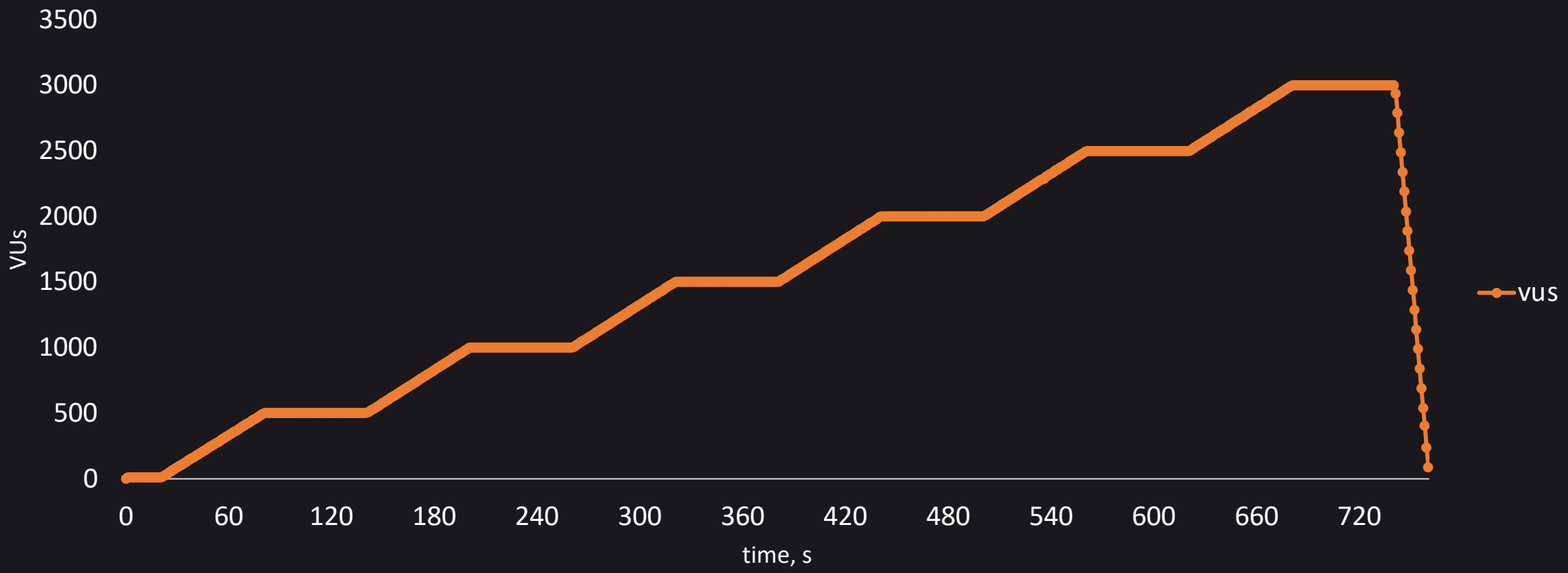
- Ramping VUs

  Позволяет понять, какое максимальное количество VUs система одновременно может обслуживать при заданном таймауте на один запрос даже после того, как закончились ресурсы (CPU)

Ramping arrival rate

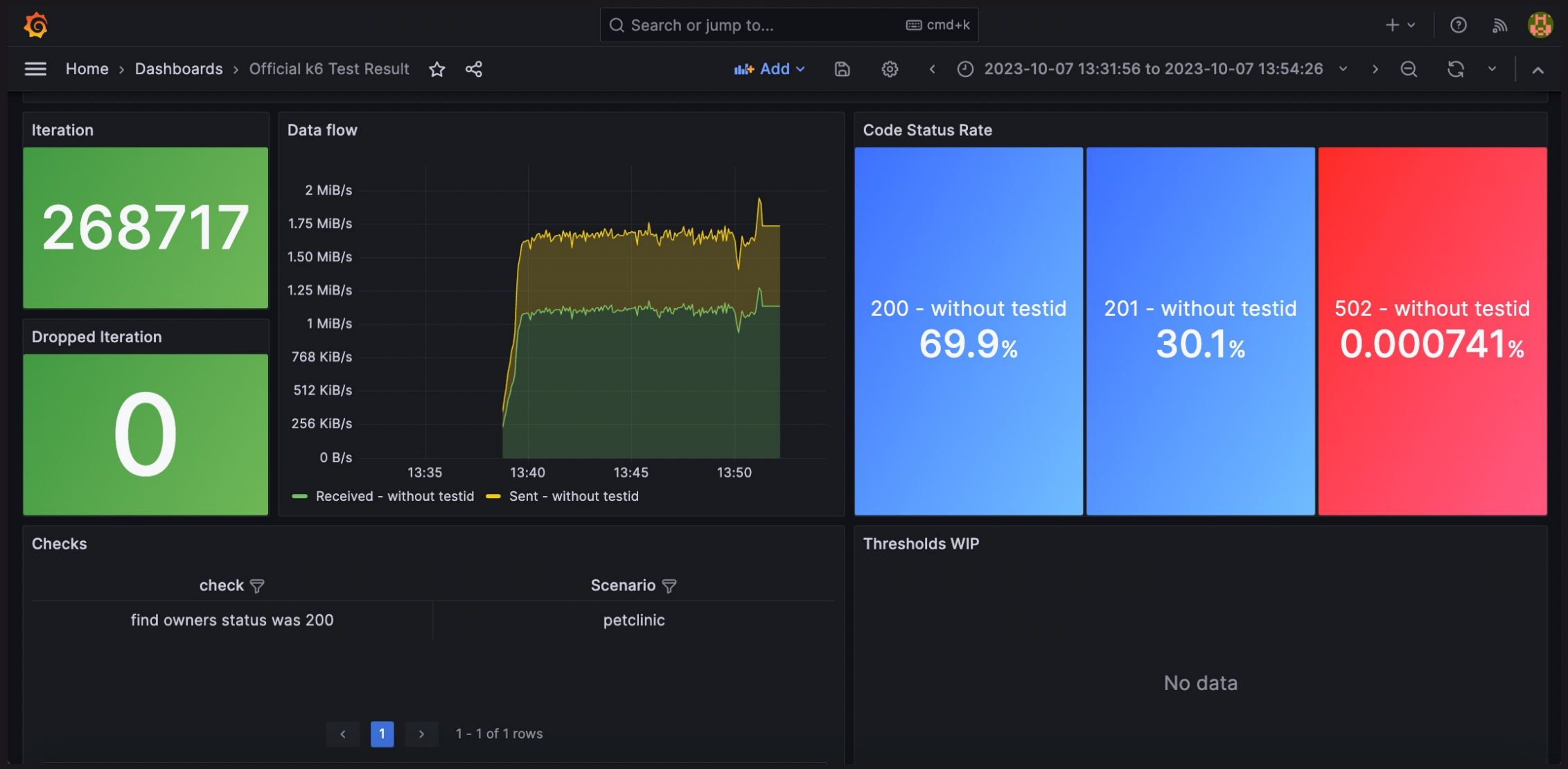


Ramping VUs

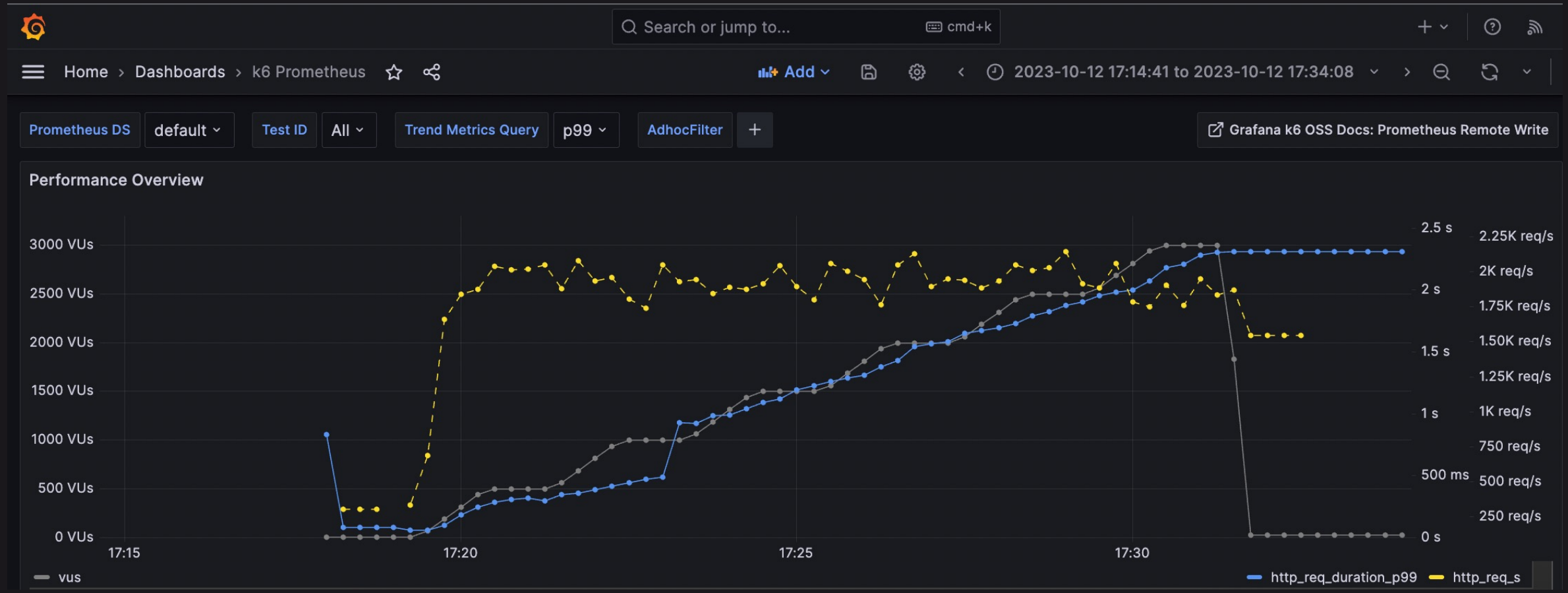

3500
3000
2500
2000
1500
1000
500
0
VUs
0
60
120
180
240
300
360
420
480
540
600
660
720
time, s
vus

K6 Grafana dashboards

K6 Grafana dashboards



K6 Grafana dashboards



K6 Grafana dashboards

webmvc-jdbc-tomcat


```
checks.........................: 99.99%  ✓ 2910826     ✗ 11
http_req_duration..............: avg=411.92ms min=0s      med=398.07ms max=2.1s
p(90)=741.6ms  p(95)=767.43ms
  { expected_response:true }...: avg=411.92ms min=2.47ms med=398.07ms max=2.1s
p(90)=741.6ms  p(95)=767.43ms
http_req_failed................: 0.00%   ✓ 11          ✗ 2910829
http_reqs......................: 2910840 3830.028091/s
iteration_duration.............: avg=4.11s    min=37.6ms med=3.94s    max=8.75s
p(90)=7.42s    p(95)=7.75s
iterations.....................: 289739  381.233084/s
vus............................: 88      min=0         max=3000
vus_max........................: 3000    min=1914      max=3000
```

CRUD: способы реализации

Profile \ Server	tomcat	jetty	undertow	netty
webmvc-jdbc	p(95)=767.43ms 3830.028091/s	p(95)=866.27ms 3553.384859/s	p(95)=991.5ms 3051.901671/s	✗
webmvc-jdbc-loom	p(95)=1.92s 3163.810703/s	p(95)=2.05s 3037.724714/s	✗	✗
webflux-jdbc-reactive	p(95)=1.39s 2799.482267/s	p(95)=1.61s 2618.064934/s	p(95)=1.32s 2890.171394/s	p(95)=1.17s 3295.238807/s
webflux-jdbc-loom	p(95)=1.9s 2491.669639/s	p(95)=2.06s 2355.754691/s	p(95)=1.63s 2692.307119/s	p(95)=1.56s 2790.748679/s
webflux-jdbc-coroutine	p(95)=1.47s 2601.091587/s	p(95)=1.69s 2451.942169/s	p(95)=1.28s 2953.696585/s	p(95)=1.26s 2996.382065/s
webflux-r2dbc-reactive	p(95)=2.38s 1740.996606/s	p(95)=2.49s 1706.607781/s	p(95)=2.17s 1730.307006/s	p(95)=1.96s 1967.406926/s
webflux-r2dbc-coroutine	p(95)=2.49s 1587.399999/s	p(95)=2.63s 1537.078602/s	p(95)=2.31s 1625.422698/s	p(95)=2.23s 1745.692372/s

Анализируем реал-тайм метрики

Даже для теста в 12 минут CSV файл достаточно большой:

- 2.8 GB
- 24 млн строк

Excel такое точно не потянет, что делать?

Даже если чем-то откроем файл, то что дальше? Как анализировать данные?

Postgres выручает!

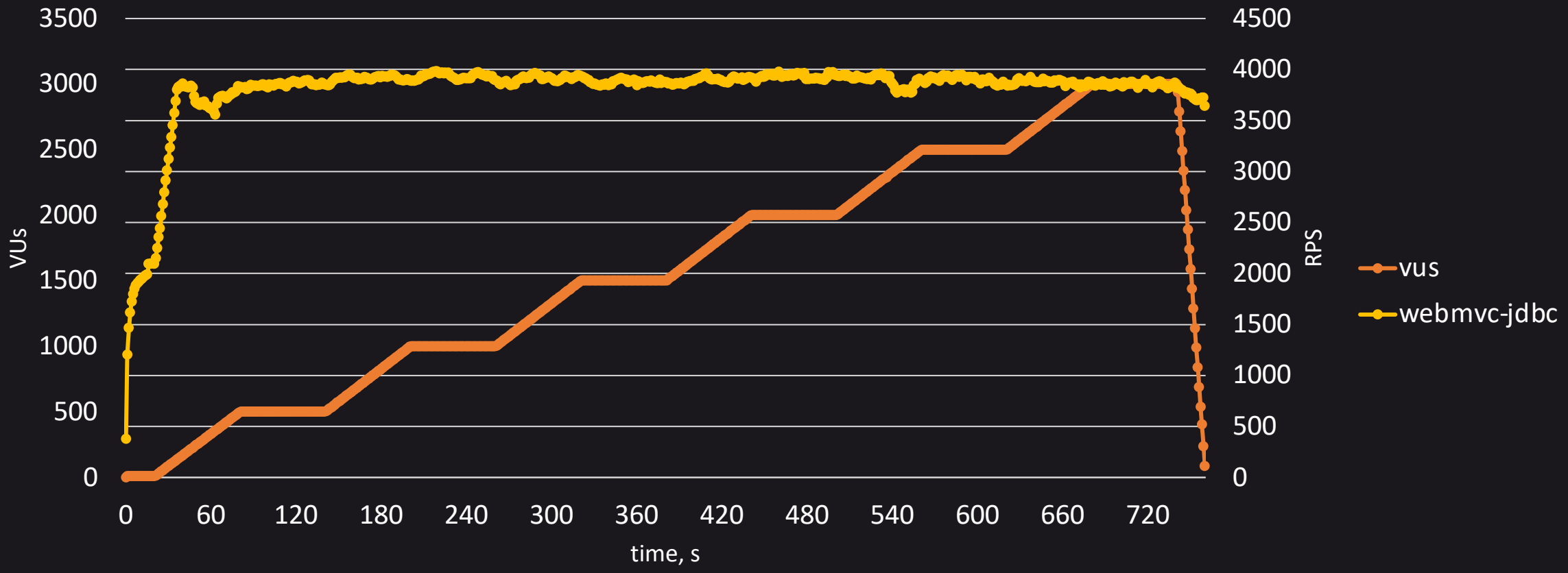

```
create table ”webmvc-jdbc-tomcat”
(
    metric_name         text,
    timestamp           bigint,
    metric_value        numeric,
    ...
);
```

```
postgres=# COPY ”webmvc-jdbc-tomcat”
FROM ‘/Users/anton/webmvc-jdbc-tomcat.csv'
WITH (FORMAT csv, header);
```

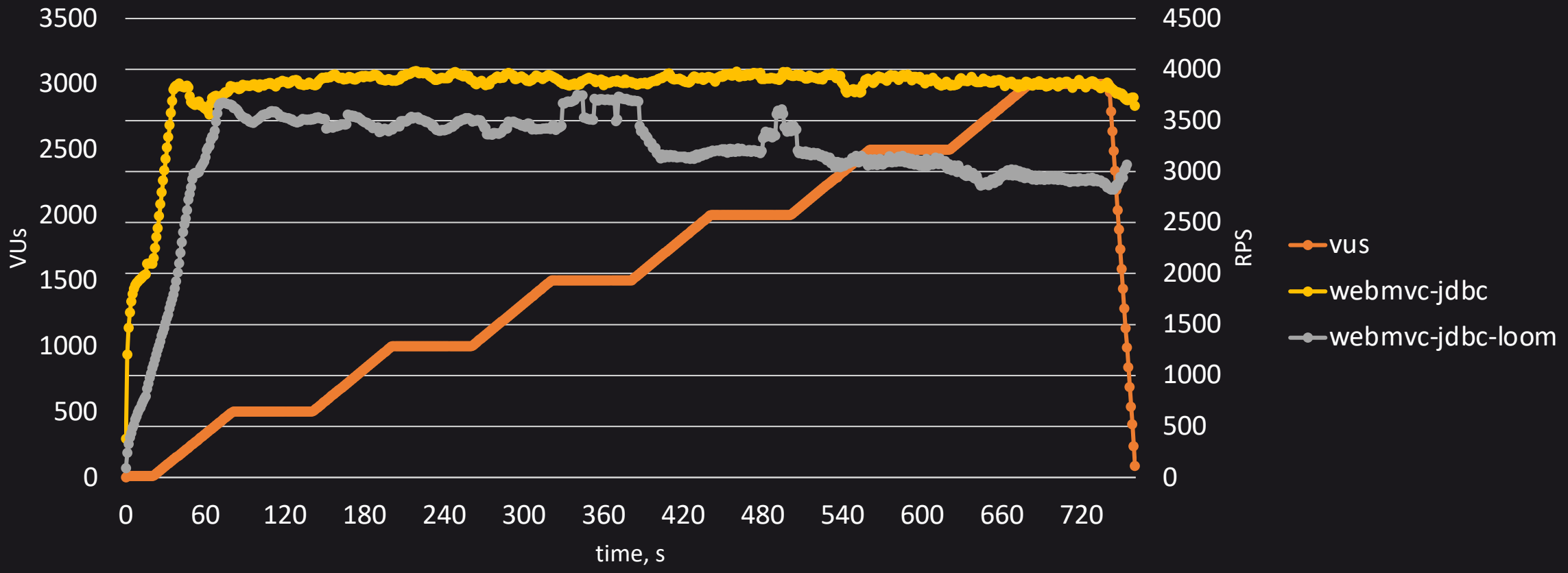
Превращаем 24 млн строк в 800


```
select t.*,
  avg(rps_raw) over (order by t.time rows between 15 preceding and current row) rps
from (select (t1.timestamp - (select min(timestamp) from "webmvc-jdbc-tomcat")) time,
 max(t1.metric_value) vus,
 count(t2.metric_value) rps_raw,
 min(t2.metric_value) http_req_duration_min,
 max(t2.metric_value) http_req_duration_max,
 percentile_disc(0.50) within group (order by t2.metric_value) http_req_duration_p50,
 percentile_disc(0.95) within group (order by t2.metric_value) http_req_duration_p95,
 percentile_disc(0.99) within group (order by t2.metric_value) http_req_duration_p99
from "webmvc-jdbc-tomcat" t1, "webmvc-jdbc-tomcat" t2
where t2.metric_name = 'http_req_duration' and t1.metric_name = 'vus'
  and t1.timestamp = t2.timestamp
group by t1.timestamp
order by time) t;
```

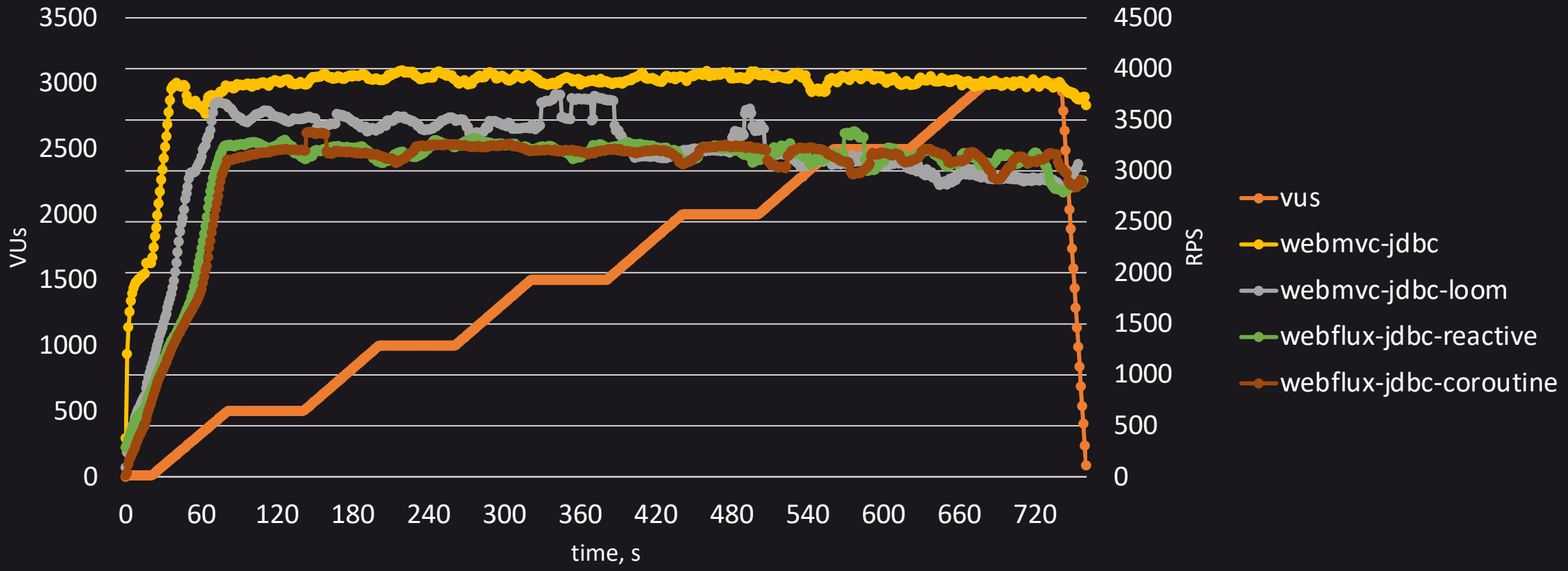
RPS


3500
3000
2500
2000
1500
1000
500
0
VUs
4500
4000
3500
3000
2500
2000
1500
1000
500
0
RPS
0
60
120
180
240
300
360
420
480
540
600
660
720
time, s
vus
webmvc-jdbc

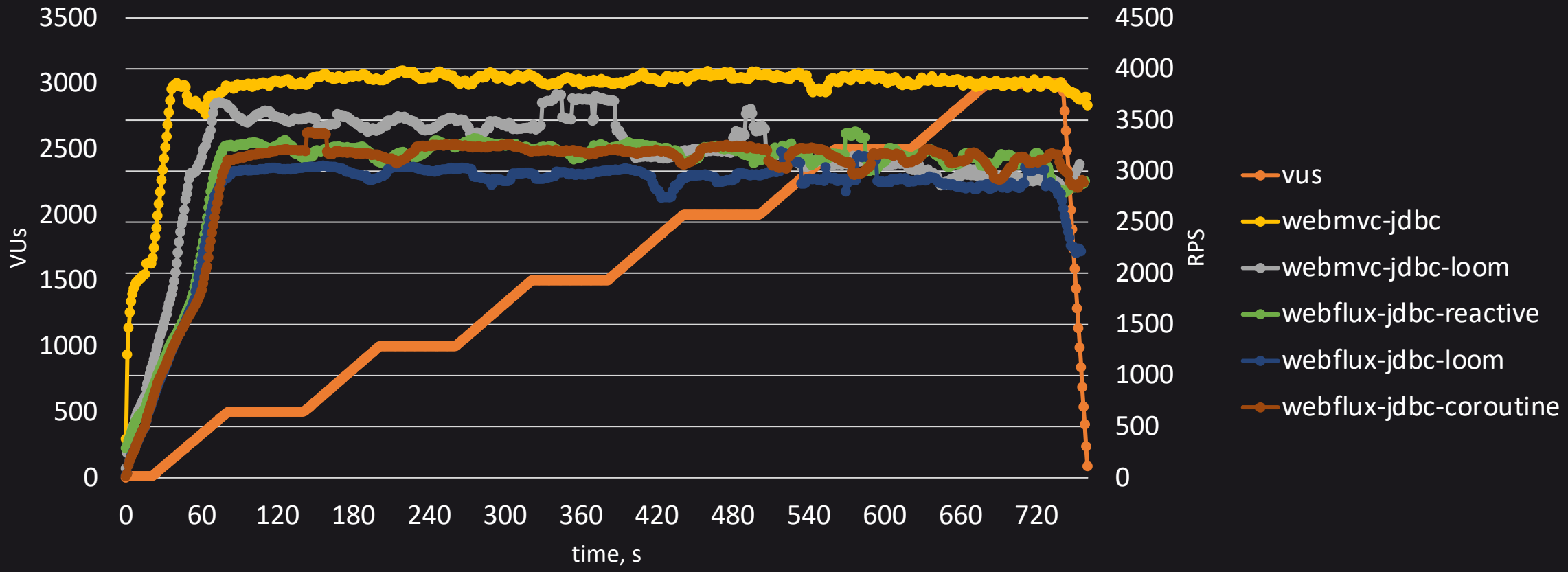
RPS


3500
3000
2500
2000
1500
1000
500
0
VUs
4500
4000
3500
3000
2500
2000
1500
1000
500
0
RPS
0
60
120
180
240
300
360
420
480
540
600
660
720
time, s
vus
webmvc-jdbc
webmvc-jdbc-loom

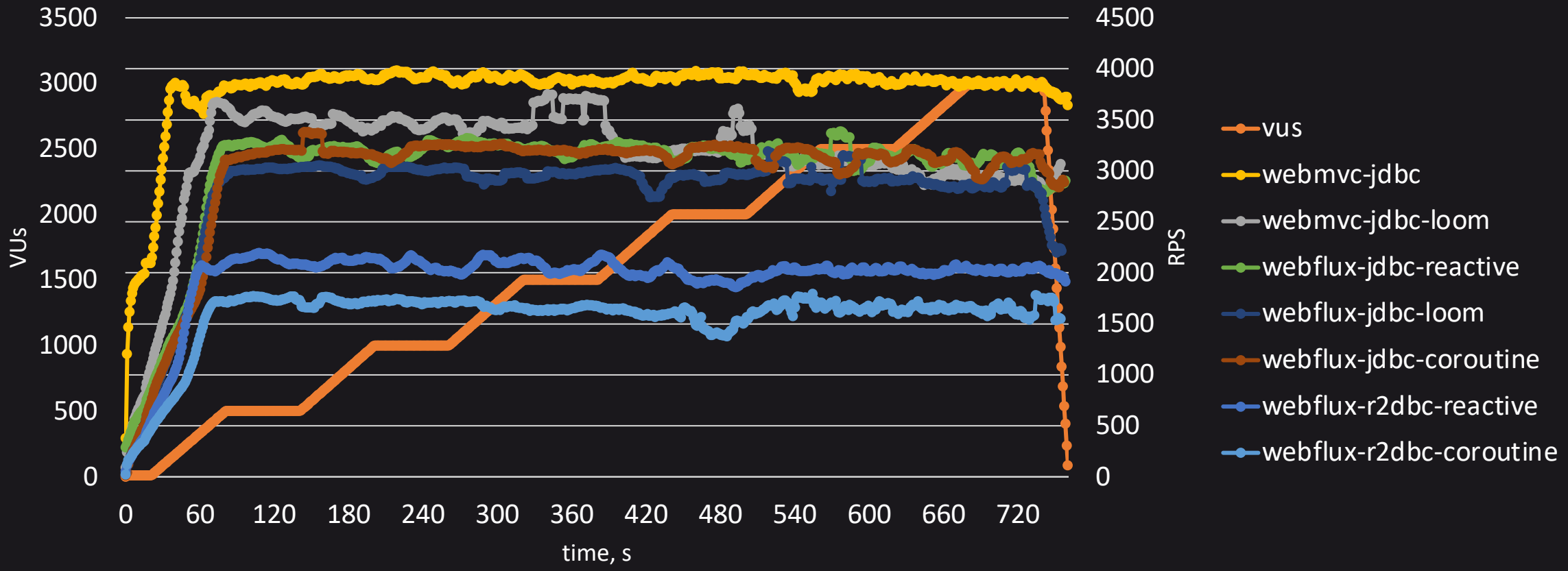
RPS


3500
3000
2500
2000
1500
1000
500
0
VUs
4500
4000
3500
3000
2500
2000
1500
1000
500
0
RPS
0
60
120
180
240
300
360
420
480
540
600
660
720
time, s
vus
webmvc-jdbc
webmvc-jdbc-loom
webflux-jdbc-reactive
webflux-jdbc-coroutine

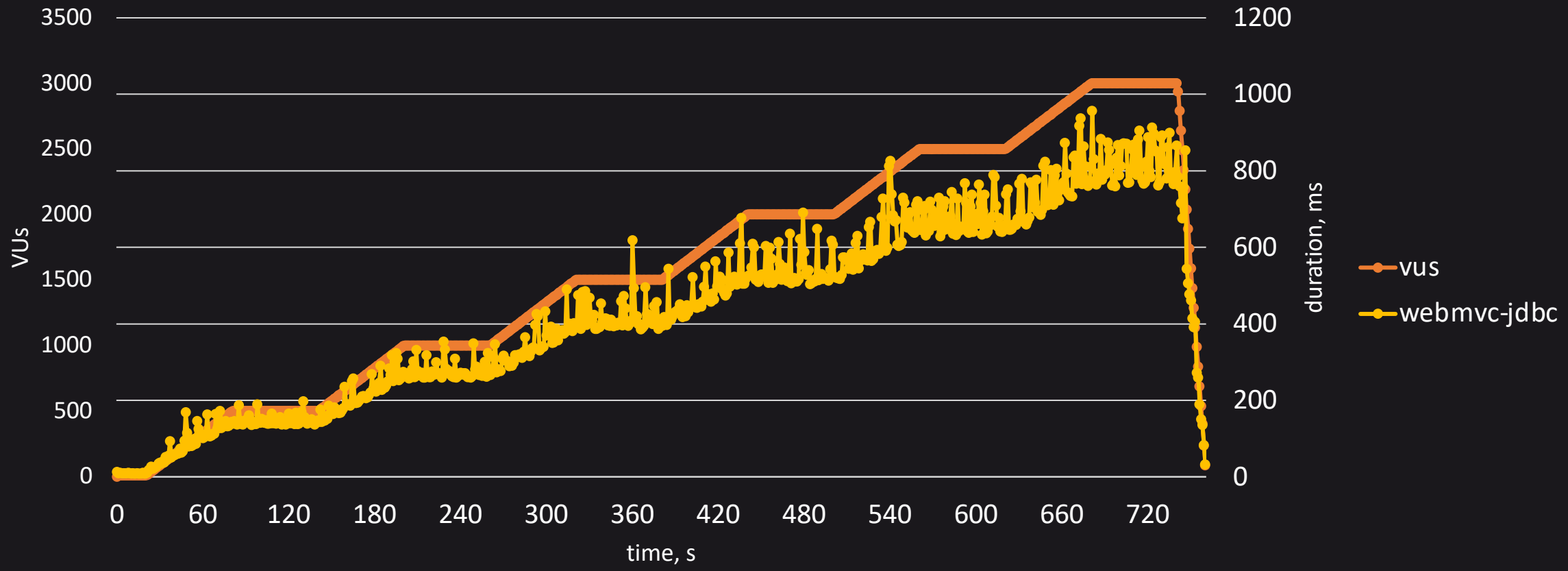
RPS


3500
3000
2500
2000
1500
1000
500
0
VUs
4500
4000
3500
3000
2500
2000
1500
1000
500
0
RPS
0
60
120
180
240
300
360
420
480
540
600
660
720
time, s
vus
webmvc-jdbc
webmvc-jdbc-loom
webflux-jdbc-reactive
webflux-jdbc-loom
webflux-jdbc-coroutine

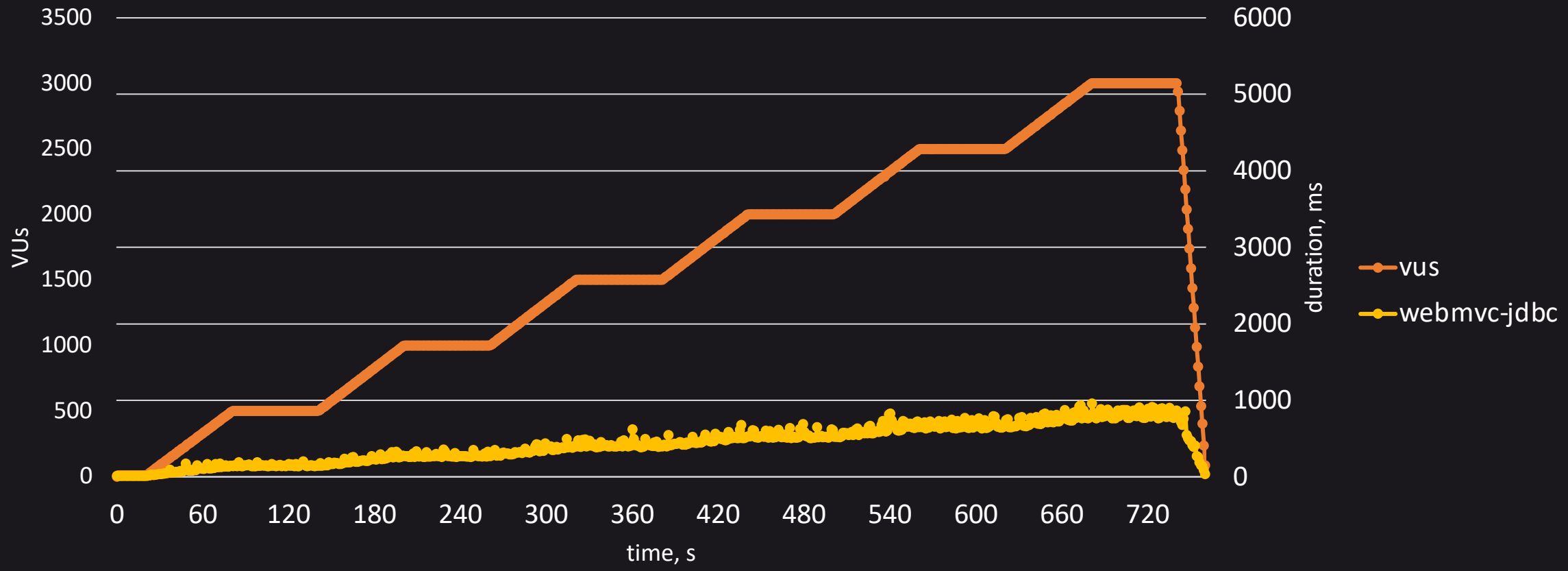
RPS


3500
3000
2500
2000
1500
1000
500
0
VUs
4500
4000
3500
3000
2500
2000
1500
1000
500
0
RPS
0
60
120
180
240
300
360
420
480
540
600
660
720
time, s
vus
webmvc-jdbc
webmvc-jdbc-loom
webflux-jdbc-reactive
webflux-jdbc-loom
webflux-jdbc-coroutine
webflux-r2dbc-reactive
webflux-r2dbc-coroutine

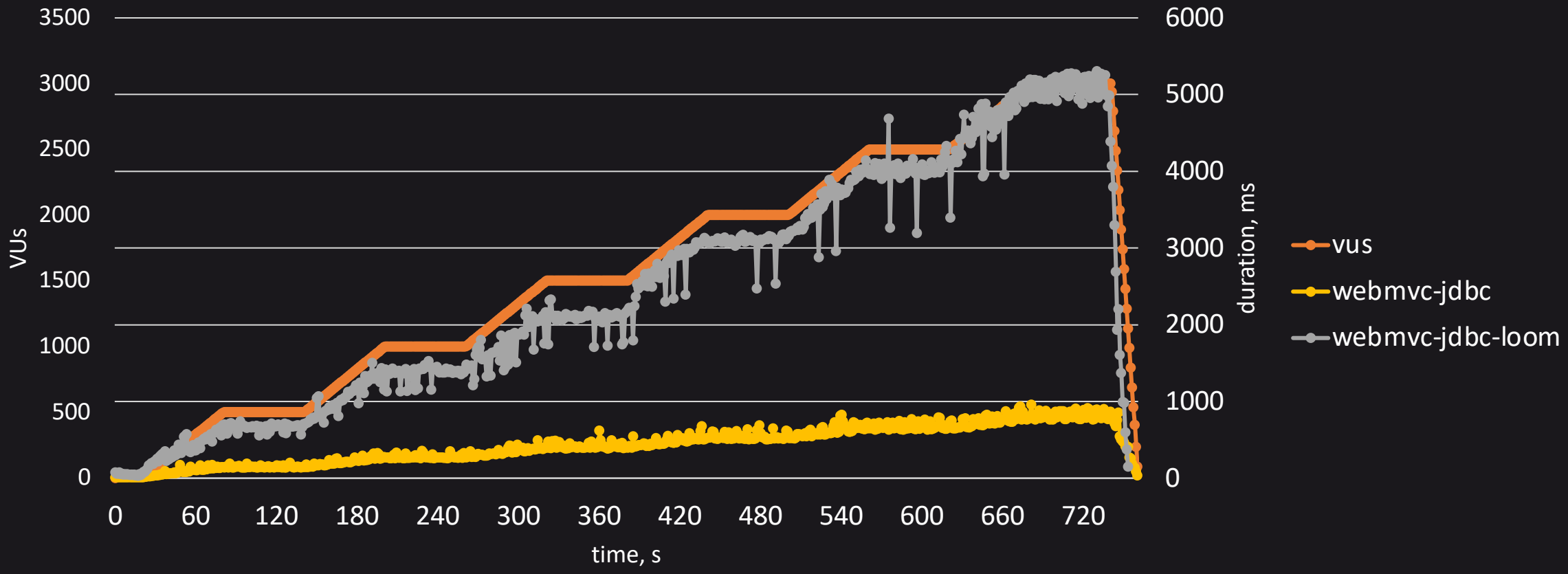
request duration: p(99)



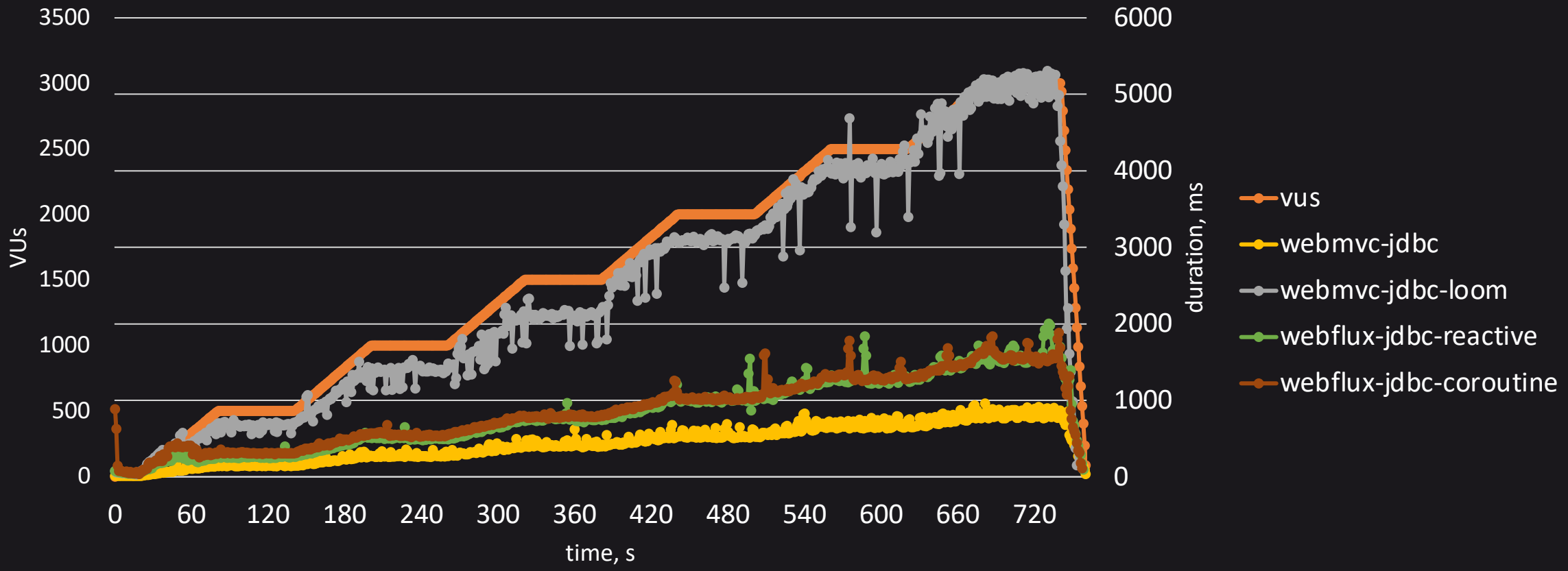
request duration: p(99)



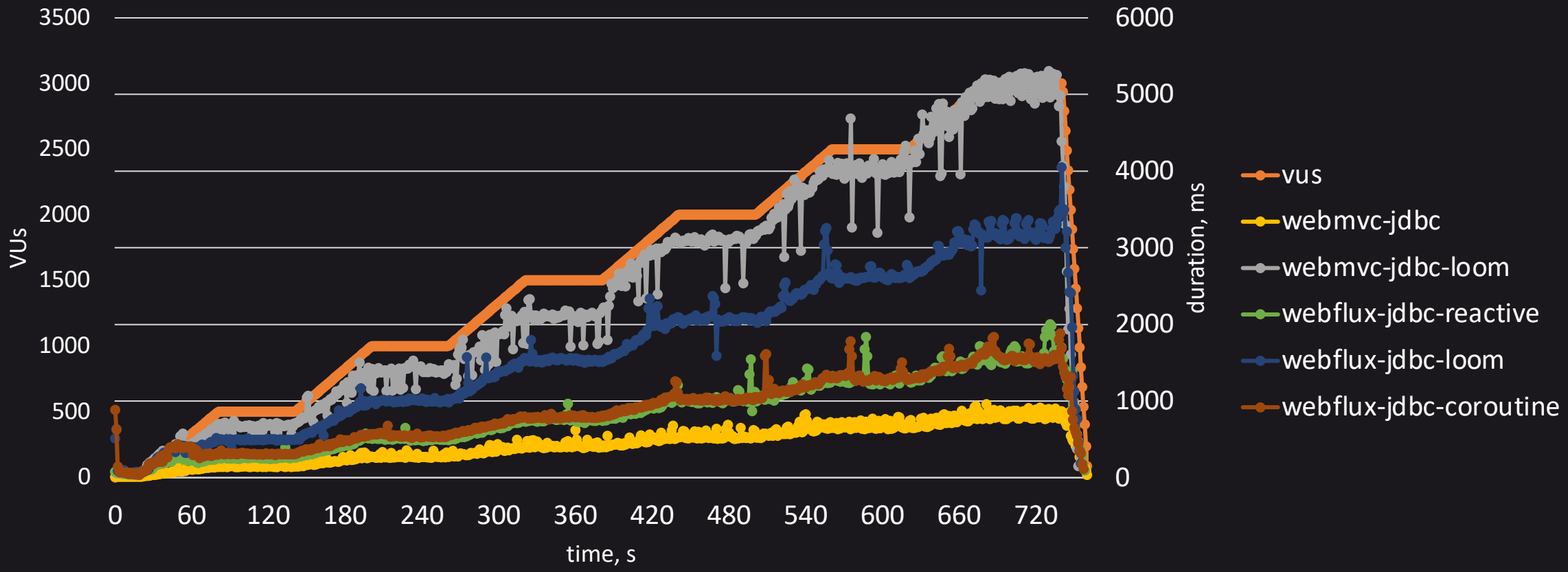
request duration: p(99)



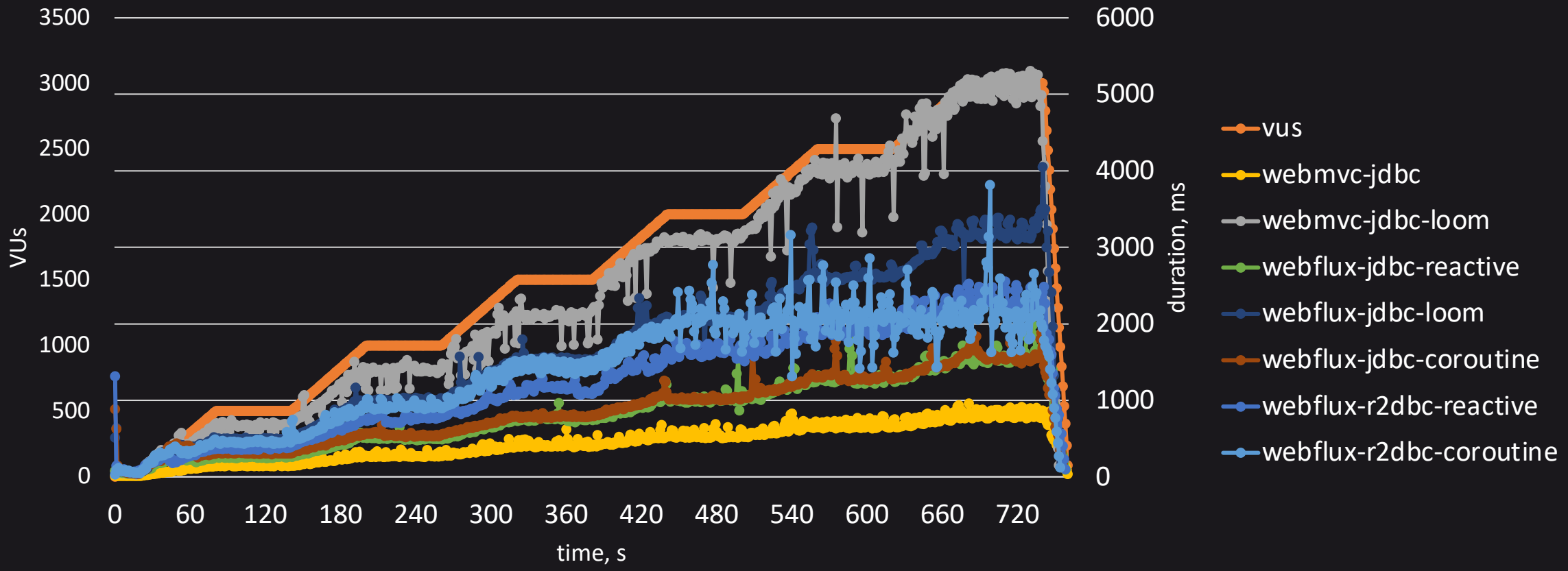
request duration: p(99)



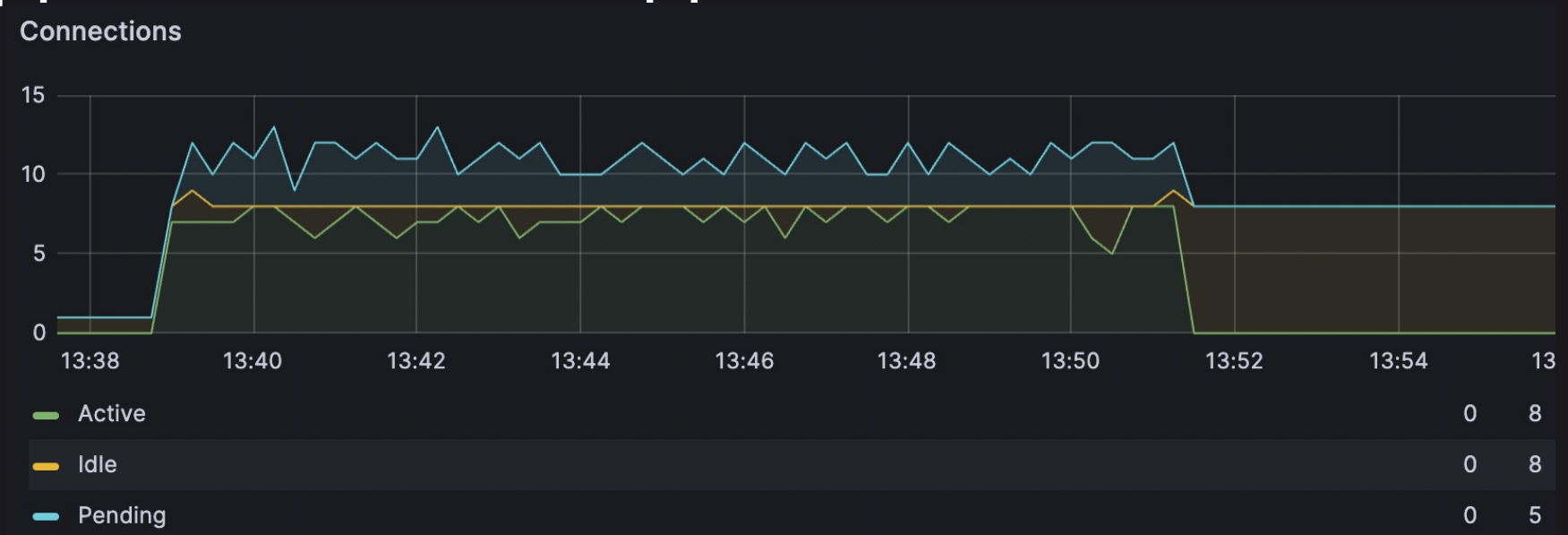
request duration: p(99)



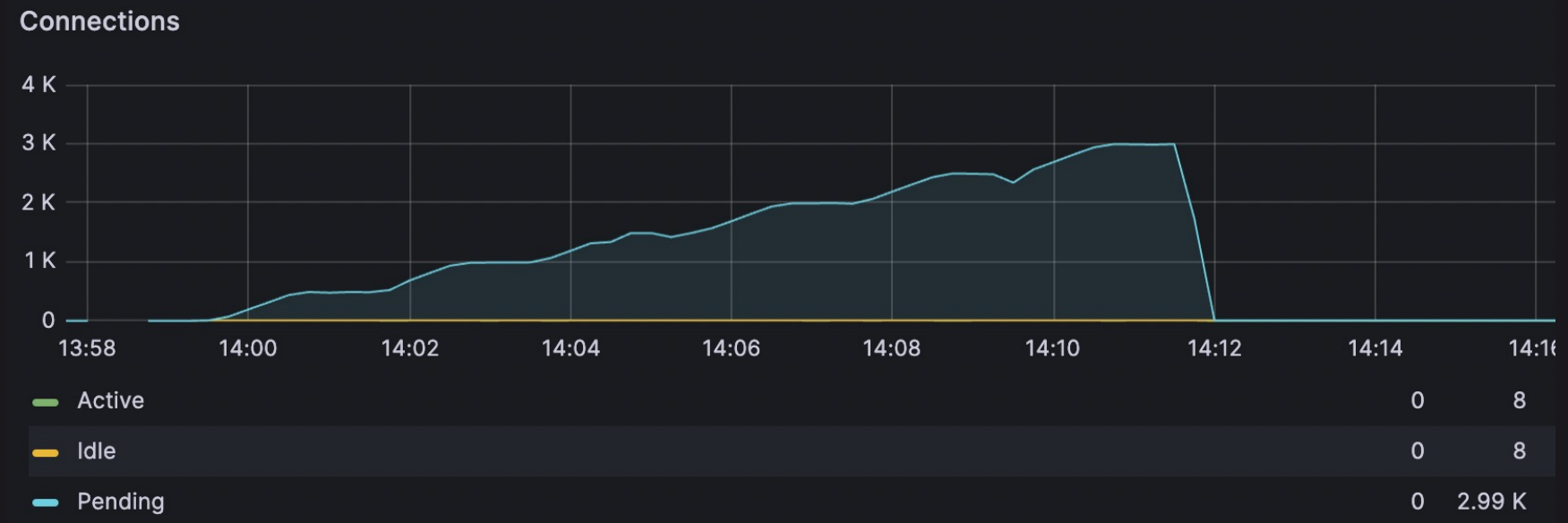
request duration: p(99)



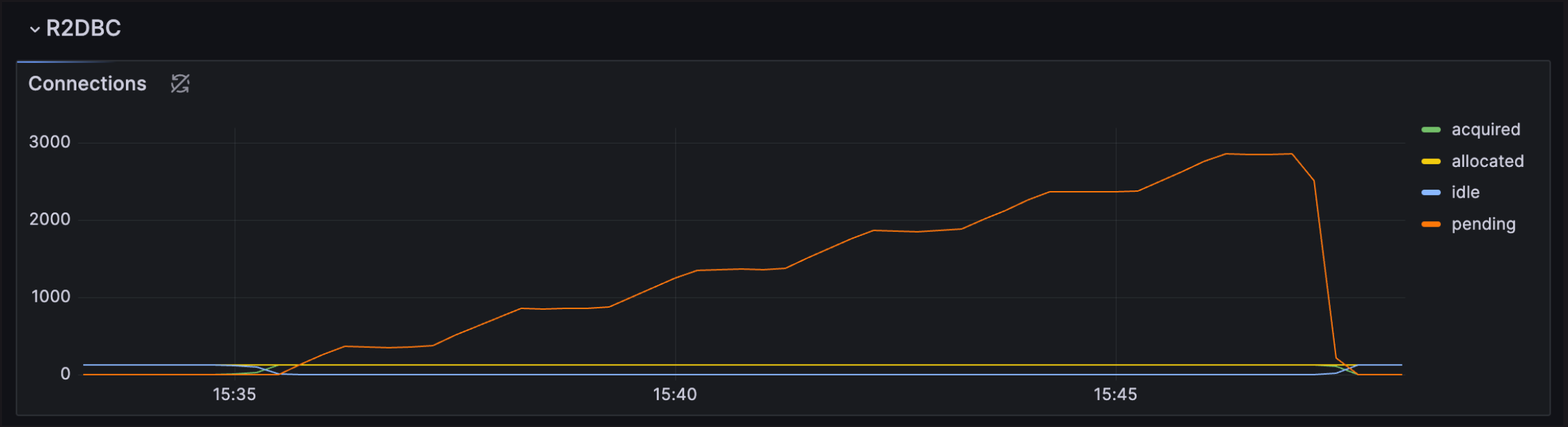
Пул соединений к базе данных: JDBC



Loom



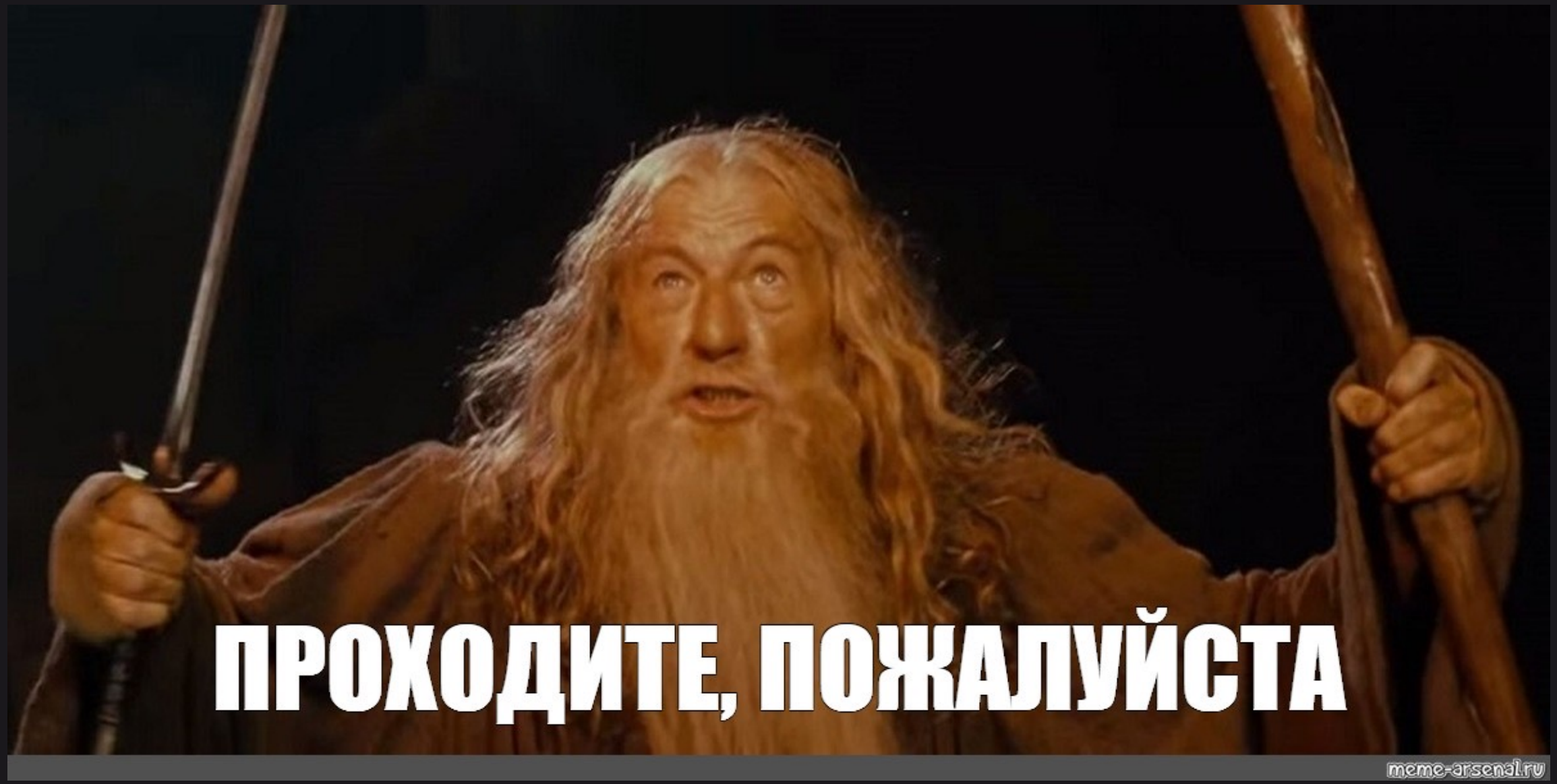
Пул соединений к базе данных: R2DBC



Проблема health check’a

Проблема health check’a: webmvc

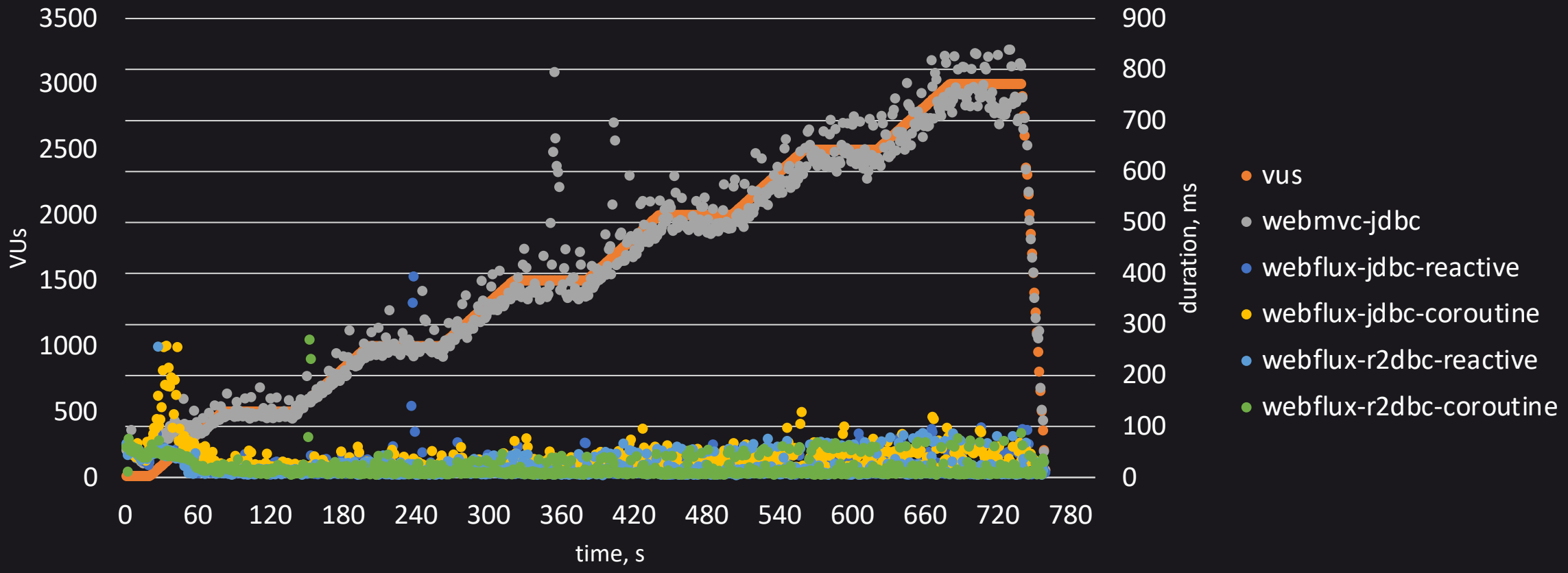
Проблема health check’a: webflux



Тестовый сценарий: health check

1. Create owner
2. Create pet
3. Create visit
4. Read owner
5. Read pet
6. Read visit
7. Find owners by last name
8. Update pet
9. Update visit
10. Update owner
11. Health check

Health check: request duration: p(99)



Где потенциально r2dbc может выигрывать?


```kotlin
// R2dbcOwnerRepository.kt
fun deleteById(id: Int): Mono<Boolean> = client.inContext { ctx ->
    ctx.deleteVisitsByOwnerId(id)
        .then(ctx.deletePetsByOwnerId(id))
        .then(ctx.deleteOwnerById(id)).map { it > 0 }
}

fun deleteById(id: Int): Mono<Boolean> = client.inContext { ctx ->
    Mono.zip(
        ctx.deleteVisitsByOwnerId(id).toMono(),
        ctx.deletePetsByOwnerId(id).toMono(),
        ctx.deleteOwnerById(id).toMono()
    ).map { it.t3 > 0 }
}
```

Есть ли аналог Mono.zip в корутинах?


```kt
// CoroutineOwnerRepository.kt
suspend fun deleteById(id: Int): Boolean = client.inContextCoroutine { ctx ->
    ctx.deleteVisitsByOwnerId(id).awaitSingle()
    ctx.deletePetsByOwnerId(id).awaitSingle()
    return@inContextCoroutine ctx.deleteOwnerById(id).awaitSingle() > 0
}

suspend fun deleteById(id: Int): Boolean = client.inContextCoroutine {
    coroutineScope {
        val res1 = async { it.deleteVisitsByOwnerId(id).awaitSingle() }
        val res2 = async { it.deletePetsByOwnerId(id).awaitSingle() }
        val res3 = async { it.deleteOwnerById(id).awaitSingle() }
        res1.await()
        res2.await()
        res3.await() > 0
    }
}
```

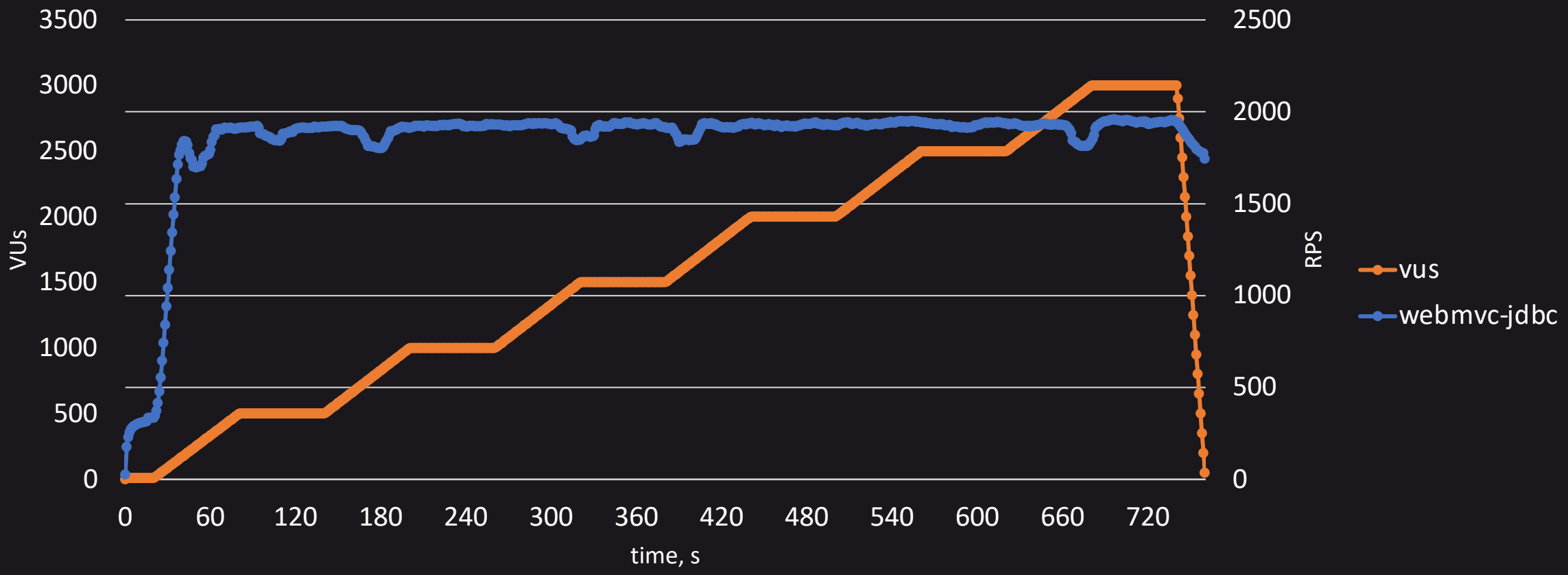
pg_sleep()


```kotlin
// R2dbcReactiveClinicService.kt
@Transactional(transactionManager = "connectionFactoryTransactionManager", readOnly = true)
override fun sleep(times: Int, millis: Int, zip: Boolean): Mono<Unit> {
    return if (zip) {
        Mono.zip(List(times) { sleepRepository.sleep(millis) }) {}
    } else {
        sleepRepository.sleep(millis).repeat(times - 1L).then().then(Unit.toMono())
    }
}
```

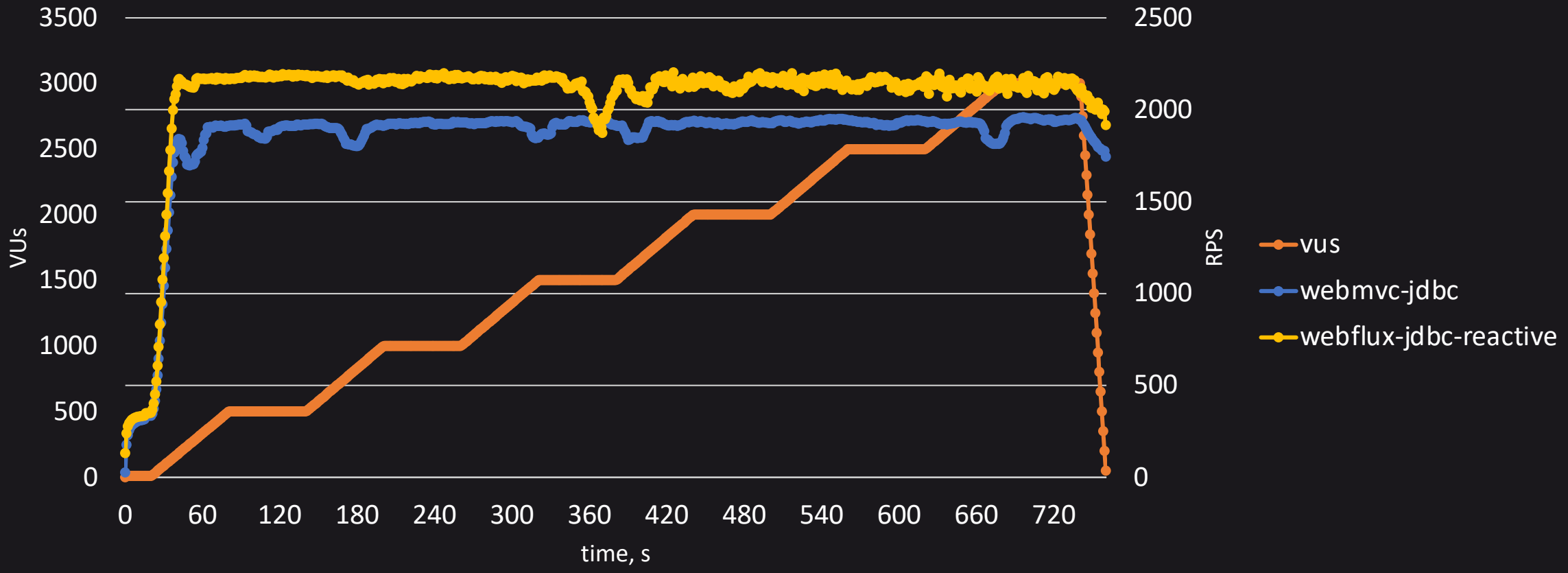
pg_sleep() с корутинами


```kotlin
// R2dbcCoroutineClinicService.kt
@Transactional(transactionManager = "connectionFactoryTransactionManager", readOnly = true)
override suspend fun sleep(times: Int, millis: Int, zip: Boolean) {
    return if (zip) {
        coroutineScope {
            val list = mutableListOf<Deferred<Unit>>()
            for (i in 1..times) {
                list += async { sleepRepository.sleep(millis) }
            }
            list.forEach { it.await() }
        }
    } else {
        for (i in 1..times) {
            sleepRepository.sleep(millis)
        }
    }
}
```

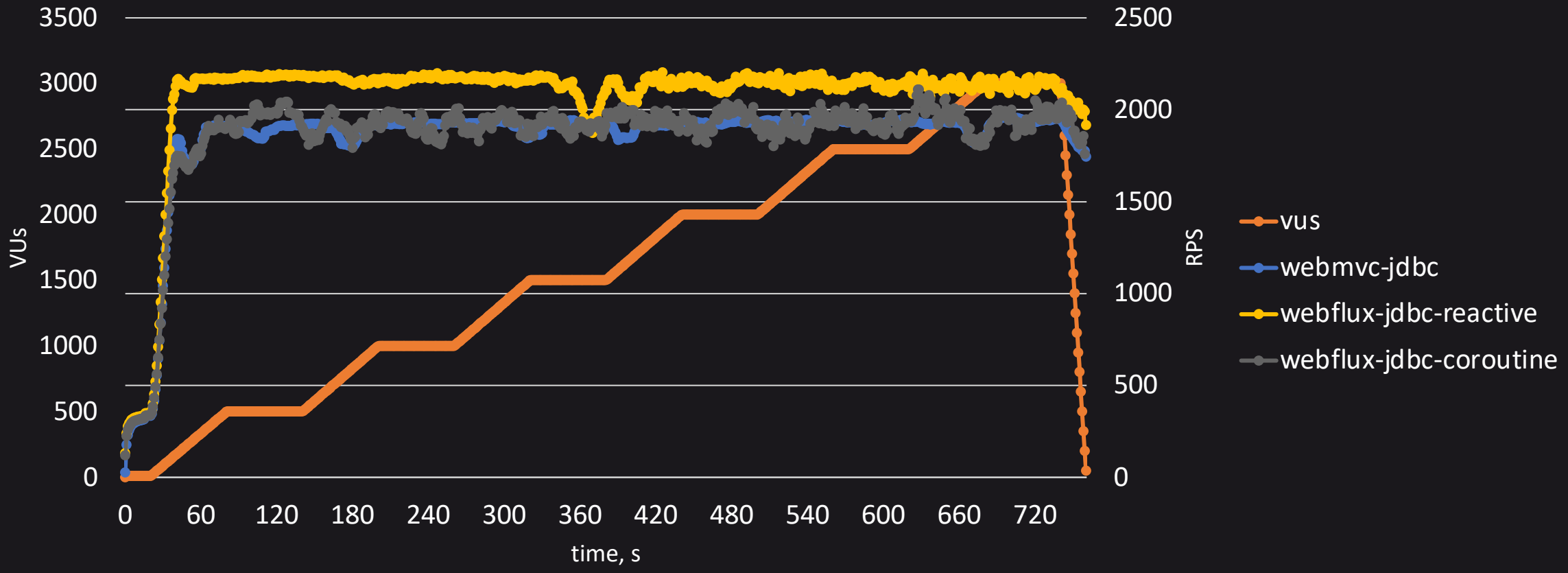
Sleep: 10 раз по 2 мс: RPS



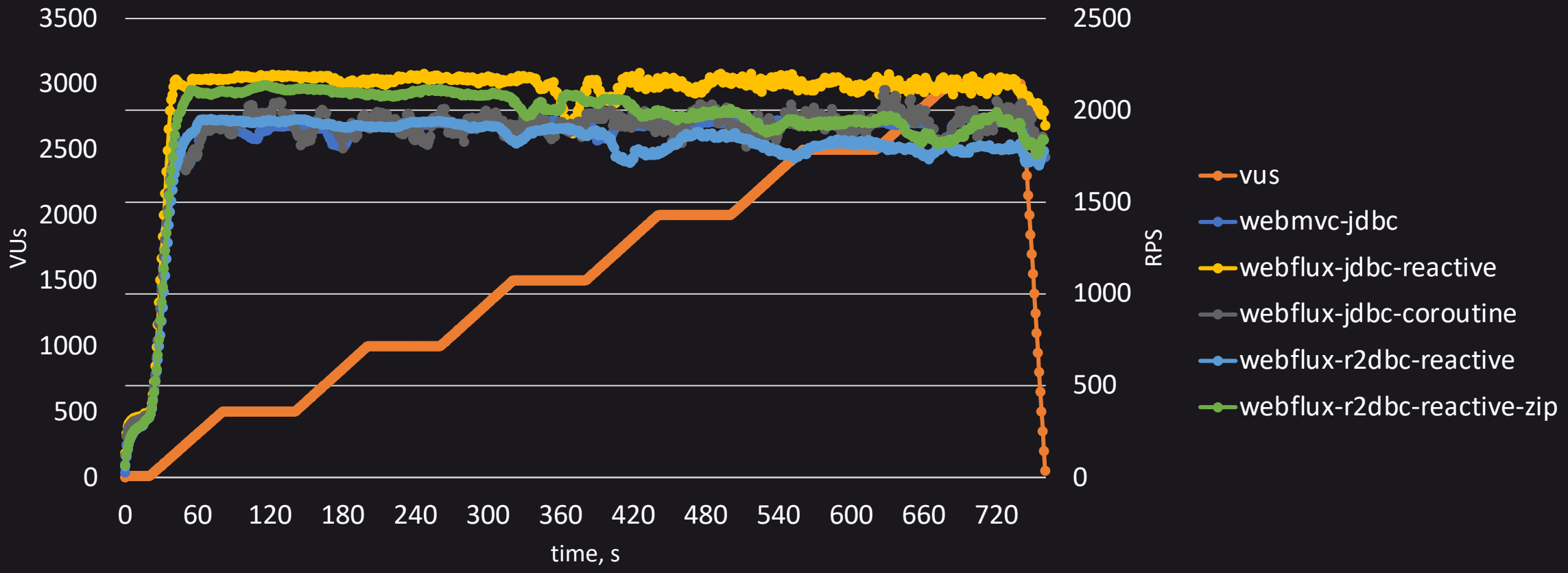
Sleep: 10 раз по 2 мс: RPS



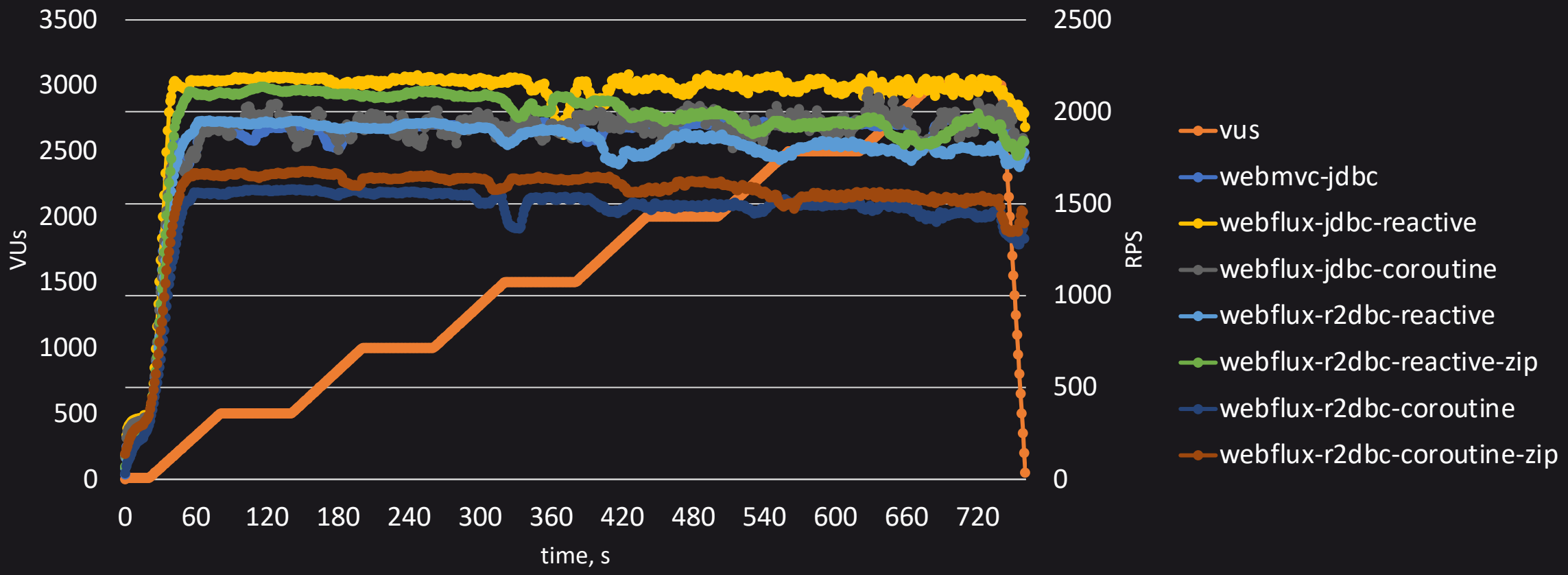
Sleep: 10 раз по 2 мс: RPS



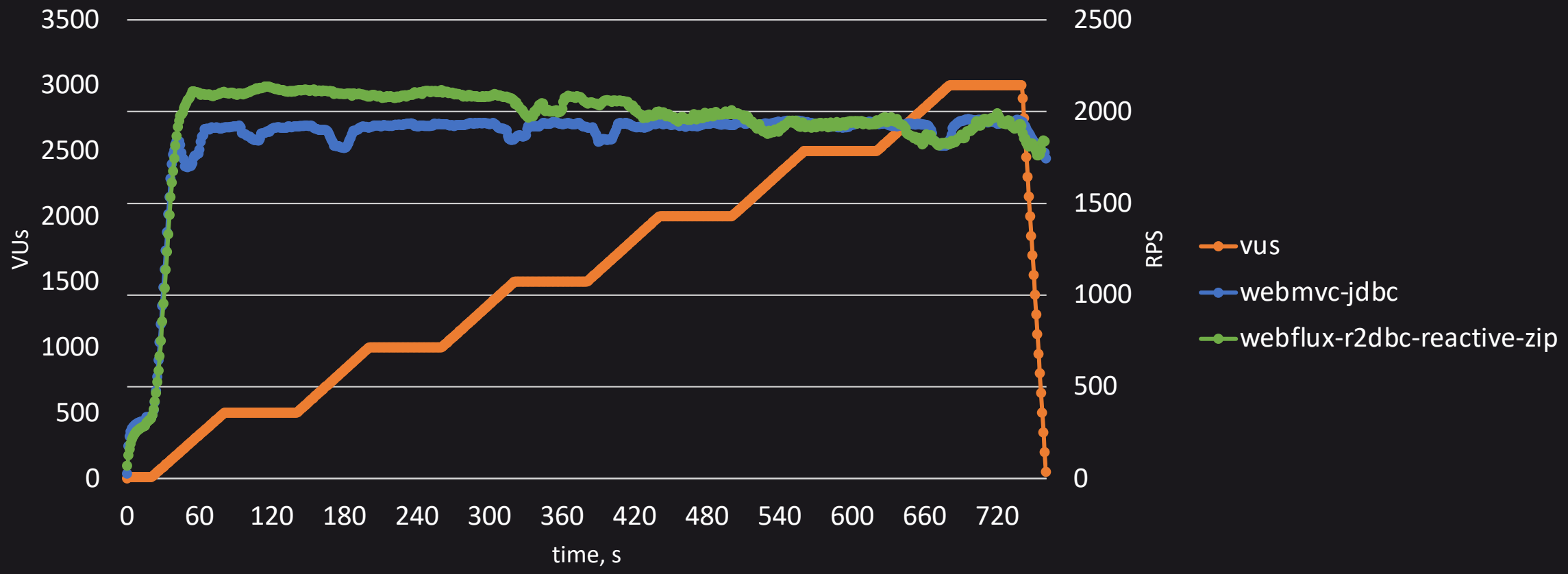
Sleep: 10 раз по 2 мс: RPS



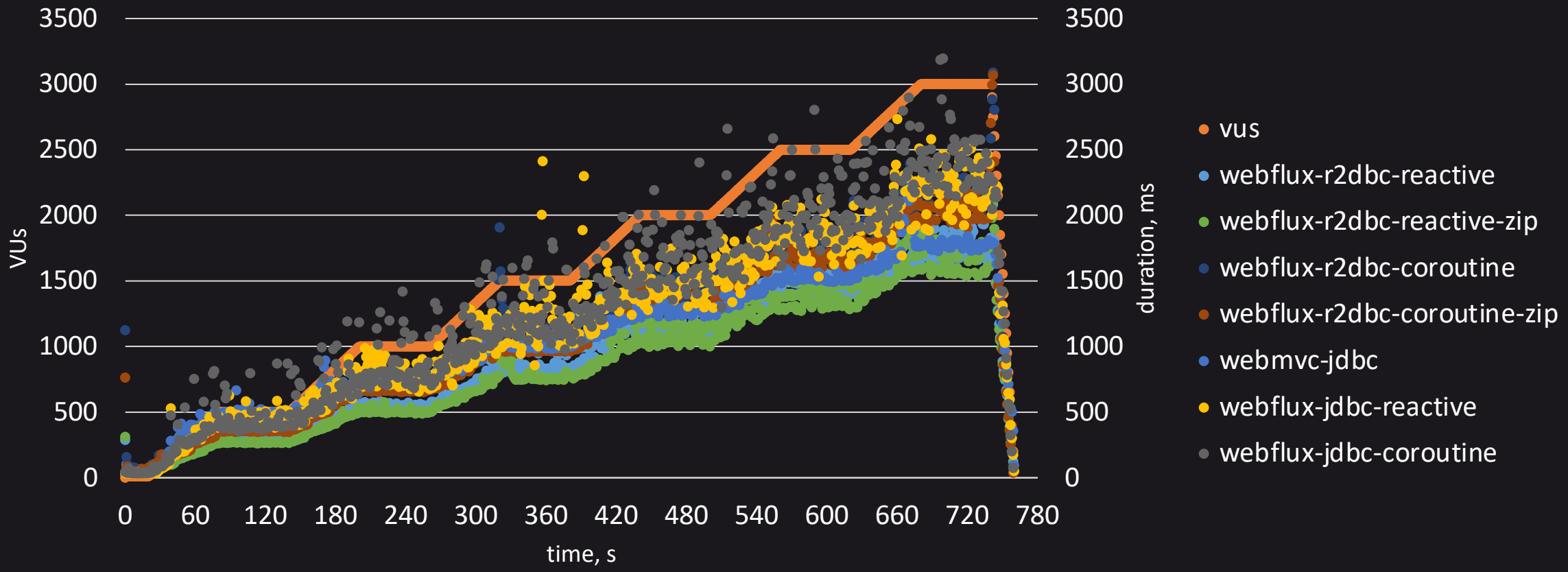
Sleep: 10 раз по 2 мс: RPS



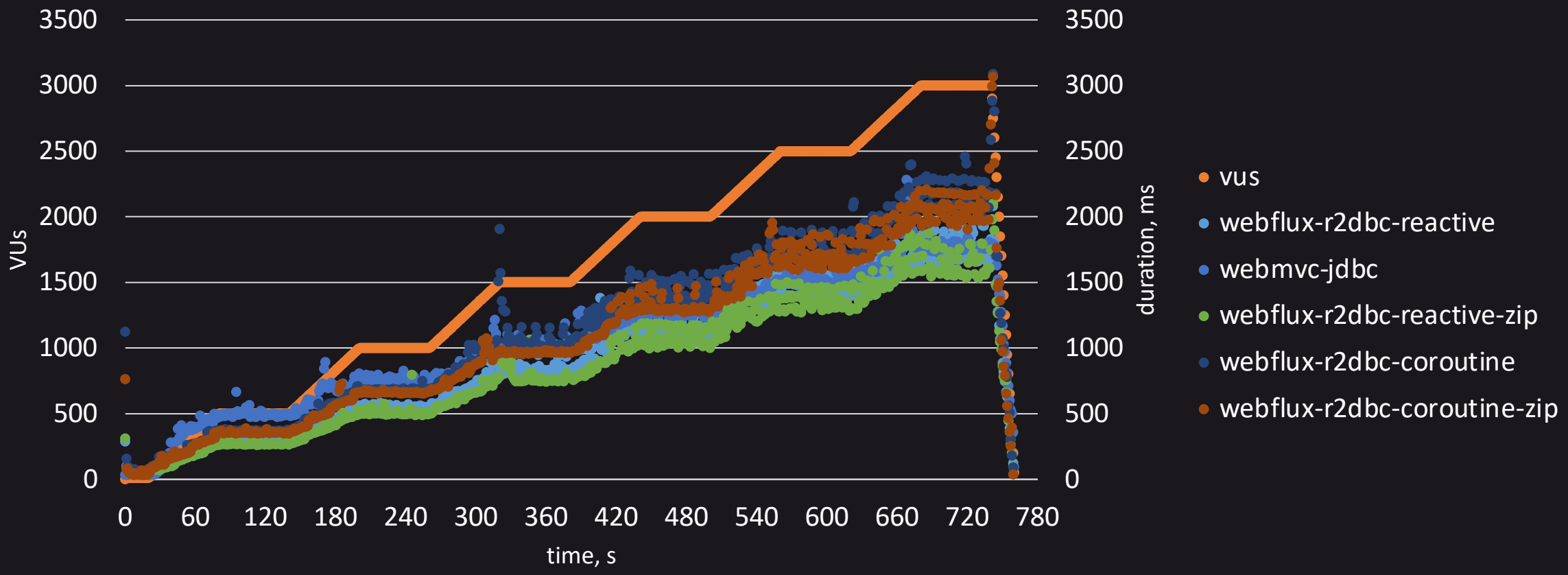
Sleep: 10 раз по 2 мс: RPS



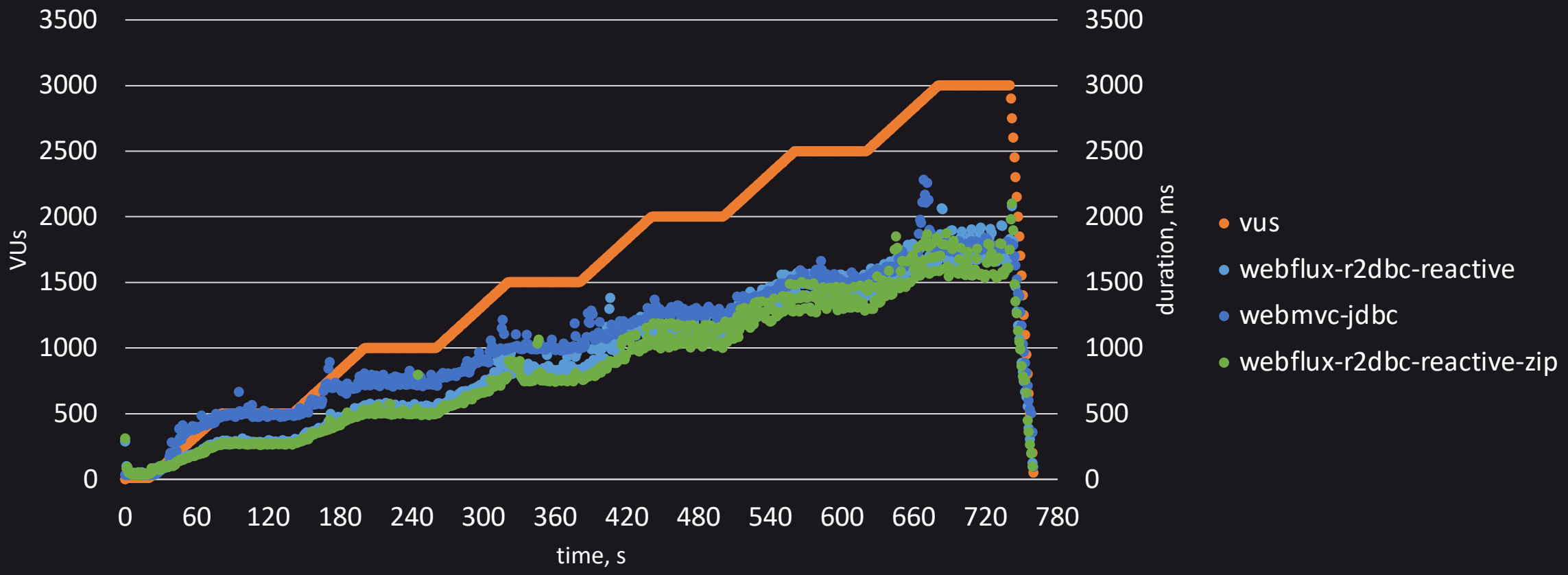
Sleep: 10 раз по 2 мс: request duration: p(99)



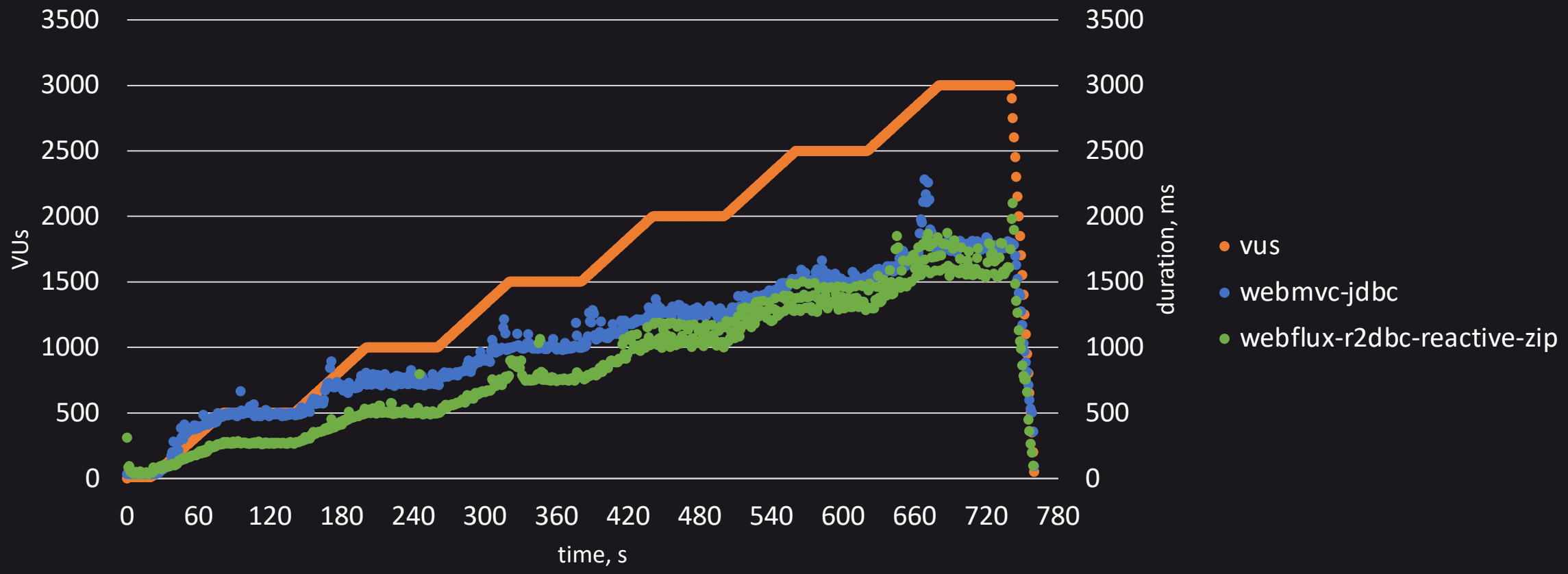
Sleep: 10 раз по 2 мс: request duration: p(99)



Sleep: 10 раз по 2 мс: request duration: p(99)



Sleep: 10 раз по 2 мс: request duration: p(99)



Выводы

1. В 99.99% случаев используем webmvc-jdbc.
2. Если все-таки нужен "микро-монолит", то используем webflux-jdbc (можно на корутинах).
3. Loom с реляционными базами не используем.
4. R2DBC не используем, тем более в связке с корутинами.
5. Проводим нагрузочное тестирование.

Ростелеком
Информационные Технологии

Спасибо за внимание

Антон Котов

telegram

kotoant