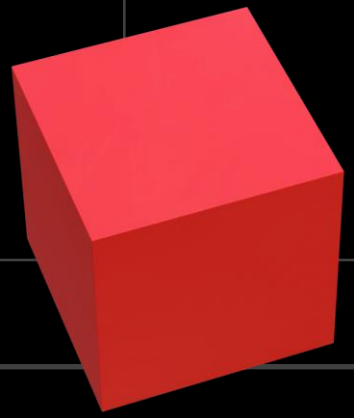
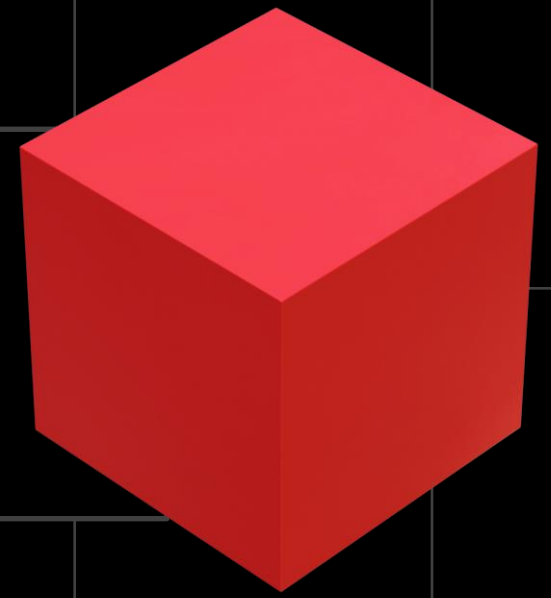
Прогревая JVM

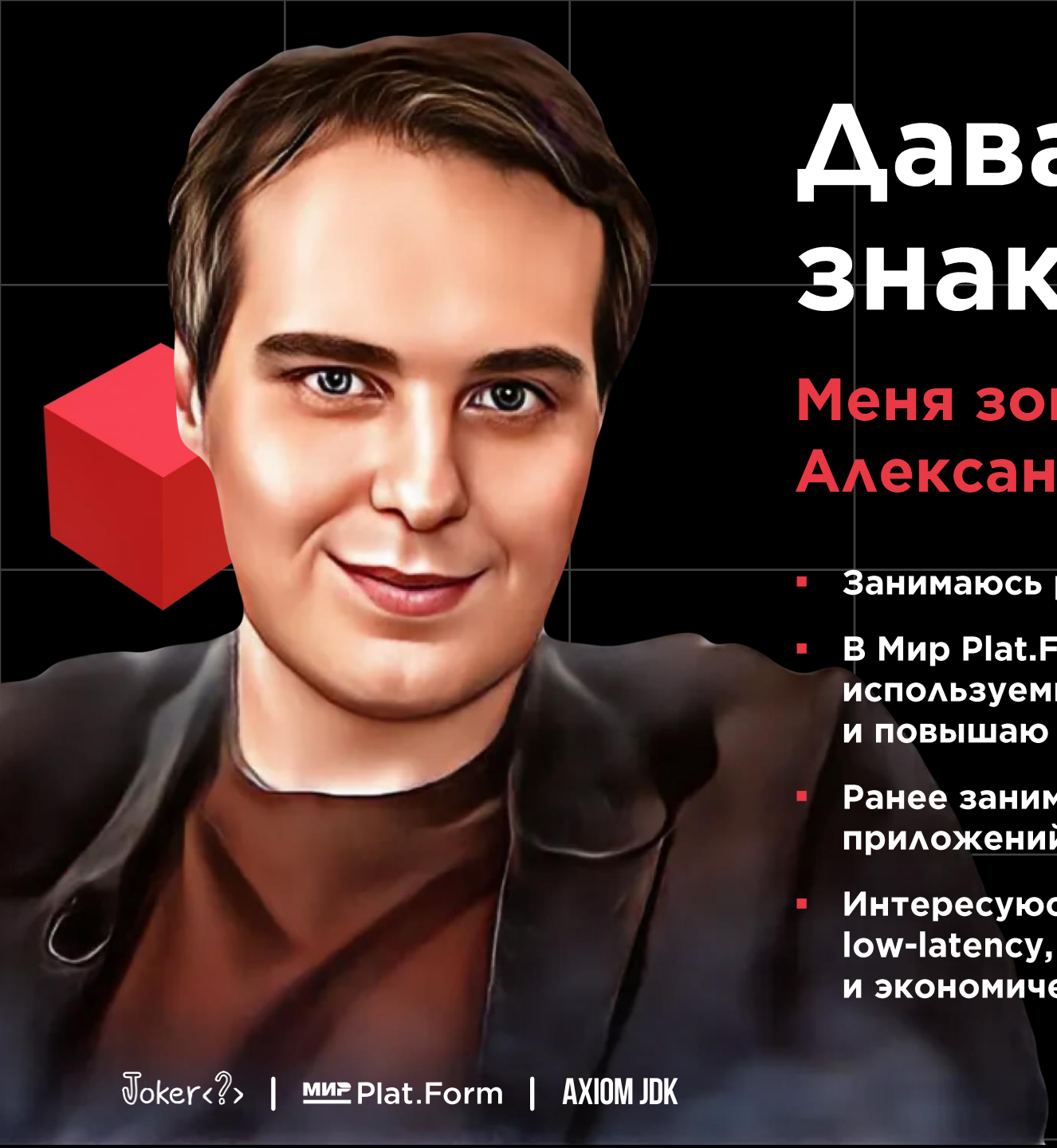
CRaC и другие фокусы

2023





Joker<?> | МИР Plat.Form | AXIOM JDK



Давайте знакомится

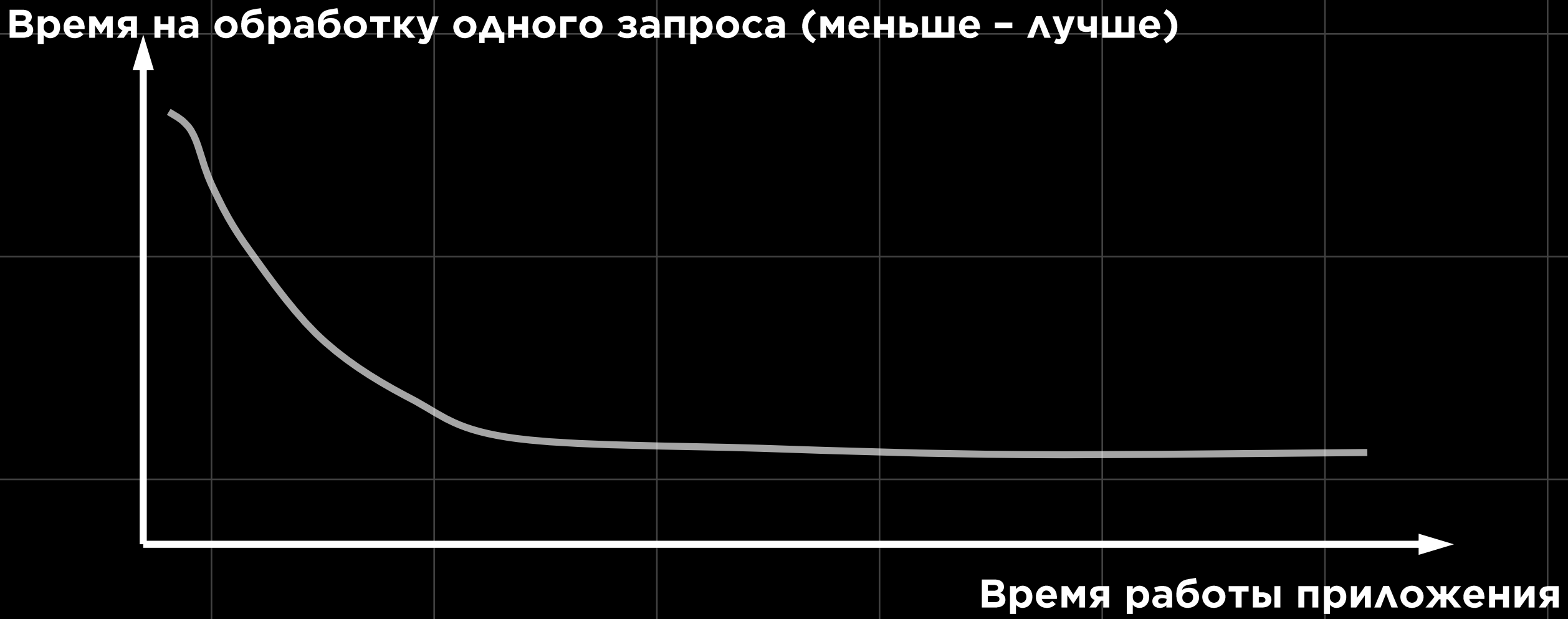
Меня зовут Александр Ланцов

- Занимаюсь разработкой в финтехе более восьми лет
- В Мир Plat.Form разрабатываю расчетные системы, используемые в платежной системе «Мир» и повышаю их производительность
- Ранее занимался разработкой алготрейдинговых приложений для крупных участников рынка
- Интересуюсь разработкой на JVM-платформе, low-latency, многопоточностью и экономическими моделями

Joker<?> | МИР Plat.Form | AXIOM JDK

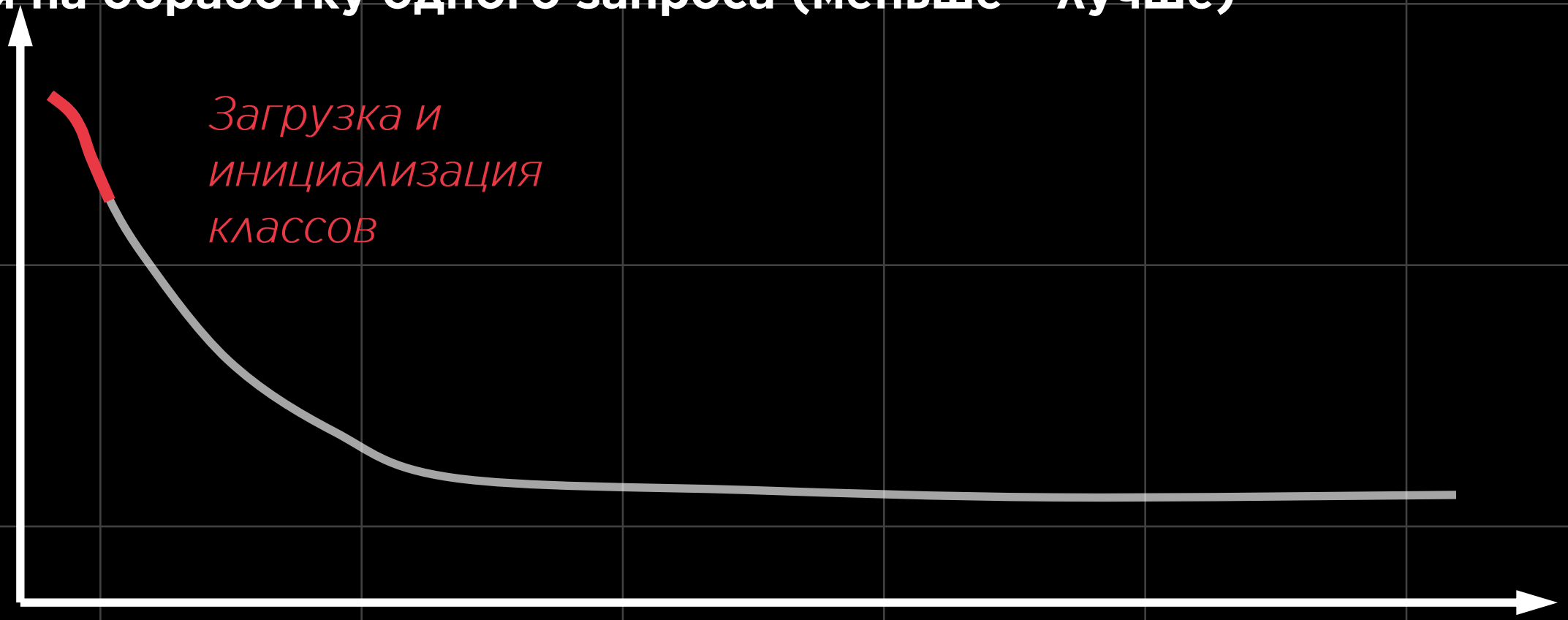
Критерии оценивания

Критерии оценивания



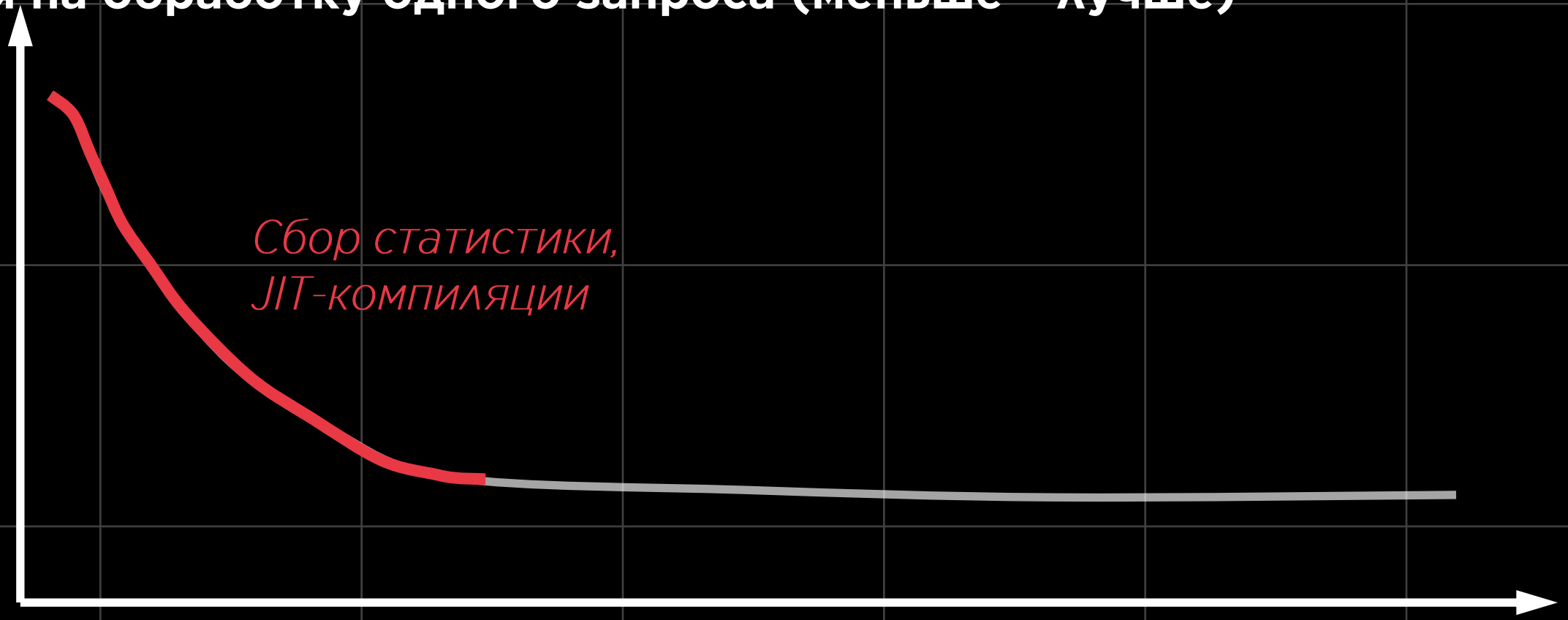
Критерии оценивания

Время на обработку одного запроса (меньше – лучше)

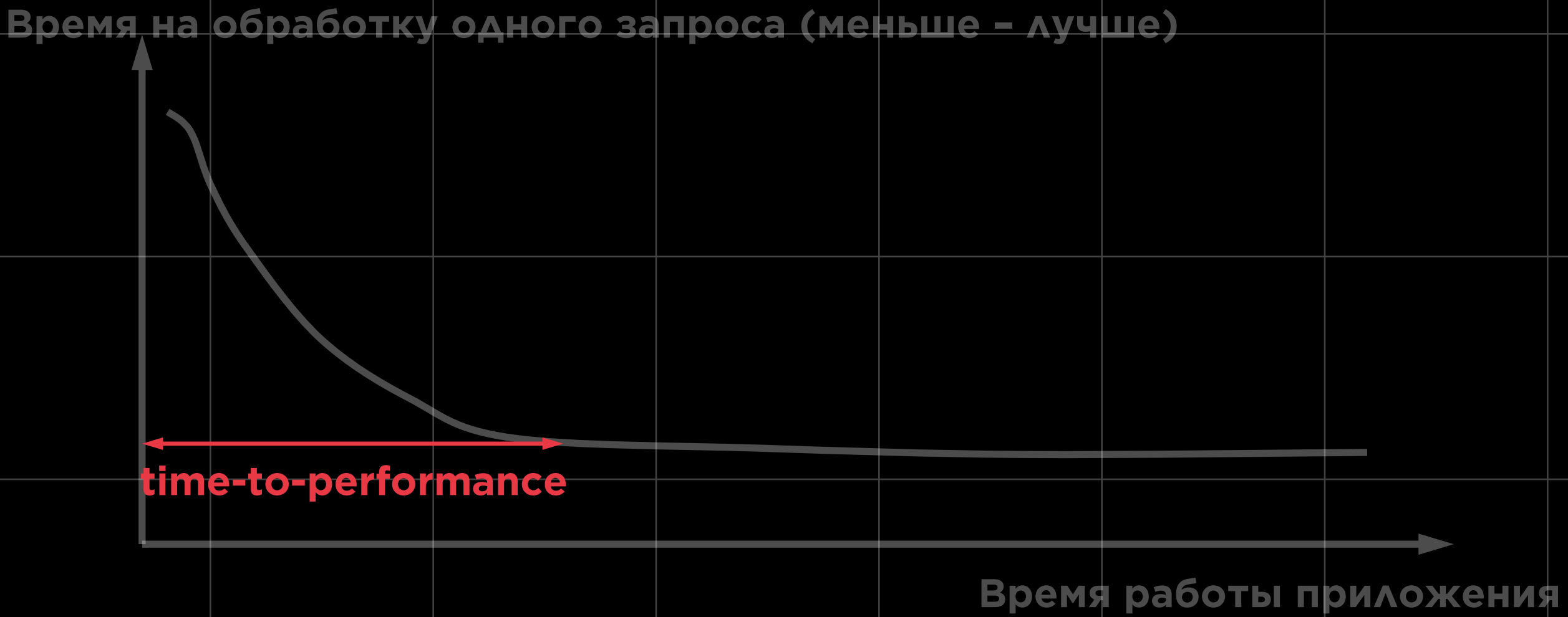


Критерии оценивания

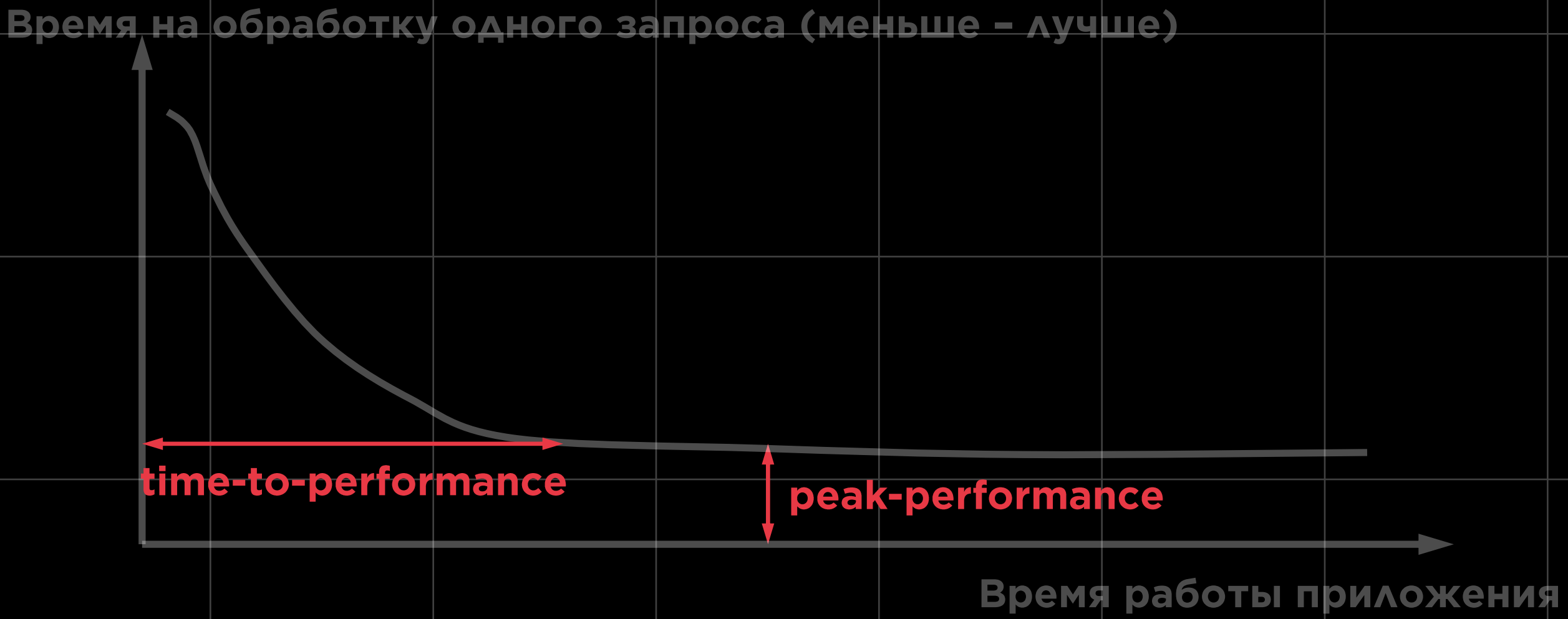
Время на обработку одного запроса (меньше – лучше)



Критерии оценивания

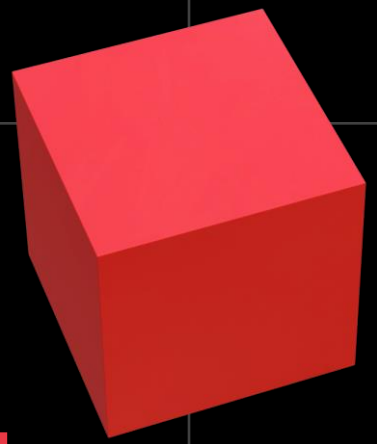


Критерии оценивания



Критерии оценивания

time-to-performance [ms]



- Время выхода на пиковую производительность
- Хотим, чтобы было поменьше

peak-throughput-performance [штук/ms]

- Сколько запросов в единицу времени обрабатываем после прогрева
- Хотим, чтобы было побольше

complexity

- Насколько сложно применить к существующему коду

Когда прогрев важен?

function-as-service

- Платим деньги за время работы

масштабирование в зависимости от нагрузки

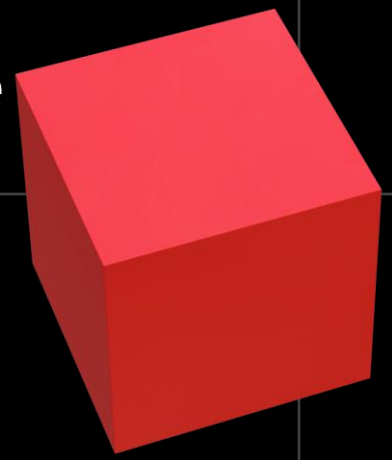
- Новые экземпляры должны работать так же быстро, как старые

CLI-инструменты

- Сценарий использования скорее всего один и тот-же

алготрейдинг

- Торговать нужно сразу на максимальной скорости



Когда прогрев важен?

function-as-service

- Платим деньги за время работы

масштабирование в зависимости от нагрузки

- Новые экземпляры должны работать так же быстро, как старые

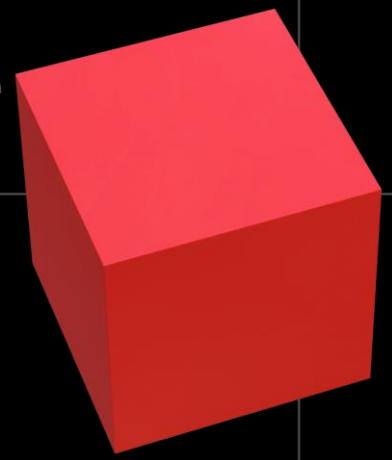
CLI-инструменты

- Сценарий использования скорее всего один и тот-же

алготрейдинг

- Торговать нужно сразу на максимальной скорости

В зависимости от приложения, прогрев нужен разный!



Подходы

- Переписать на C++/Rust/etc
- AOT-компиляция
- Использование флагов JVM
- Ручной прогрев
- Экзотические JVM: Azul Prime Platform и Alibaba Dragonwell
- Coordinated Restore at Checkpoint (CRaC)

Подходы

- Переписать на C++/Rust/etc
- AOT-компиляция
- Использование флагов JVM
- Ручной прогрев
- Экзотические JVM: Azul Prime Platform и Alibaba Dragonwell
- Coordinated Restore at Checkpoint (CRaC)

Не рассматриваем:

- Class-Data-Sharing (CDS)
- Shared-Class-Cache (Eclipse OpenJ9)

Подходы

- Переписать на C++/Rust/etc
- AOT-компиляция
- Использование флагов JVM
- Ручной прогрев
- Экзотические JVM: Azul Prime Platform и Alibaba Dragonwell
- Coordinated Restore at Checkpoint (CRaC)

Не рассматриваем:

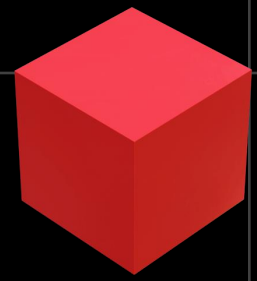
- Class-Data-Sharing (CDS)
- Shared-Class-Cache (Eclipse OpenJ9)

Использование флагов JVM

time-to-performance – ?

peak-throughput-performance – ?

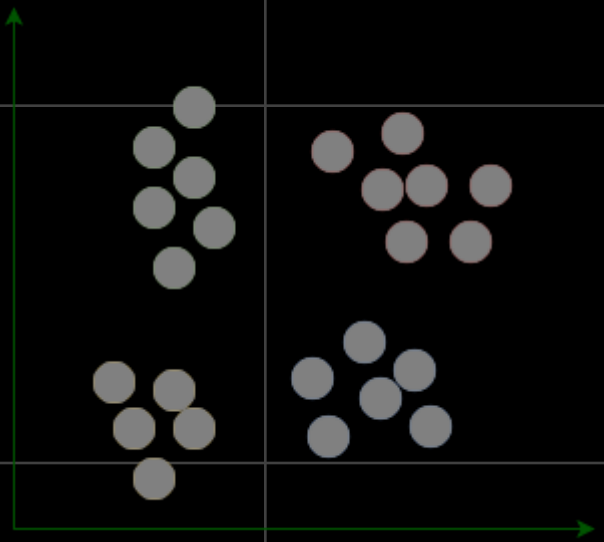
complexity - кажется, все просто



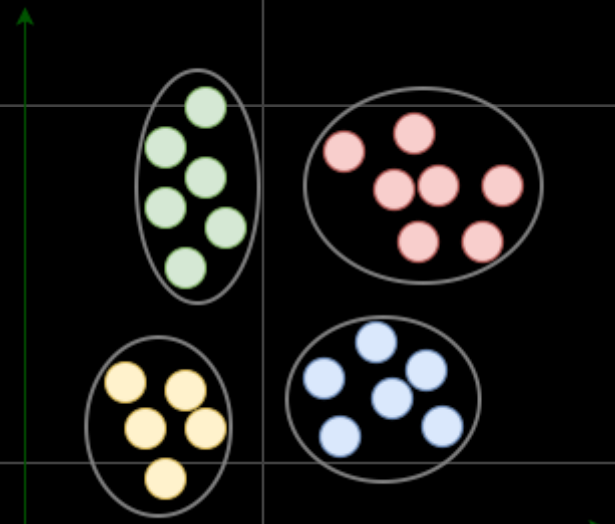
Использование флагов JVM

Бенчмарк Renaissance Benchmarking Suite

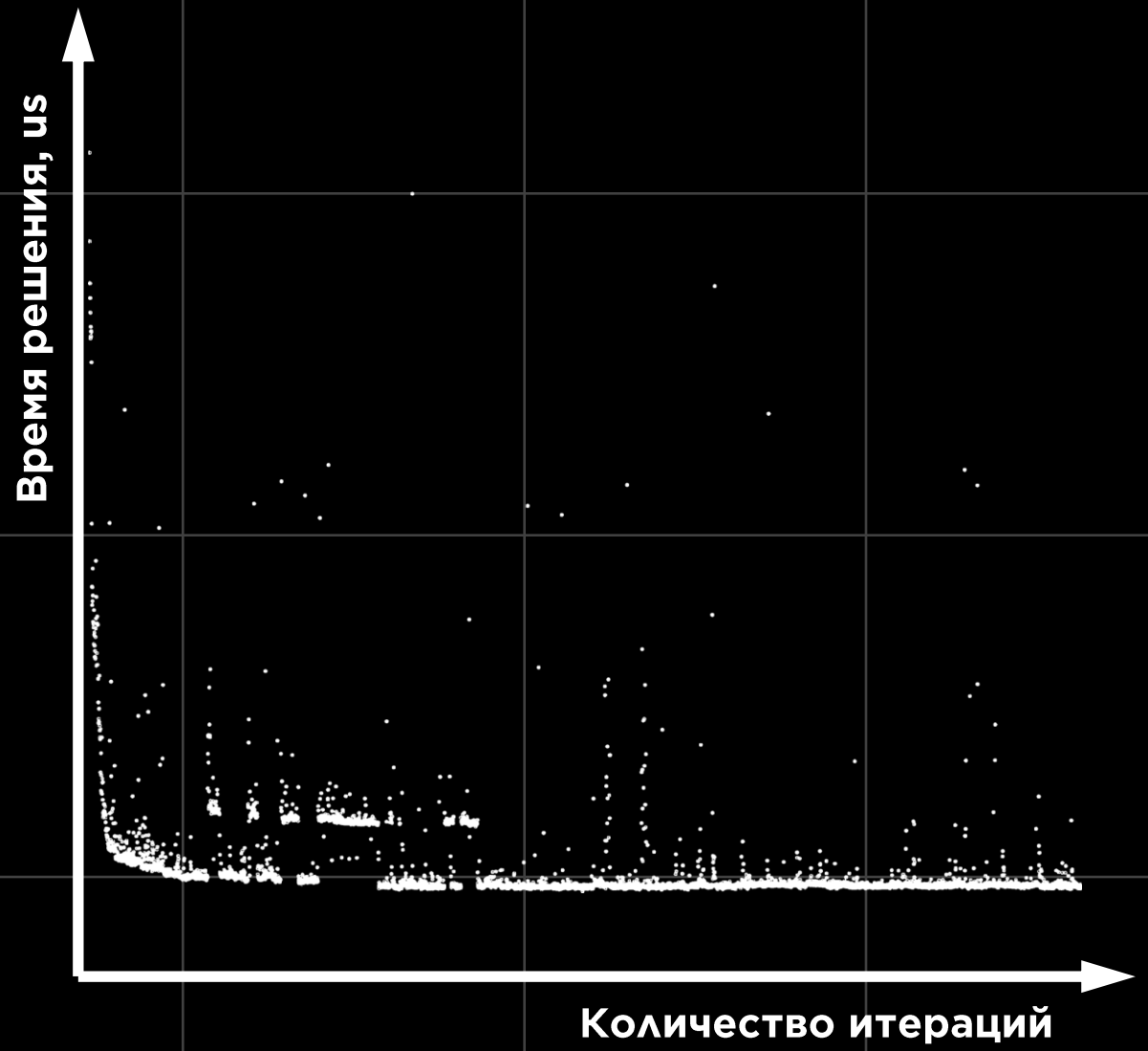
«Renaissance is a modern, open, and diversified benchmark suite for the JVM, aimed at testing JIT compilers, garbage collectors, profilers, analyzers and other tools.»



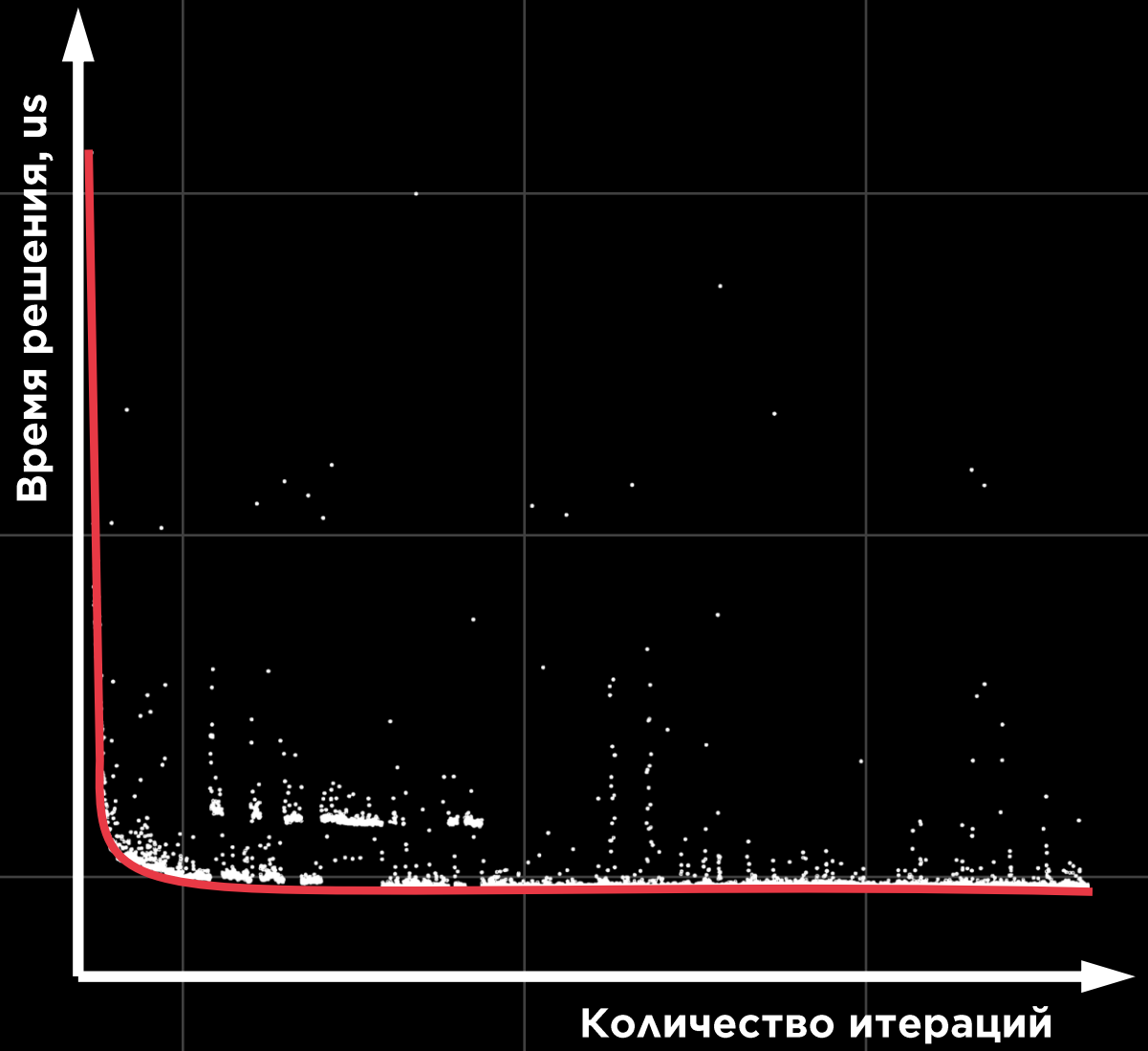
→ JavaKMeans →



JavaKMeans, стандартные настройки JVM



JavaKMeans, стандартные настройки JVM



JavaKMeans, стандартные настройки JVM

time-to-performance:
2000 итераций

peak-performance:
44 us/итерация

А почему бы просто не выставить...

-XX:CompileThreshold=1

- замысел: компилируем сразу в машинный код
- не работает ещё со времен Java 8

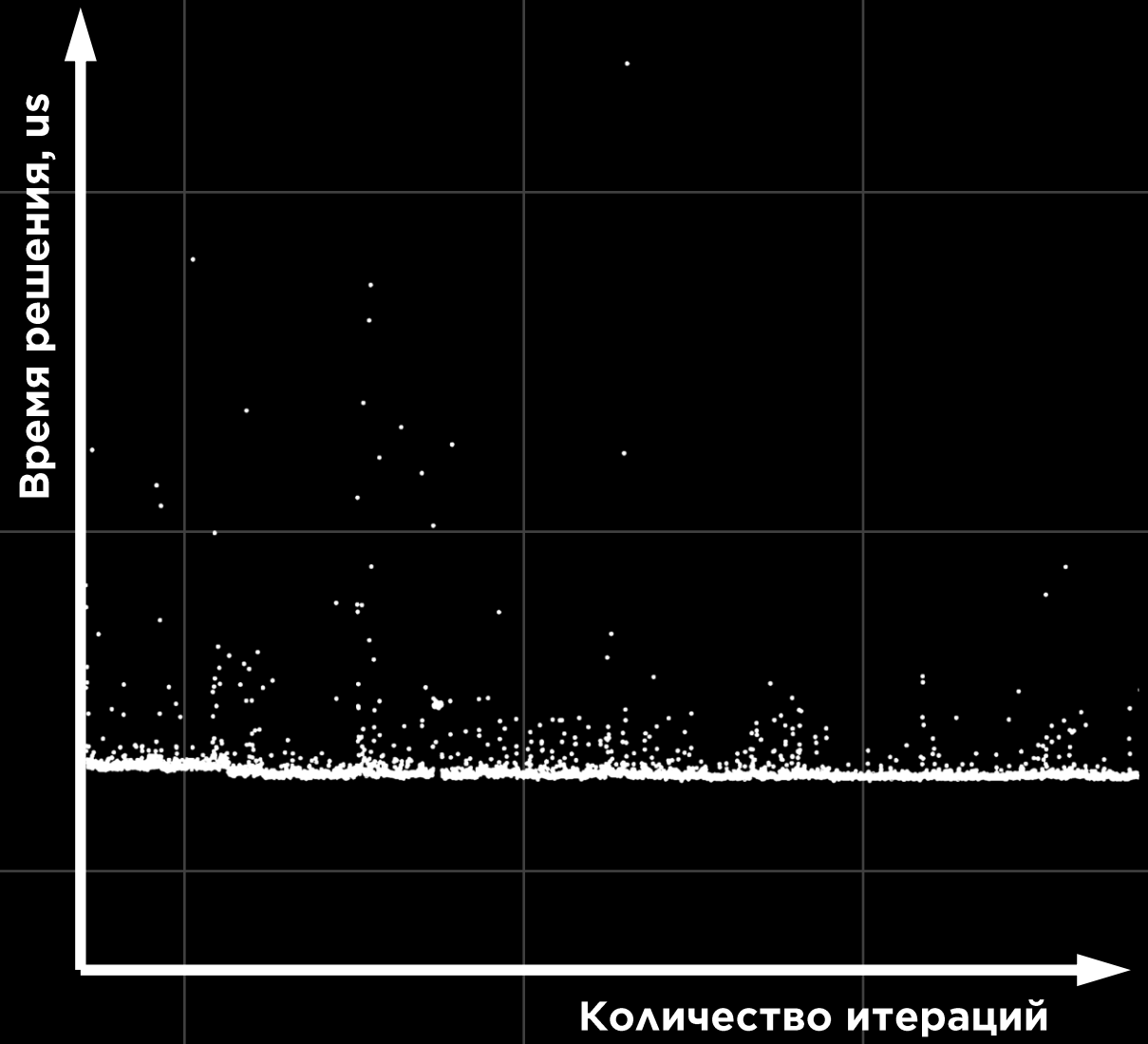
0	Интерпретатор
1	С 1 без профилирования
2	С 1 с профилированием
3	С 1 с полным профилированием
4	С 2

А почему бы просто не выставить...

-XX:-TieredCompilation -Xbatch -Xcomp

- компилируем в C2 как только можем

0	Интерпретатор
1	С 1 без профилирования
2	С 1 с профилированием
3	С 1 с полным профилированием
4	С 2

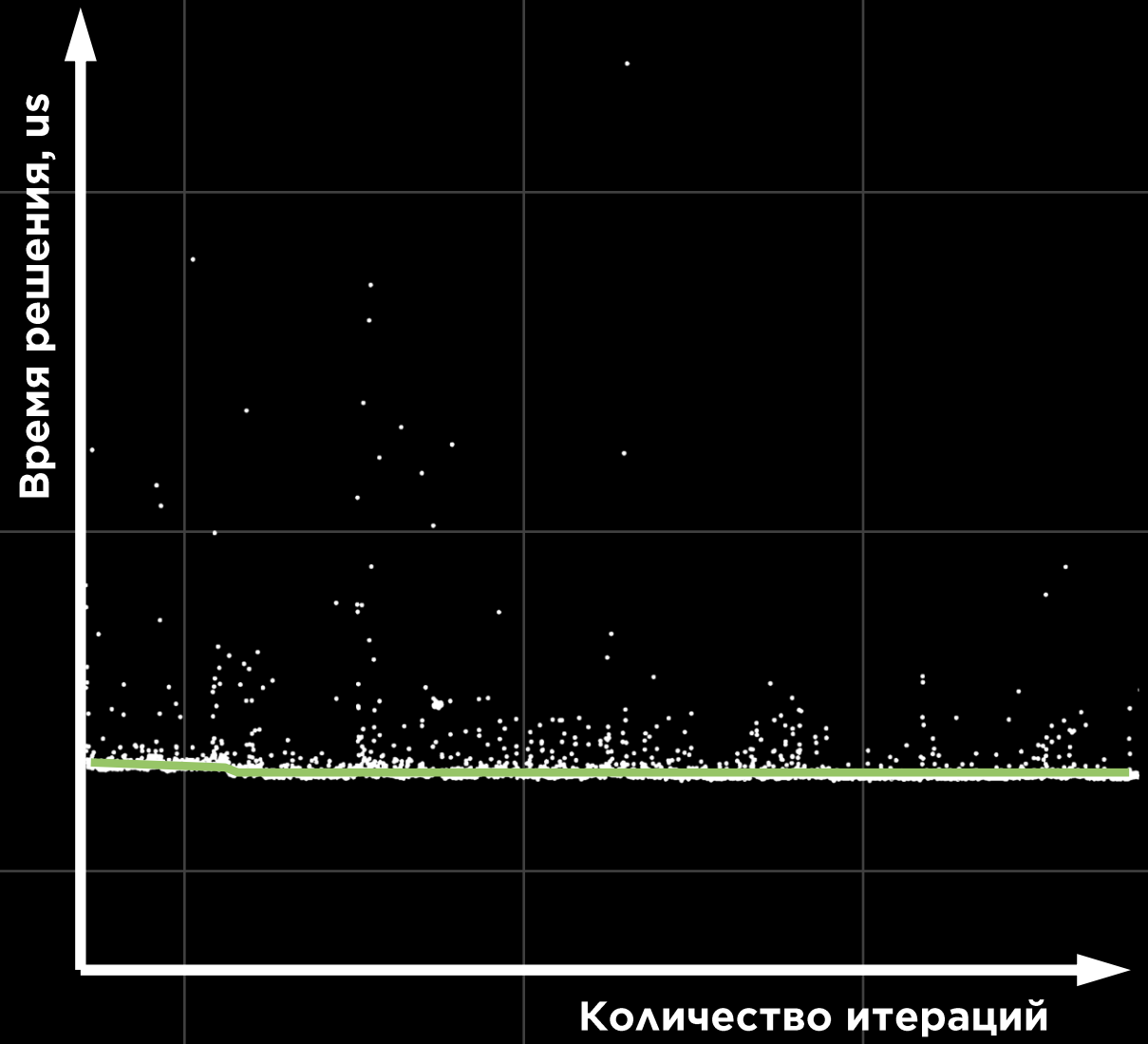


JavaKMeans, C2 asap

time-to-performance:
~0 итераций

peak-performance:
125 us/итерация

первая итерация:
140 миллисекунд

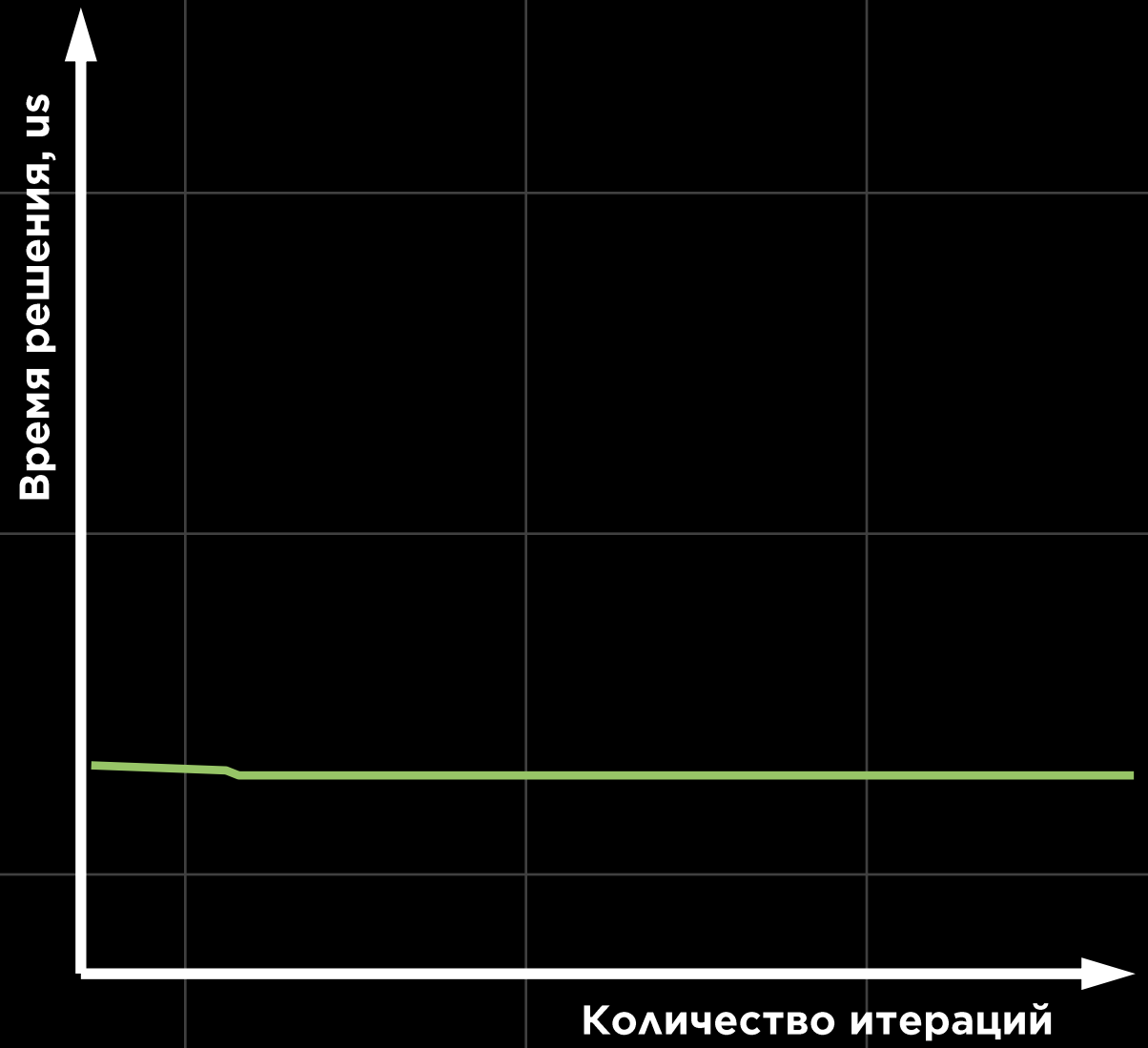


JavaKMeans, C2 asap

time-to-performance:
~0 итераций

peak-performance:
125 us/итерация

первая итерация:
140 миллисекунд



JavaKMeans, C2 asap

time-to-performance:
~0 итераций

peak-performance:
125 us/итерация

первая итерация:
140 миллисекунд

JavaKMeans, C2 asap

time-to-performance:
~0 итераций

peak-performance:
125 us/итерация

первая итерация:
140 миллисекунд

А почему бы просто не выставить...

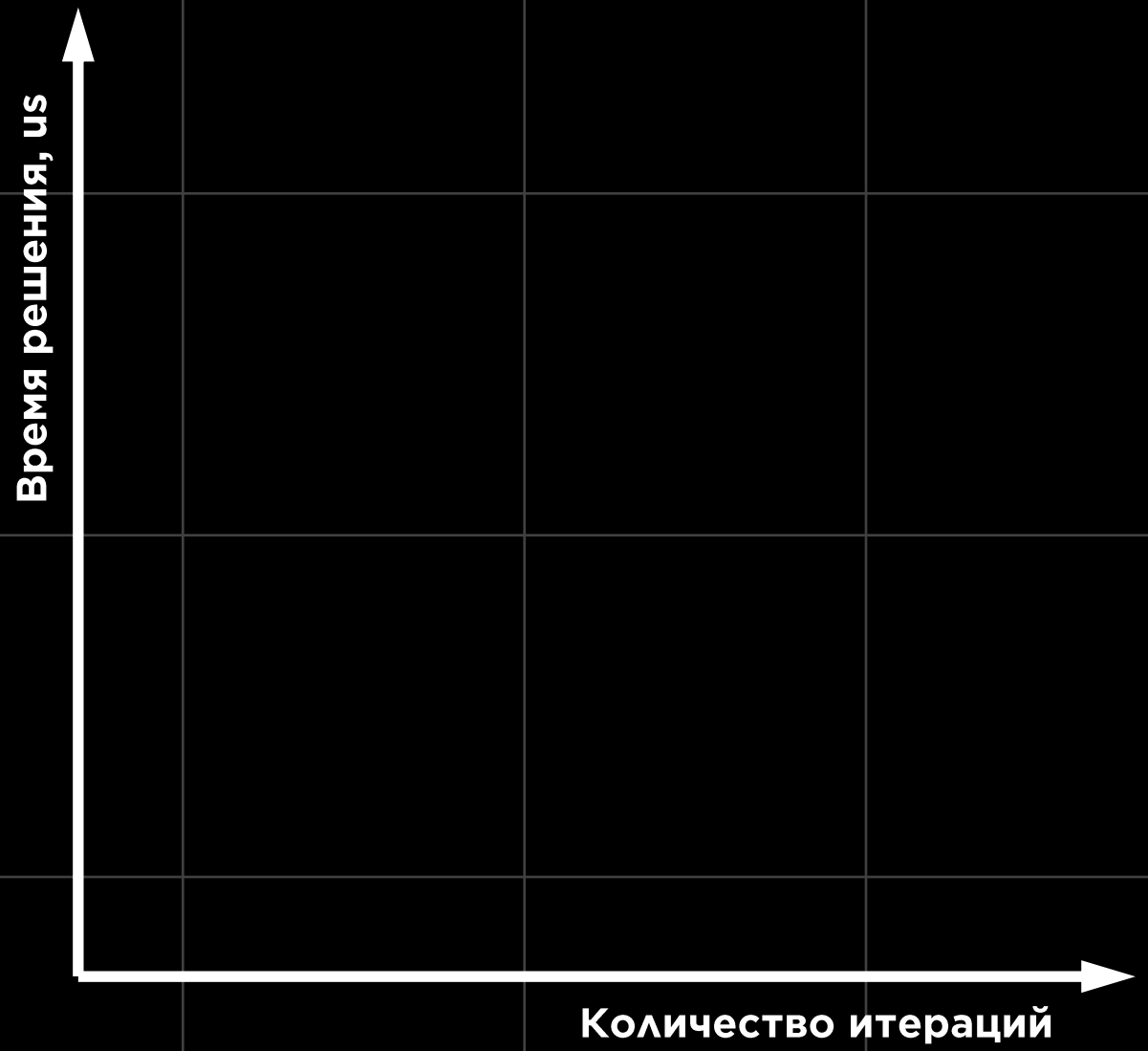
-XX:+TieredCompilation

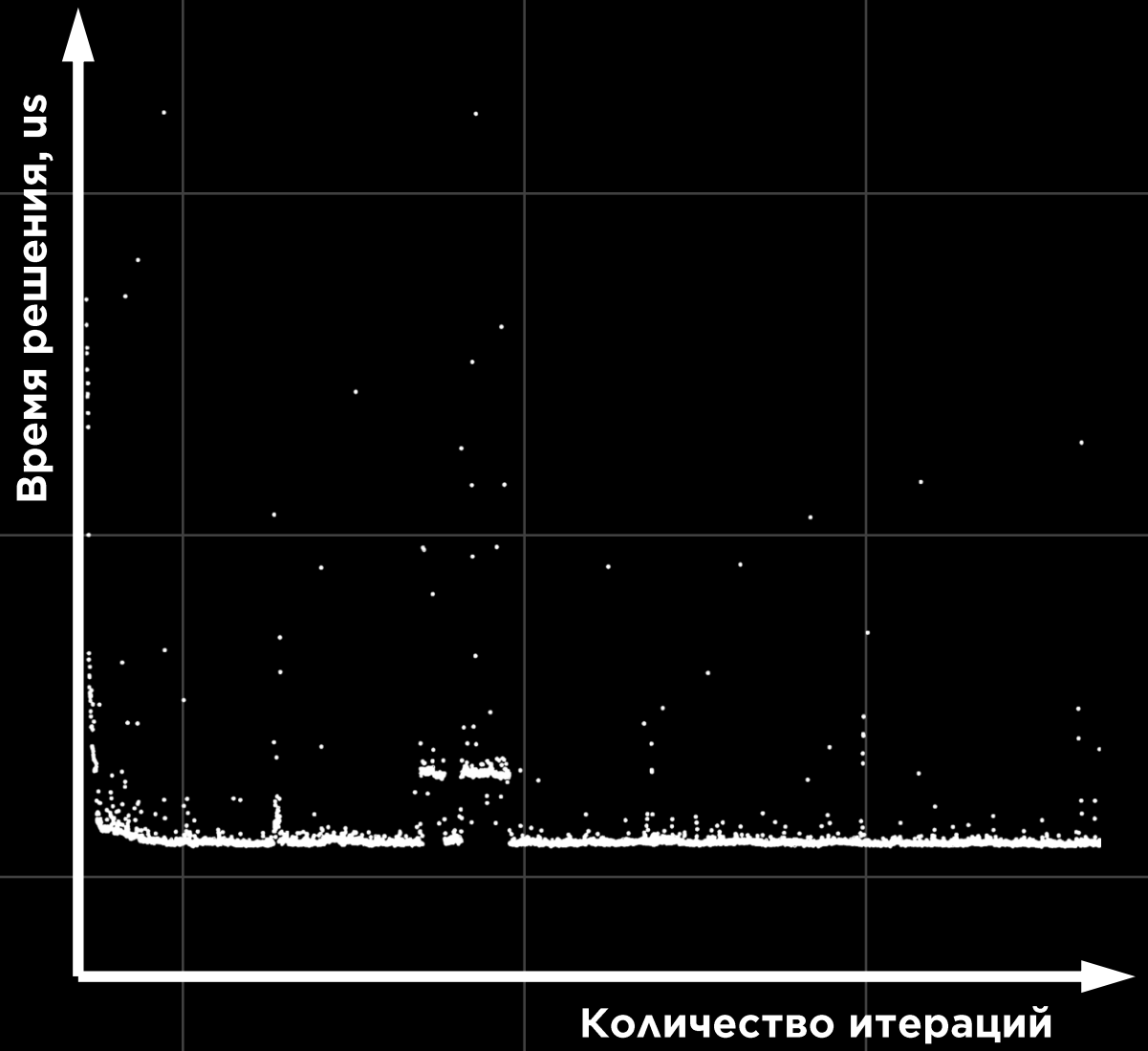
-XX:TieredStopAtLevel=1

- компилируем только в C1
- Optimizing AWS Lambda function performance for Java

0	Интерпретатор
1	С 1 без профилирования
2	С 1 с профилированием
3	С 1 с полным профилированием
4	С 2

JavaKMeans, C1 only

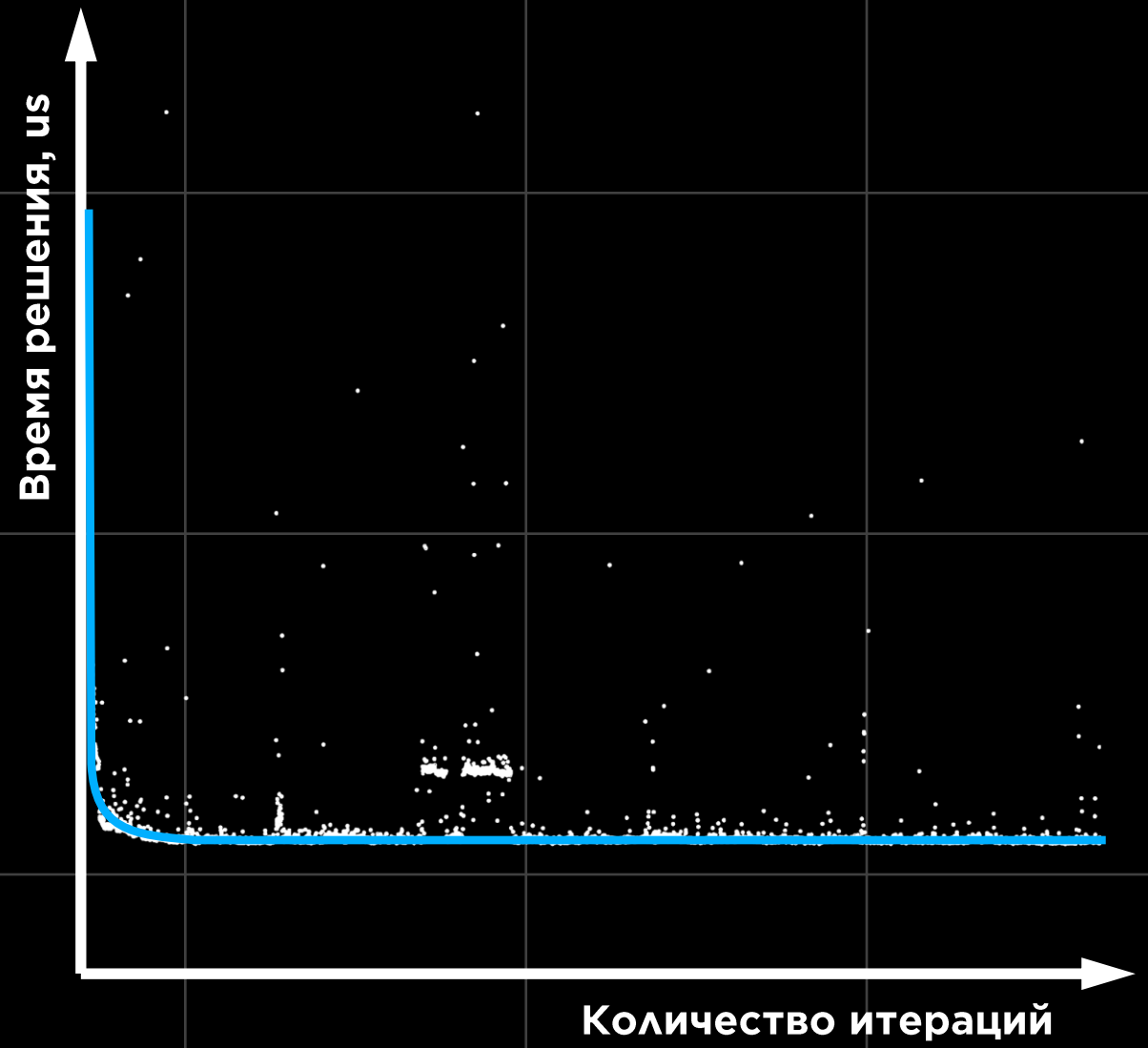




JavaKMeans, C1 only

time-to-performance:
500 итераций

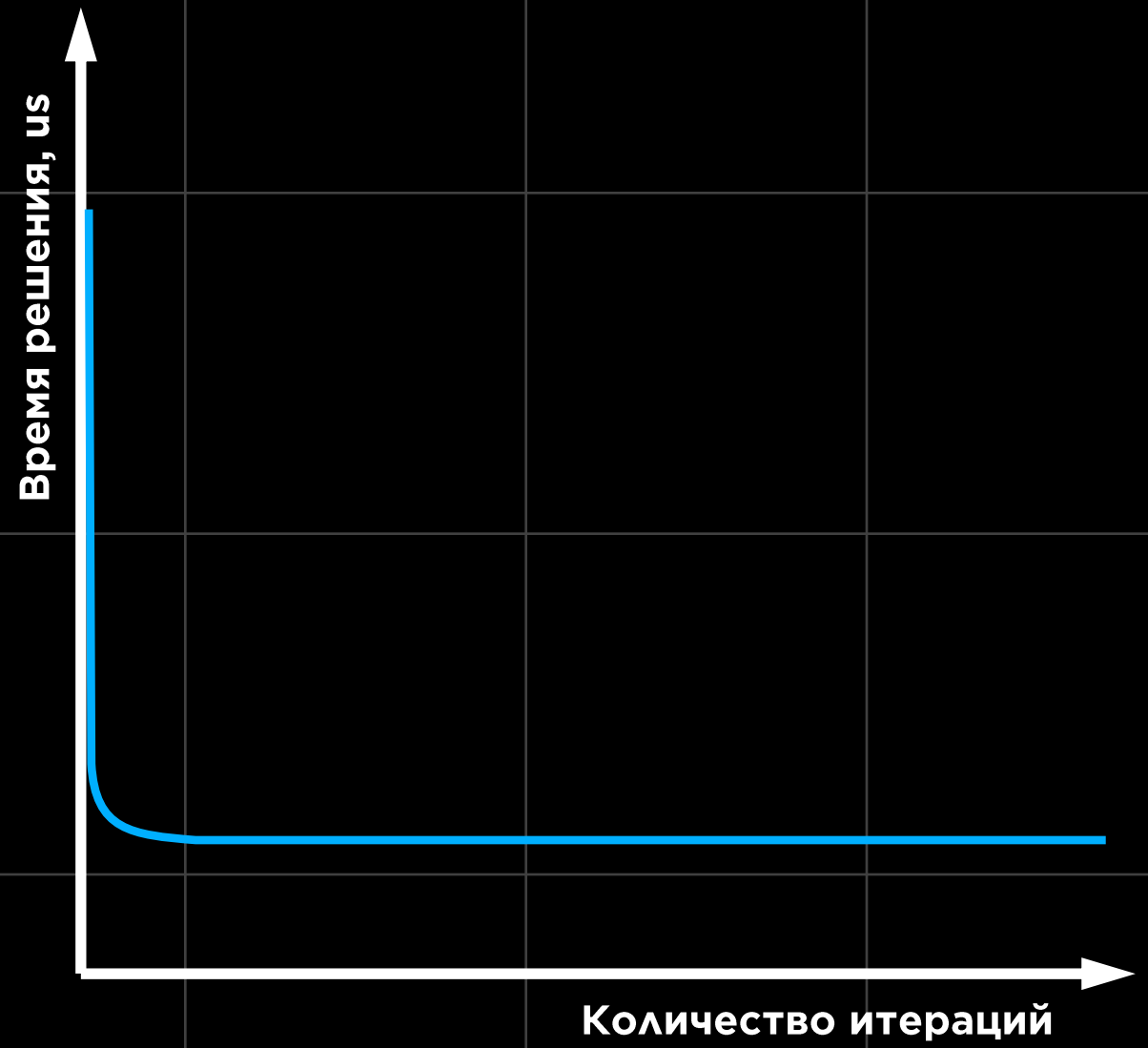
peak-performance:
80 us/итерация



JavaKMeans, C1 only

time-to-performance:
500 итераций

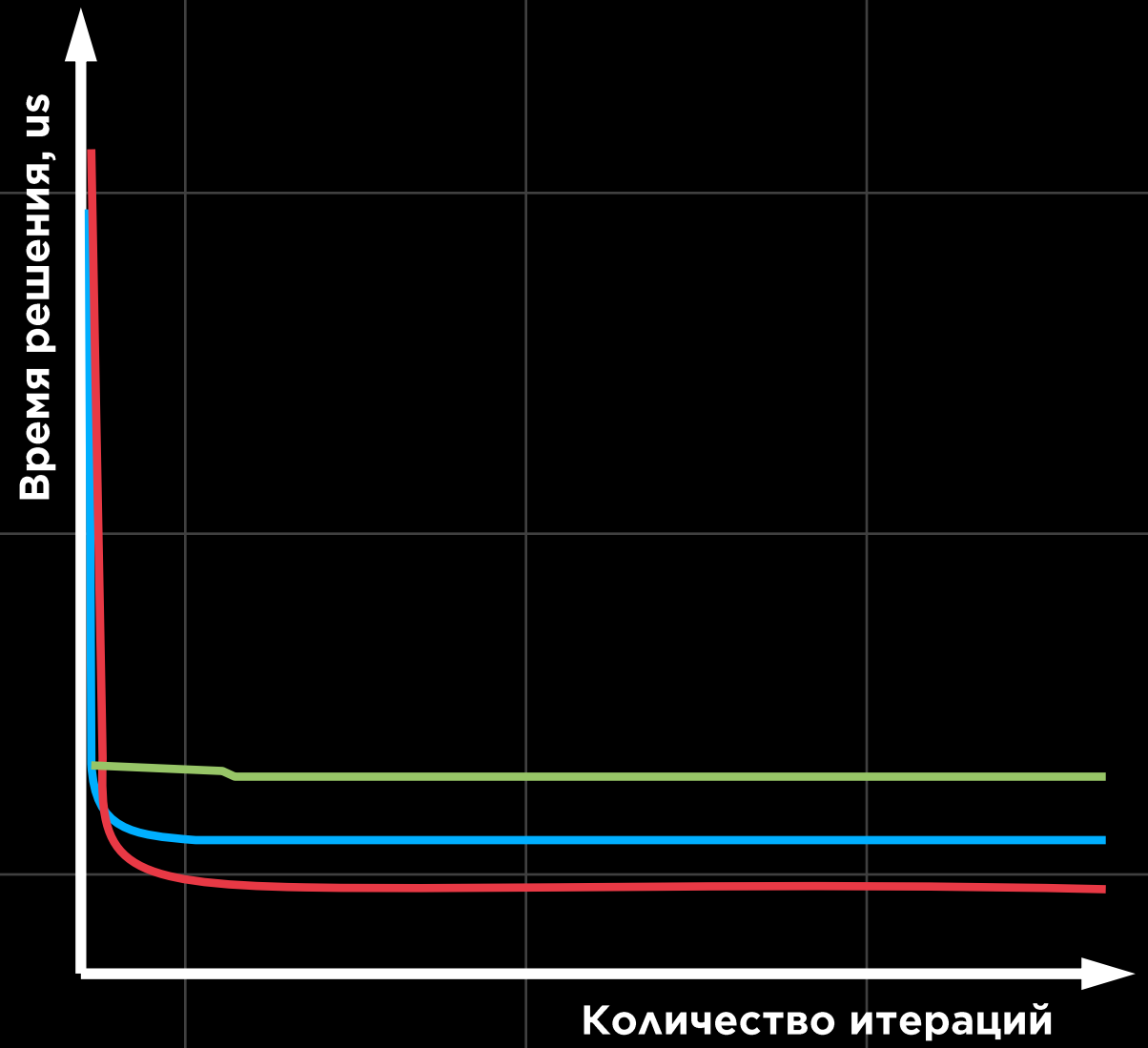
peak-performance:
80 us/итерация



JavaKMeans, C1 only

time-to-performance:
500 итераций

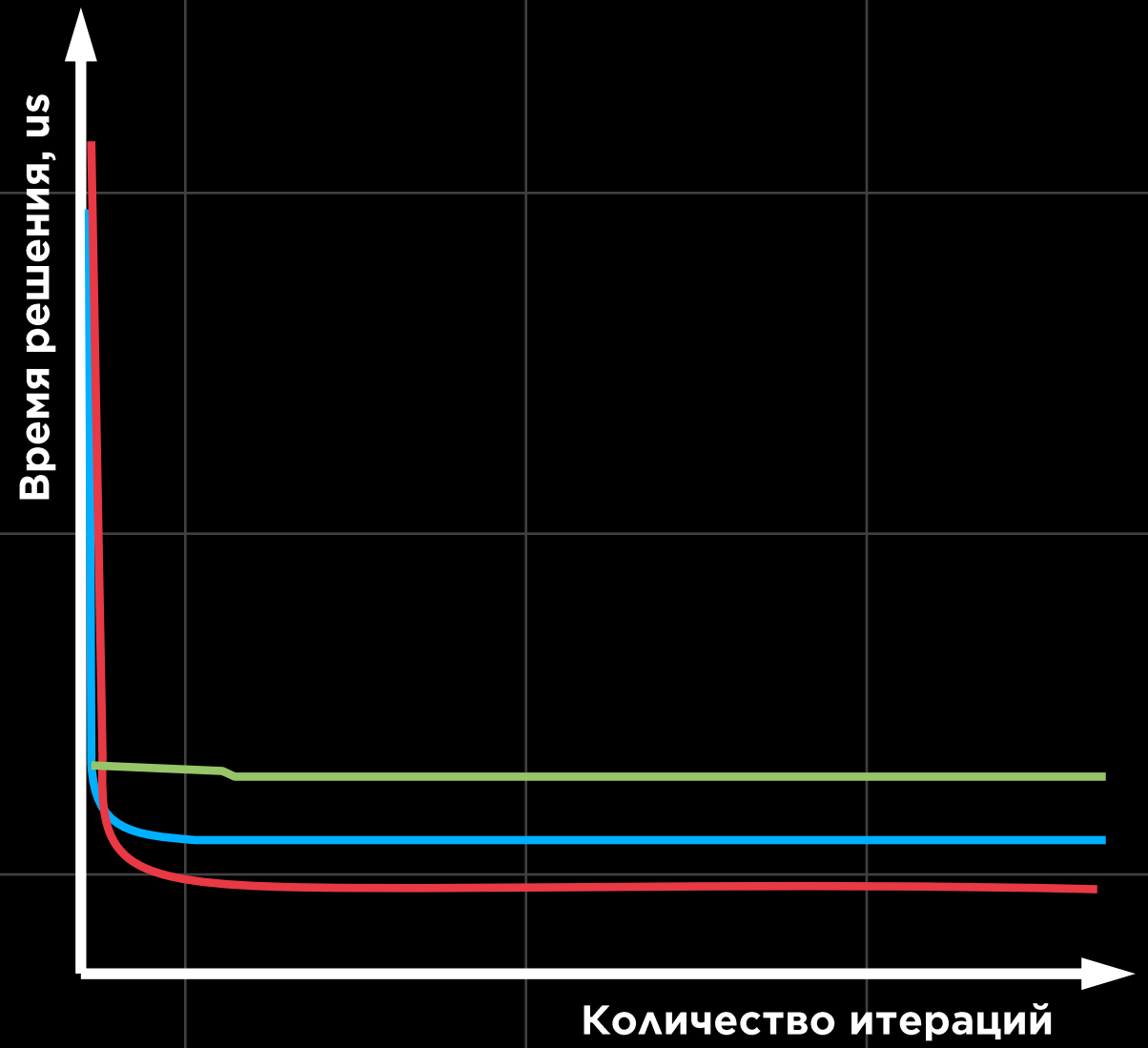
peak-performance:
80 us/итерация



JavaKMeans

time-to-performance
C2 asap: ~0 итераций
C1 only: 500 итераций
Baseline: 2000 итераций

peak-performance
C2 asap: 125 us/итерация
C1 only: 80 us/итерация
Baseline: 44 us/итерация



JavaKMeans

time-to-performance
C2 asap: ~0 итераций
C1 only: 500 итераций
Baseline: 2000 итераций

peak-performance
C2 asap: 125 us/итерация
C1 only: 80 us/итерация
Baseline: 44 us/итерация

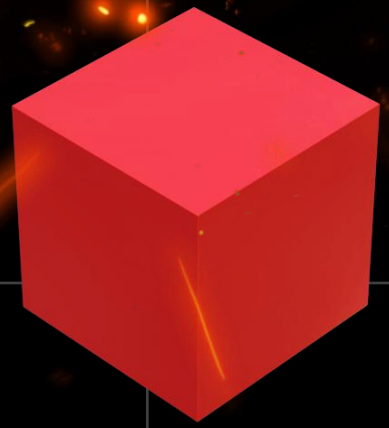
А почему бы просто не выставить...

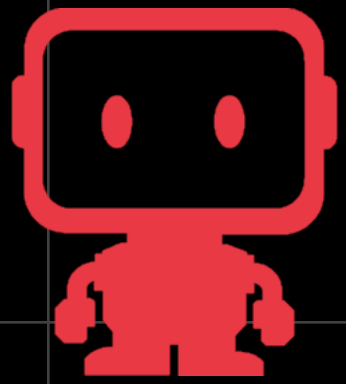
- **XX:CompileThreshold=1**
  Работало во времена динозавров (до Java 8)
- **XX:-TieredCompilation -Xbatch –Xcomp**
  Компиляция в C2 ASAP
- **XX:+TieredCompilation -XX:TieredStopAtLevel=1**
  Остановиться на C1

Просто выставить флаги не вариант

- **XX:CompileThreshold=1**
  Работало во времена динозавров (до Java 8)
- **XX:-TieredCompilation -Xbatch –Xcomp**
  Компиляция в C2 ASAP
- **XX:+TieredCompilation -XX:TieredStopAtLevel=1**
  Остановиться на C1

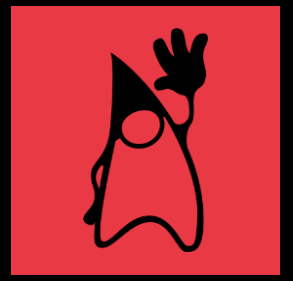
Warmup aka прогрев





Синт. нагрузка

1. Прогрев



Algotrading Application

2. Работа

Биржа

Прогрев

очень легко ошибиться

TradingEngine.java

```
void sendOrderToExchange(Order order) {
    if (MySystem.isWarmUp()) {
        ignore(order);
    } else {
        sendRealOrder(order);
    }
}
```

Прогрев

очень легко ошибиться

TradingEngine.java

```
void sendOrderToExchange(Order order) {
    if (MySystem.isWarmUp()) {
        ignore(order);
    } else {
        sendRealOrder(order);
    }
}
```

100%

случаев во
время прогрева

Прогрев

очень легко ошибиться

TradingEngine.java

```
void sendOrderToExchange(Order order) {
    if (MySystem.isWarmUp()) {
        ignore(order);
    } else {
        sendRealOrder(order);
    }
}
```

Прогрев

очень легко ошибиться

TradingEngine.java

```
void sendOrderToExchange(Order order) {
    if (MySystem.isWarmUp()) {
        ignore(order);
    } else {
        uncommonTrap();
    }
}
```

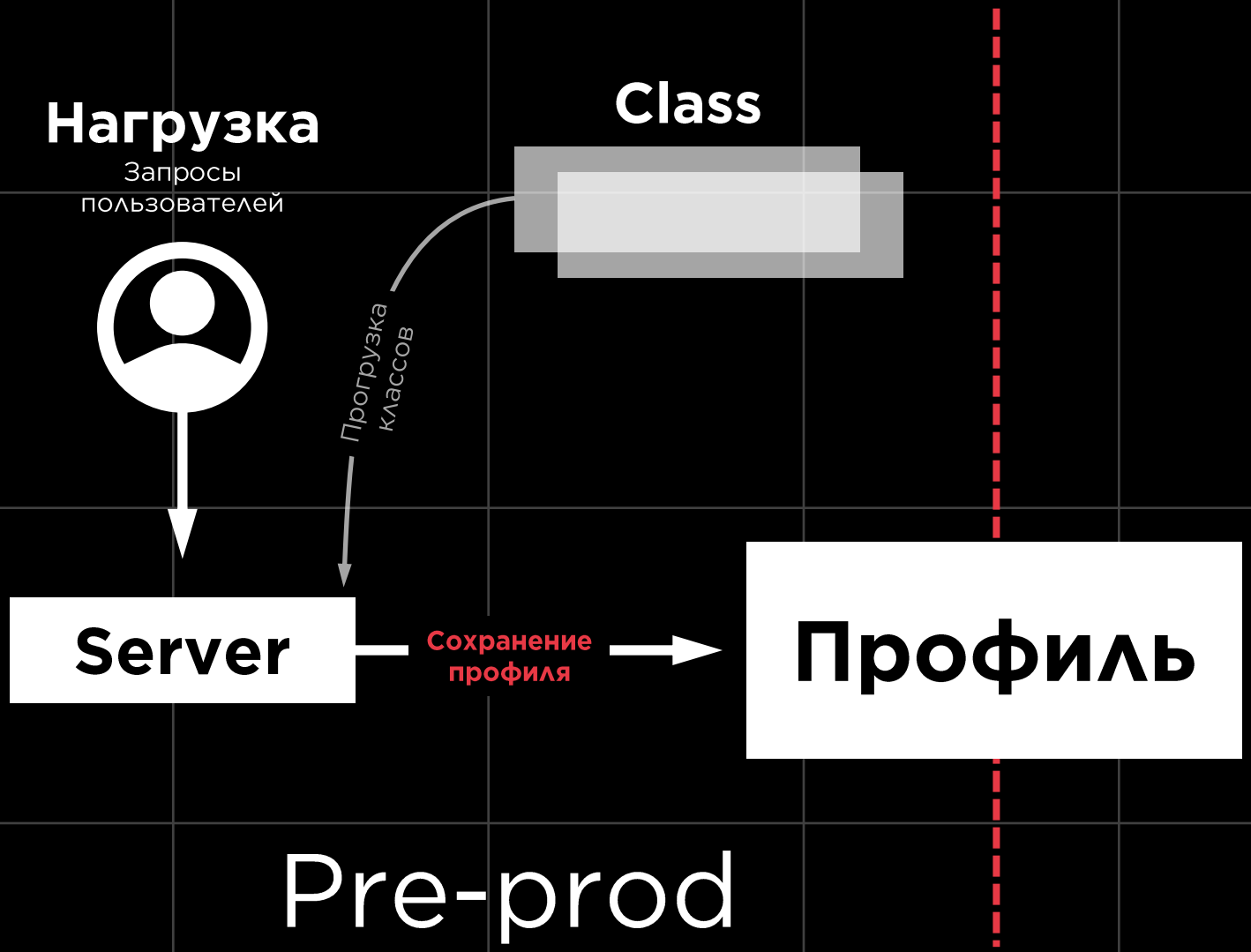
Откат
в интерпретатор

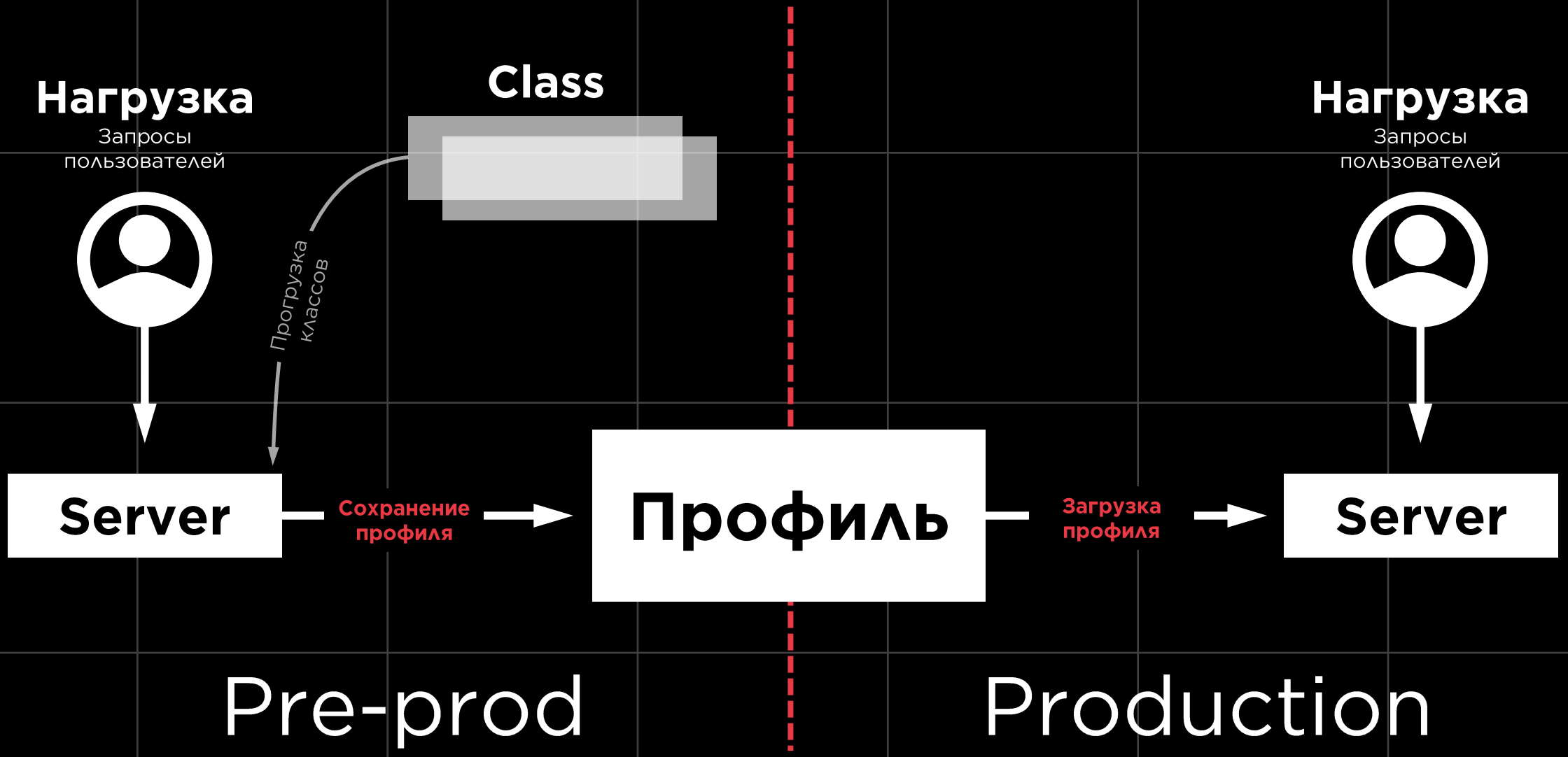
Alibaba Dragonwell

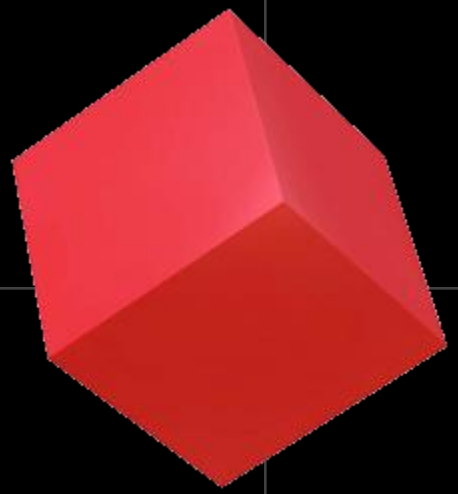
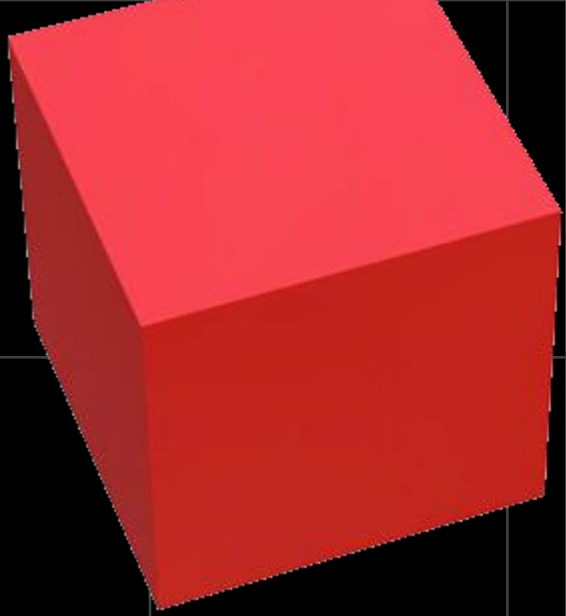
JWarmup

azul

ReadyNow


Нагрузка
Запросы пользователей
Class
Прогрузка классов
Server
Сохранение профиля
Профиль
Pre-prod


Нагрузка
Запросы пользователей
Class
Прогрузка классов
Нагрузка
Запросы пользователей
Server
Сохранение профиля
Профиль
Загрузка профиля
Server
Pre-prod
Production

Coordinated Restore at Checkpoint

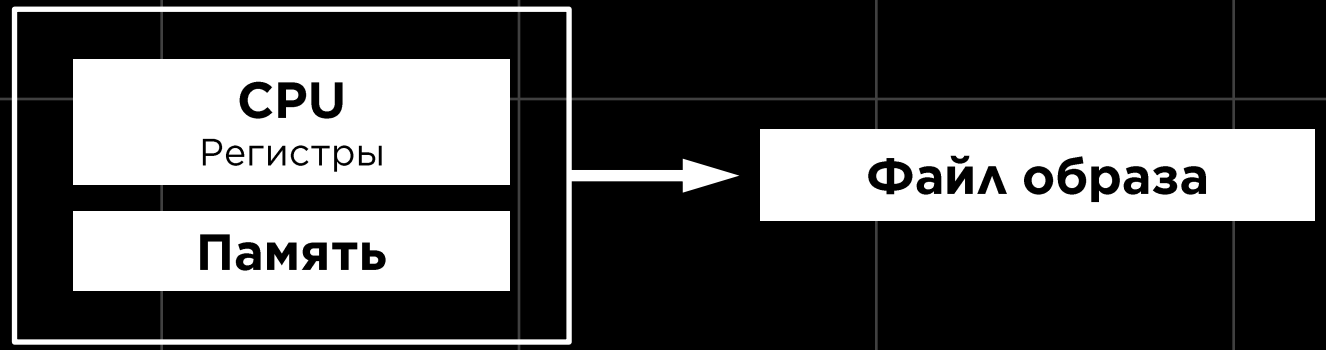
Checkpoint/Restore

In Userspace

Проект Linux:

Основная идея:

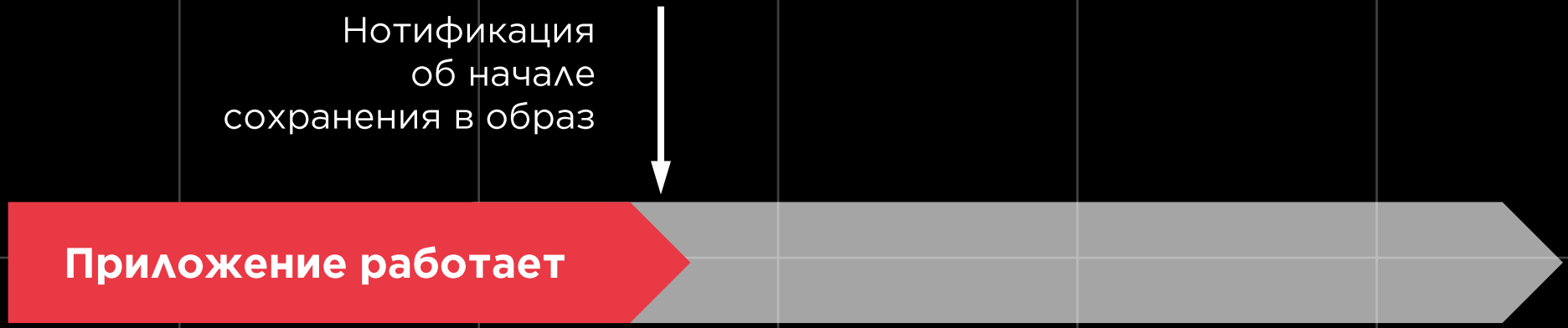
- **Заморозить запущенное приложение**
- **Создать моментальный снимок состояния приложений (в виде набора файлов)**
- **Позже использовать эти файлы для перезапуска приложения с того же места**
  - Потенциально, на другой/других физических машинах



CRaC

Co-ordinated Restore at Checkpoint

У приложения должна быть возможность реагировать на процедуру сохранения образа и загрузку из образа

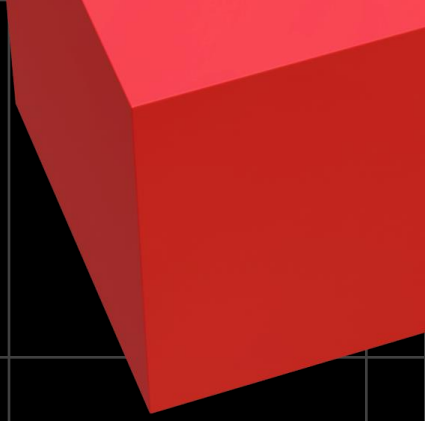


CRaC

Co-ordinated Restore at Checkpoint

У приложения должна быть возможность реагировать на процедуру сохранения образа и загрузку из образа

CRaC



- Снимаем снапшот с уже прогретой JVM
- Восстанавливаемся с полным состоянием, полностью загруженными классами, JIT-скомпилированным кодом
- Но что с ресурсами?

```
class MyServer implements Resource {
        Server server;




}
```

```
class MyServer implements Resource {
    Server server;

    @Override
    public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
        server.stop();
    }

}
```

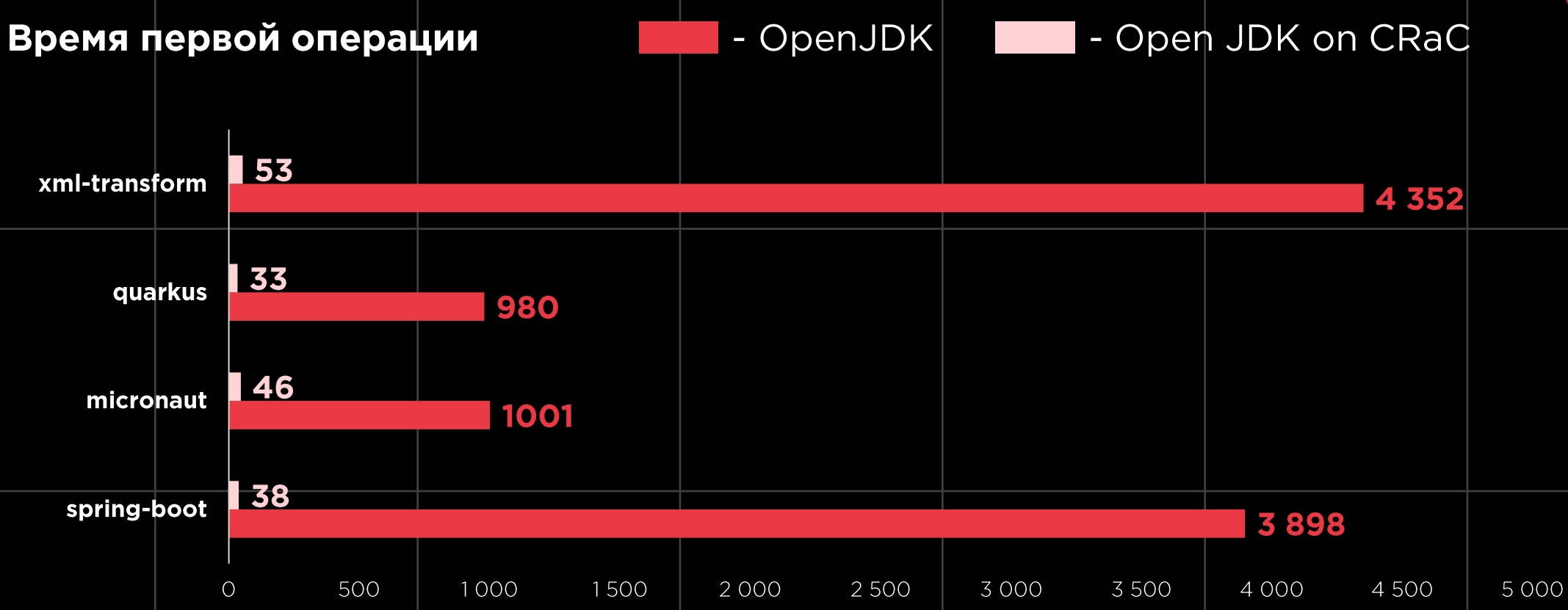
```
class MyServer implements Resource {
    Server server;

    @Override
    public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
        server.stop();
    }

    @Override
    public void afterRestore(Context<? extends Resource> context) throws Exception {
        server.start();
    }



}
```

```
class MyServer implements Resource {
        Server server;

        @Override
        public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
            server.stop();
        }

        @Override
        public void afterRestore(Context<? extends Resource> context) throws Exception {
            server.start();
        }

        public MyServer(int port, Handler handler) throws Exception {
            ...
            Core.getGlobalContext().register(this);
        }
}
```

```
class MyServer implements Resource {
        Server server;

        @Override
        public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
            server.stop();
        }

        @Override
        public void afterRestore(Context<? extends Resource> context) throws Exception {
            server.start();
        }

        public MyServer(int port, Handler handler) throws Exception {
            ...
            Core.getGlobalContext().register(this);
        }
}
```

Результаты

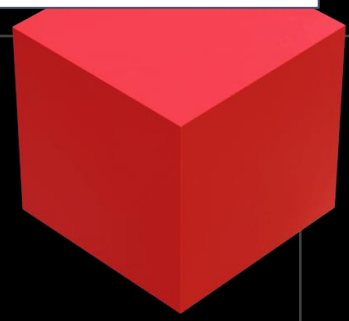
github.com/CRaC/docs



Clojure Repl: быстрый старт*

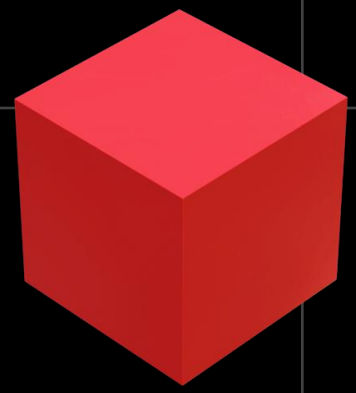
Clojure's slow start — what's inside?

```
~$ time java -cp "clj/*" clojure.main -e '(System/exit 0)'

real    0m0,236s
user    0m0,484s
sys     0m0,024s
```



Clojure Repl: быстрый старт*

```
~$ java -cp "clj/*" -XX:CRaCCheckpointTo=repl-checkpoint
clojure.main  -e '(jdk.crac.Core/checkpointRestore)'
```



* yizhepku.github.io/clojure-crac

Clojure Repl: быстрый старт*

```
~$ java -cp "clj/*" -XX:CRaCCheckpointTo=repl-checkpoint
clojure.main  -e '(jdk.crac.Core/checkpointRestore)'
```

```
~$ time java -cp "clj/*" -XX:CRaCRestoreFrom=repl-checkpoint
clojure.main -e '(System/exit 0)'

real    0m0,016s
user    0m0,017s
sys     0m0,009s
```

* yizhepku.github.io/clojure-crac

Clojure Repl: быстрый старт*

```
~$ java -cp "clj/*" -XX:CRaCCheckpointTo=repl-checkpoint
clojure.main  -e '(jdk.crac.Core/checkpointRestore)'
```

```
~$ time java -cp "clj/*" -XX:CRaCRestoreFrom=repl-checkpoint
clojure.main -e '(System/exit 0)'

real    0m0,016s
user    0m0,017s
sys     0m0,009s
```

Было:

```
real    0m0,236s
user    0m0,484s
sys     0m0,024s
```

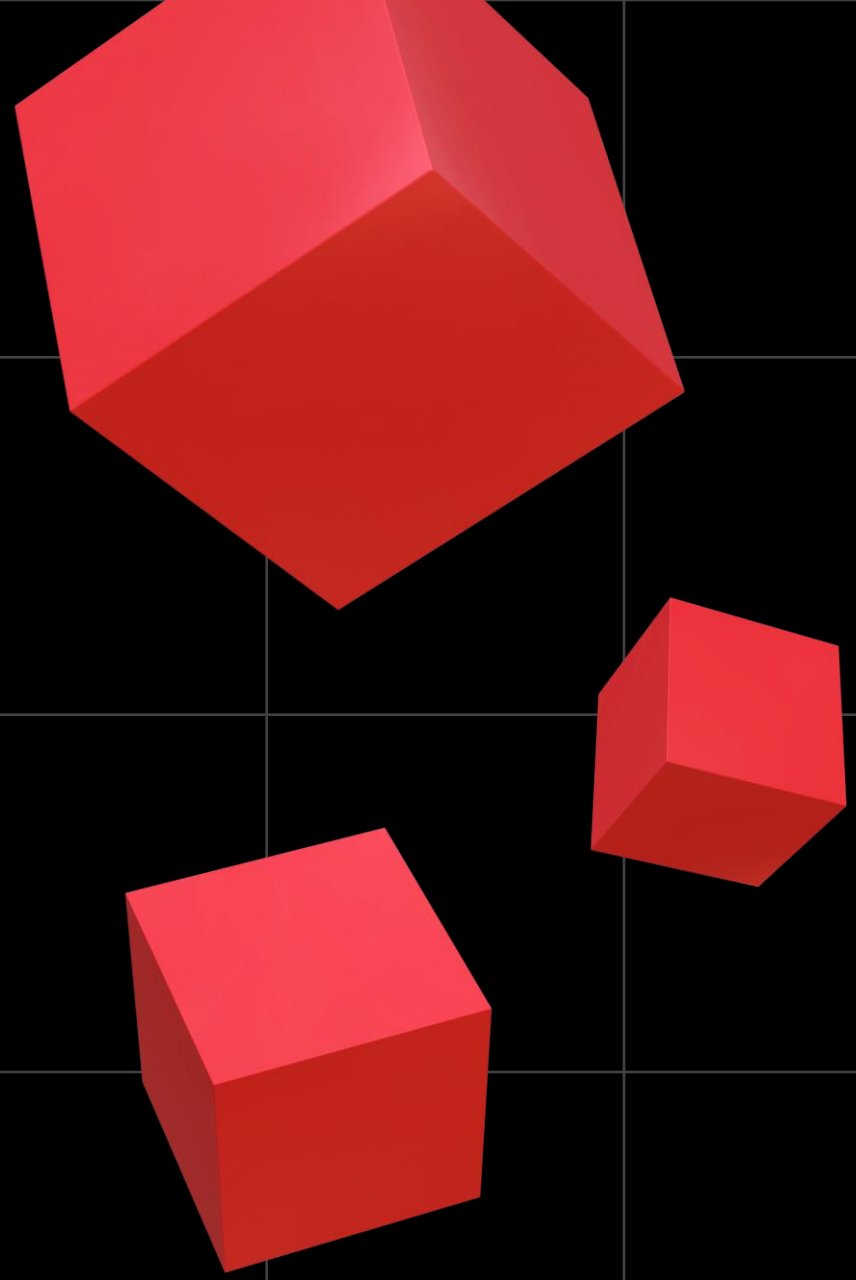
* yizhepku.github.io/clojure-crac

CRaC:

Схема использования

Итоги

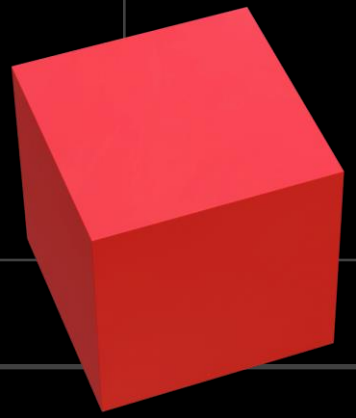
- Флагов много, но нужно читать код,
  а не ответы 5+ летней давности
- Ручной прогрев - дело хрупкое
  + риск деоптимизаций
- Переиспользование jit-профиля
  с прошлого раза - экзотические JVM
- Интероп с C++ не такая уж плохая идея
- Crac: F5/F9 в мире JVM,
  но без координации не обойтись
- Полностью готовых решений нет

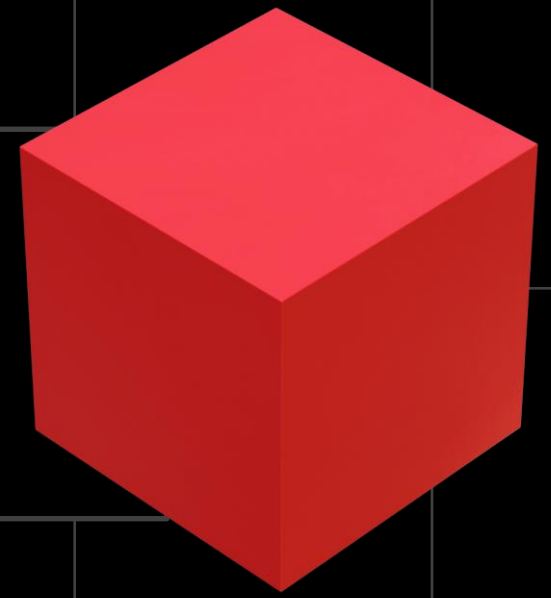


Прогреем

вместе!

2023





Joker<?> | МИР Plat.Form | AXIOM JDK