А можно погорячее?

Чем и как мы прогреваем Spring микросервисы

Балдин Сергей

Кто Говорит?

11 лет в IT

8 лет Java/Kotlin Backend Dev

1.5 года в Команде Надежности в TBank

Для кого этот доклад?

Я разработчик сервиса, который:

- Работает окружении с ограниченными ресурсами
- Должен иметь стабильную задержку на старте
- Должен держать высокую нагрузку на старте

О Чем Поговорим?

Теплые и холодные JVM

Расскажем о проблеме и углубимся в теорию

01

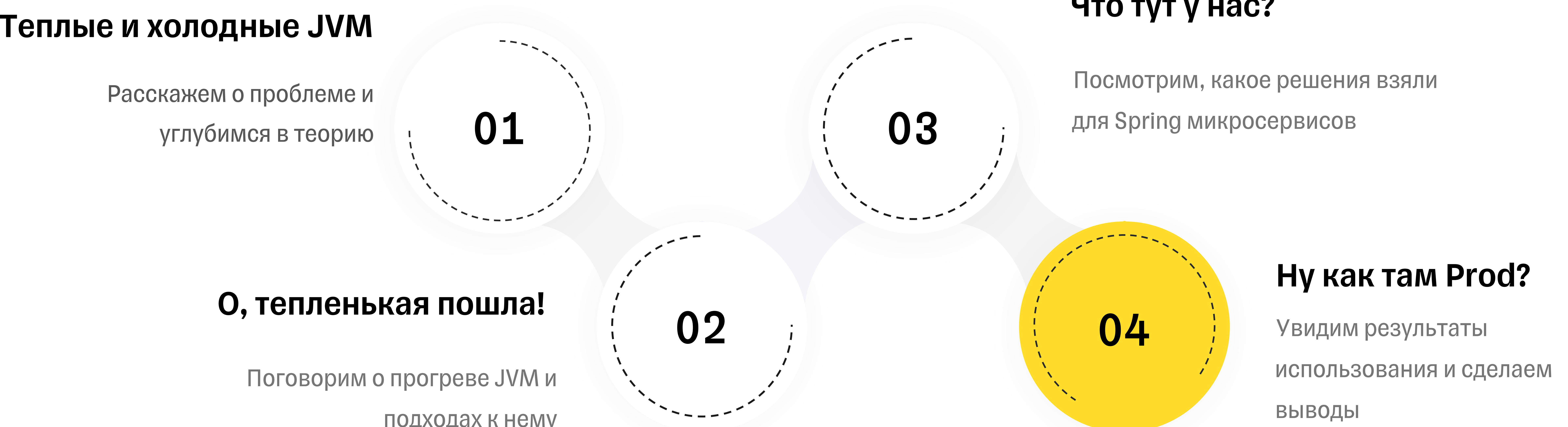
О Чем Поговорим?

О Чем Поговорим?

Что тут у нас?

Посмотрим на то к какому решение мы пришли

О Чем Поговорим?



Теплые и Холодные JVM

Проблематика

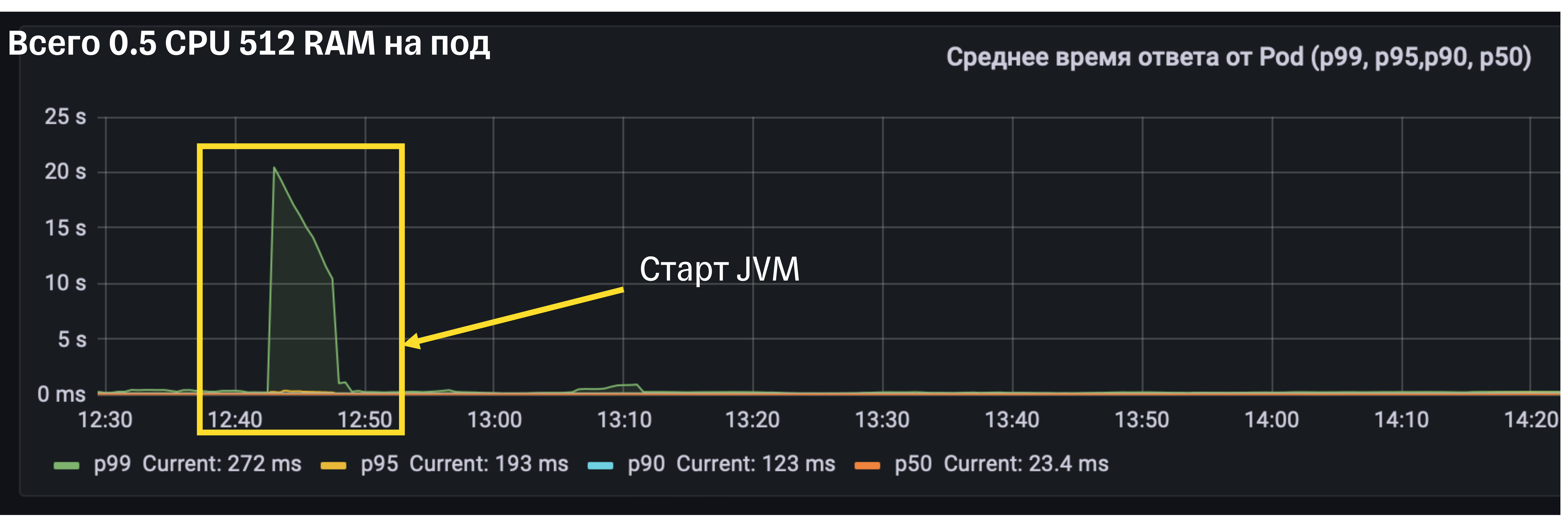
Проблематика

Ресурсы всего кластера не могут расти бесконечно

Сервисы имеют жесткие ограничения на потребляемые ресурсы

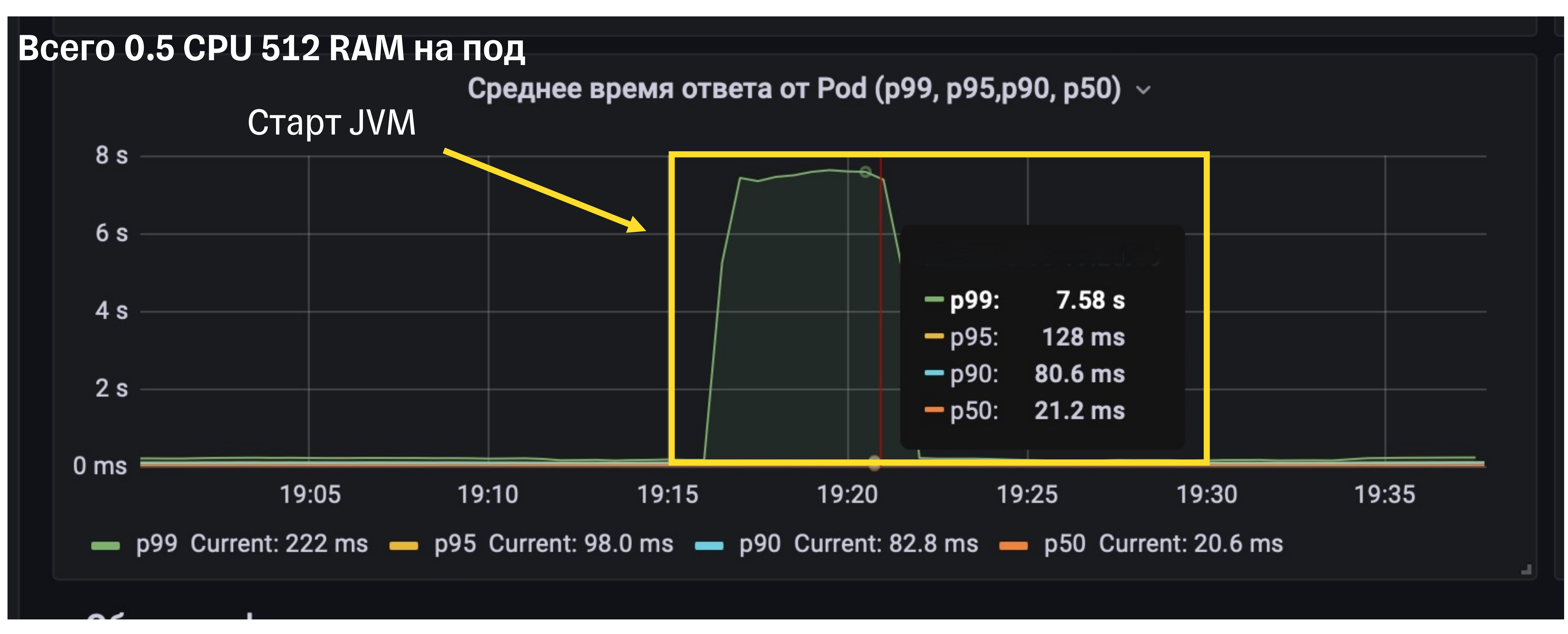
Проблематика

Негативных эффект жестких ограничений на графиках времени ответа:



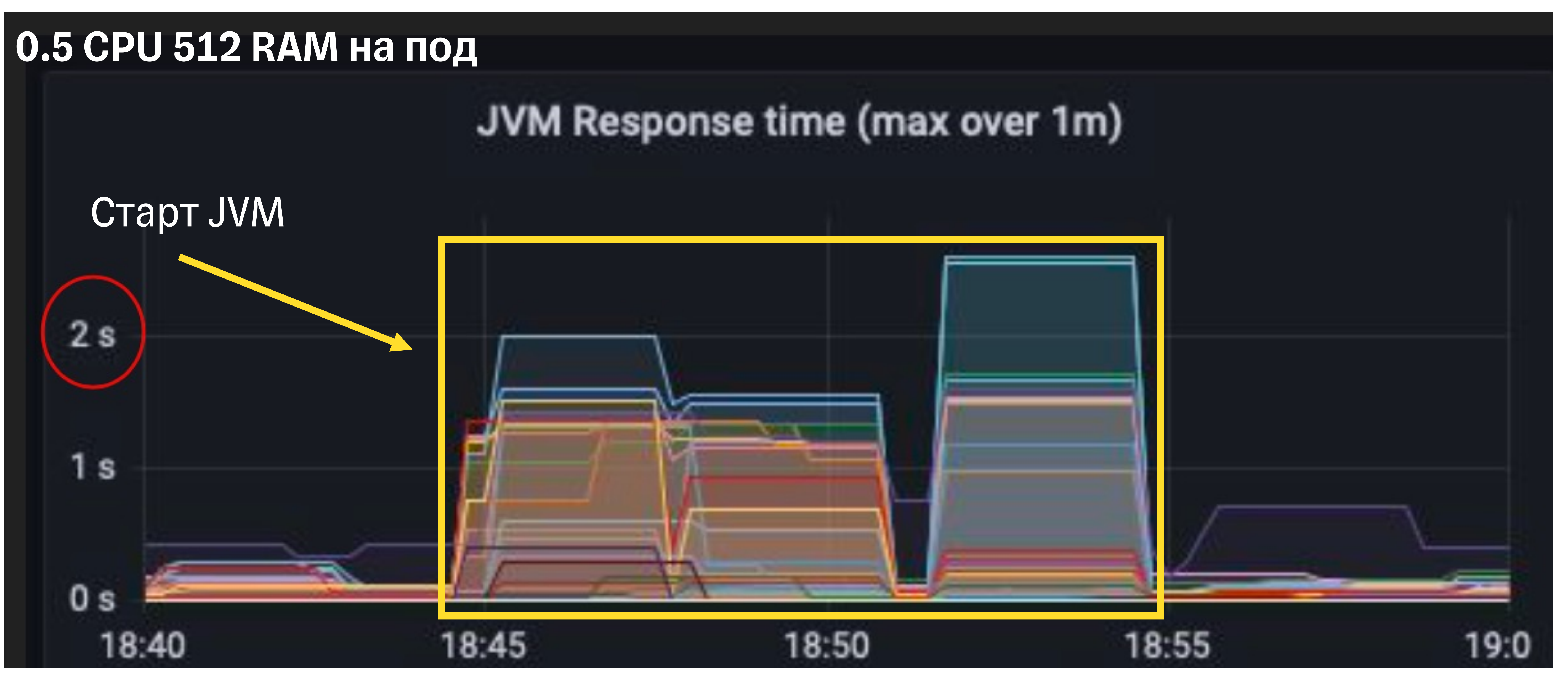
Проблематика

Хорошо видны пики задержки после старта приложения:



Проблематика

Даже небольшие пики задержки в 2с выше среднего времени ответа:



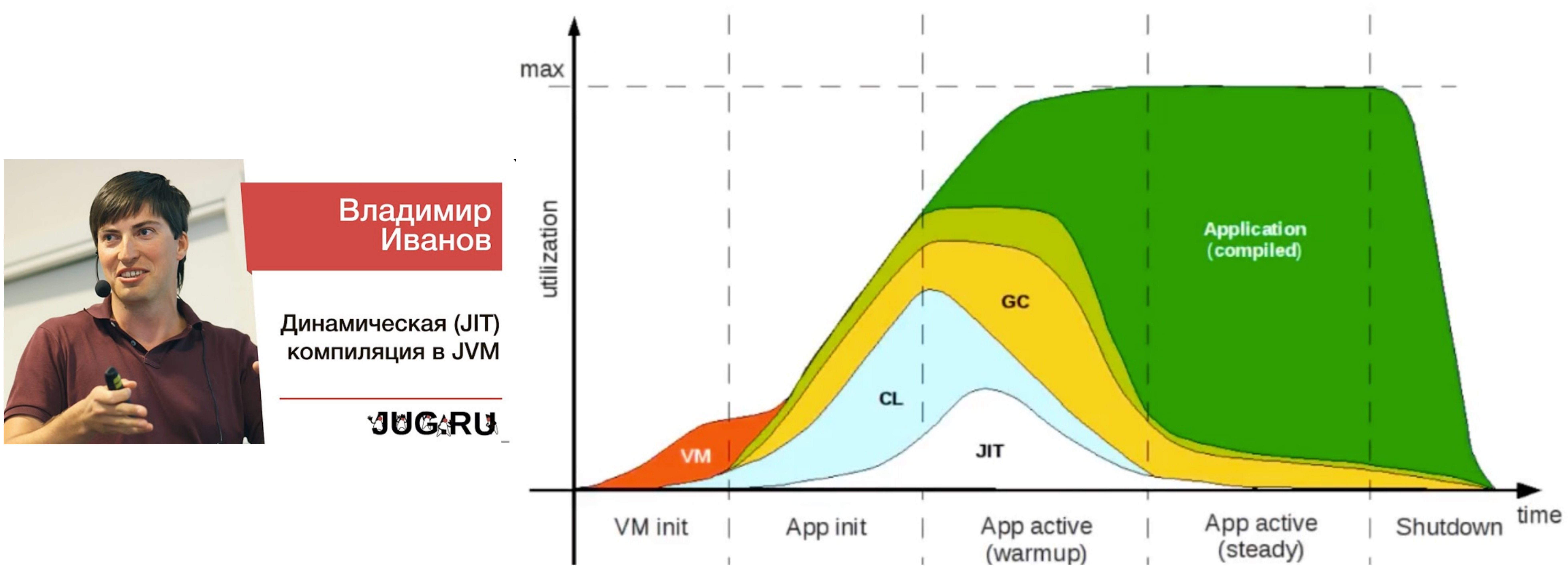
Проблематика

За эти 20/8/2 секунды мы можем потерять клиентов и деньги компании!

Теплые и Холодные JVM

Немного истории

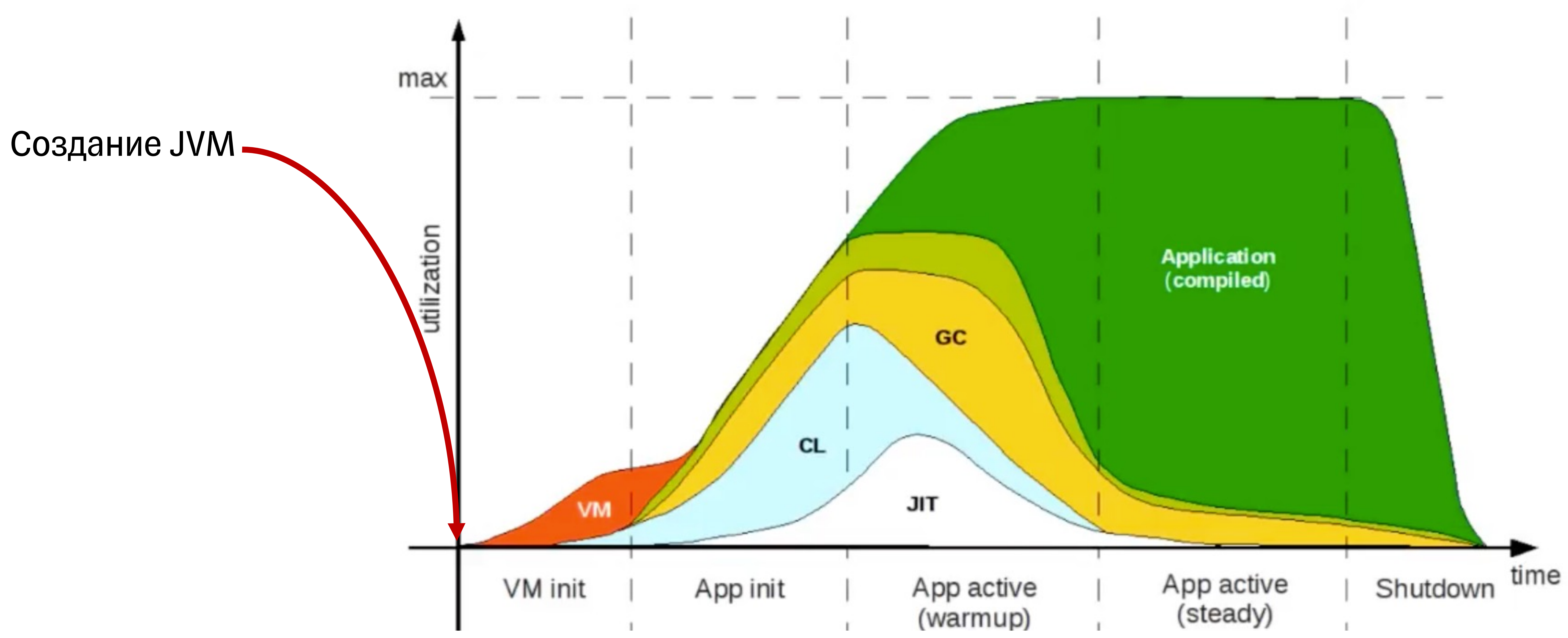
Жизненный цикл приложения JVM



https://cr.openjdk.org/~vlivanov/talks/2015_JIT_Overview.pdf

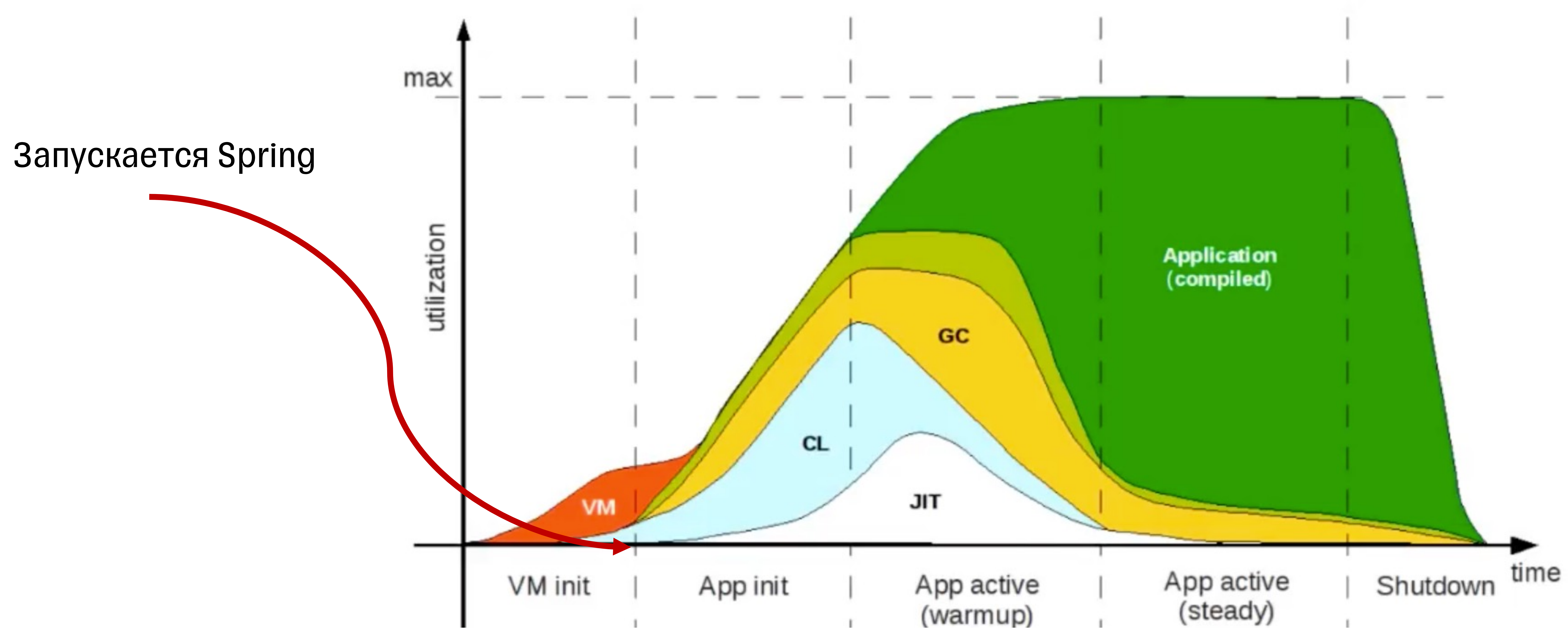
Жизненный цикл приложения JVM

Создание JVM



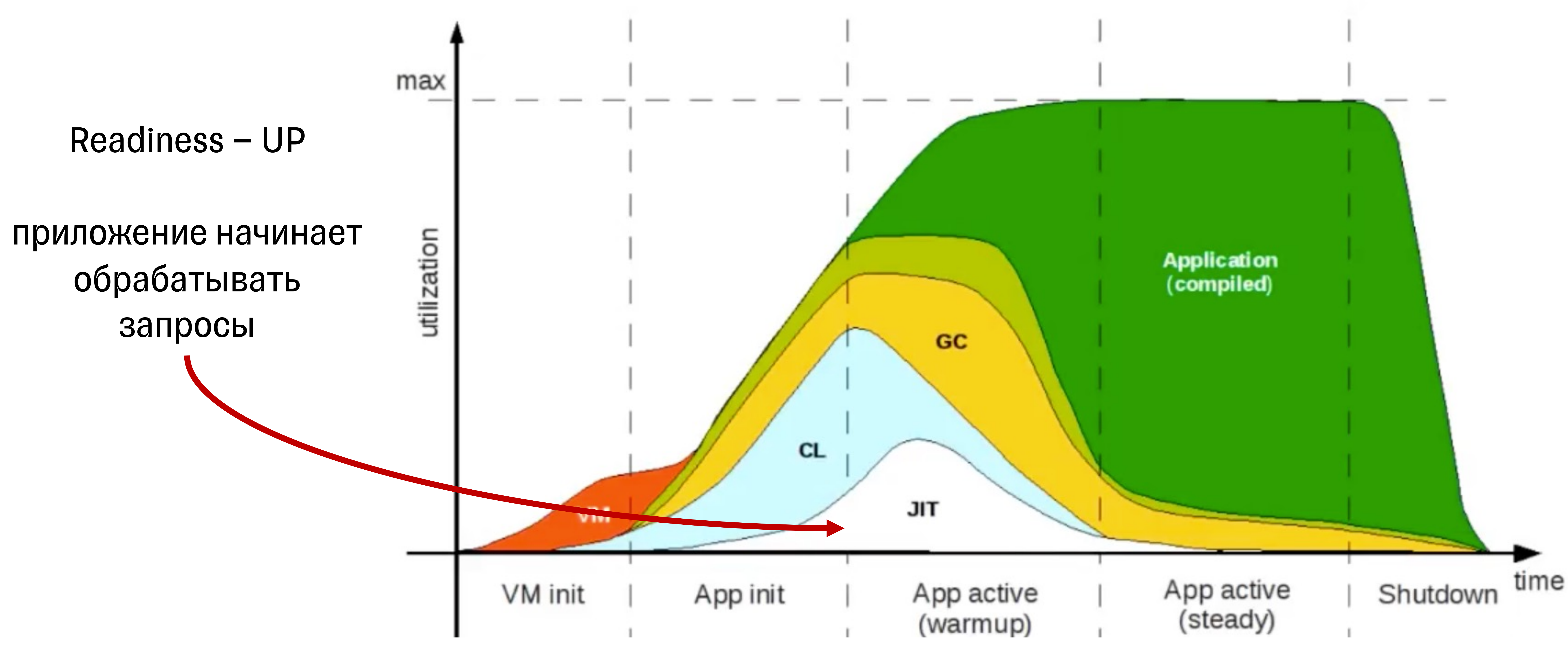
https://cr.openjdk.org/~vlivanov/talks/2015_JIT_Overview.pdf

Жизненный цикл приложения JVM



https://cr.openjdk.org/~vlivanov/talks/2015_JIT_Overview.pdf

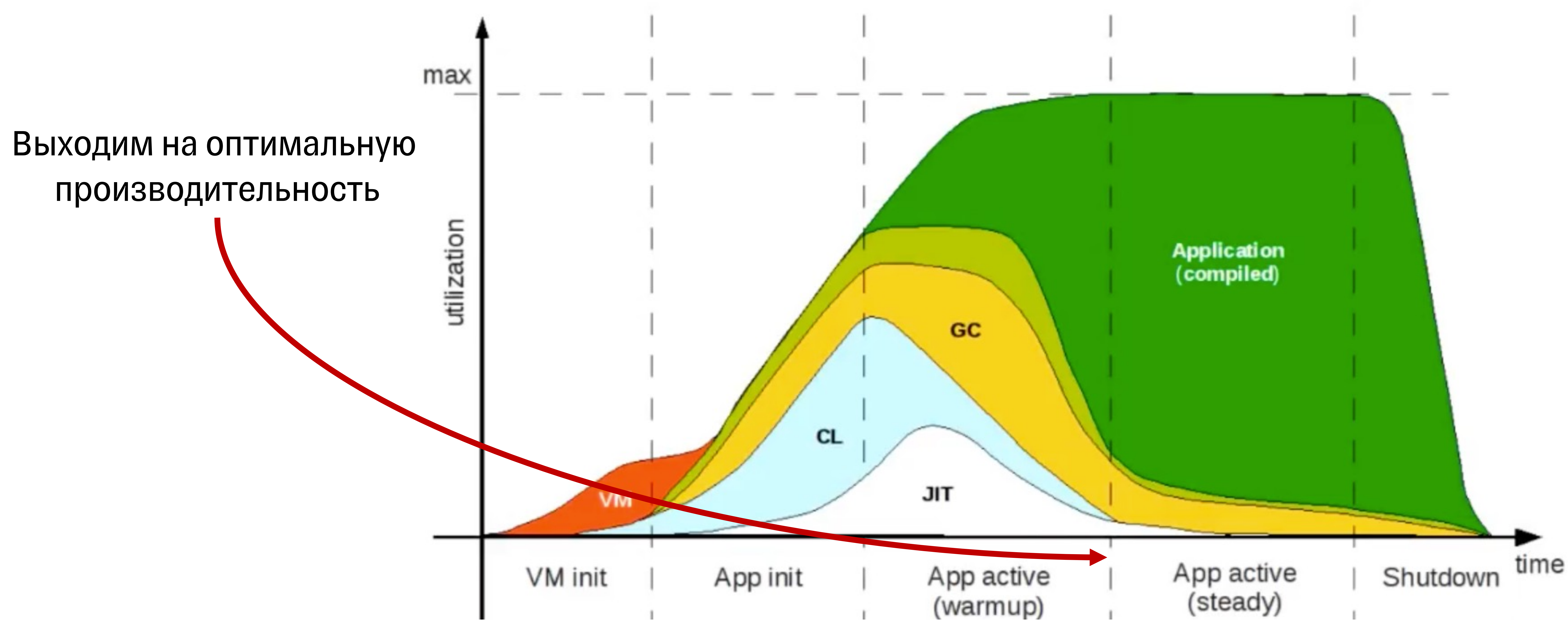
Жизненный цикл приложения JVM



https://cr.openjdk.org/~vlivanov/talks/2015_JIT_Overview.pdf

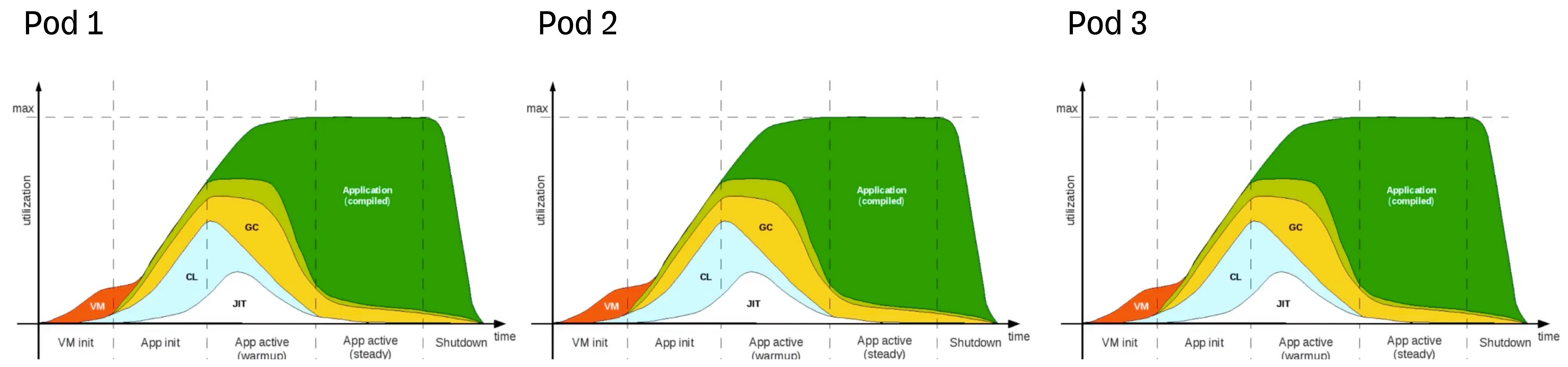
Жизненный цикл приложения JVM

Выходим на оптимальную производительность



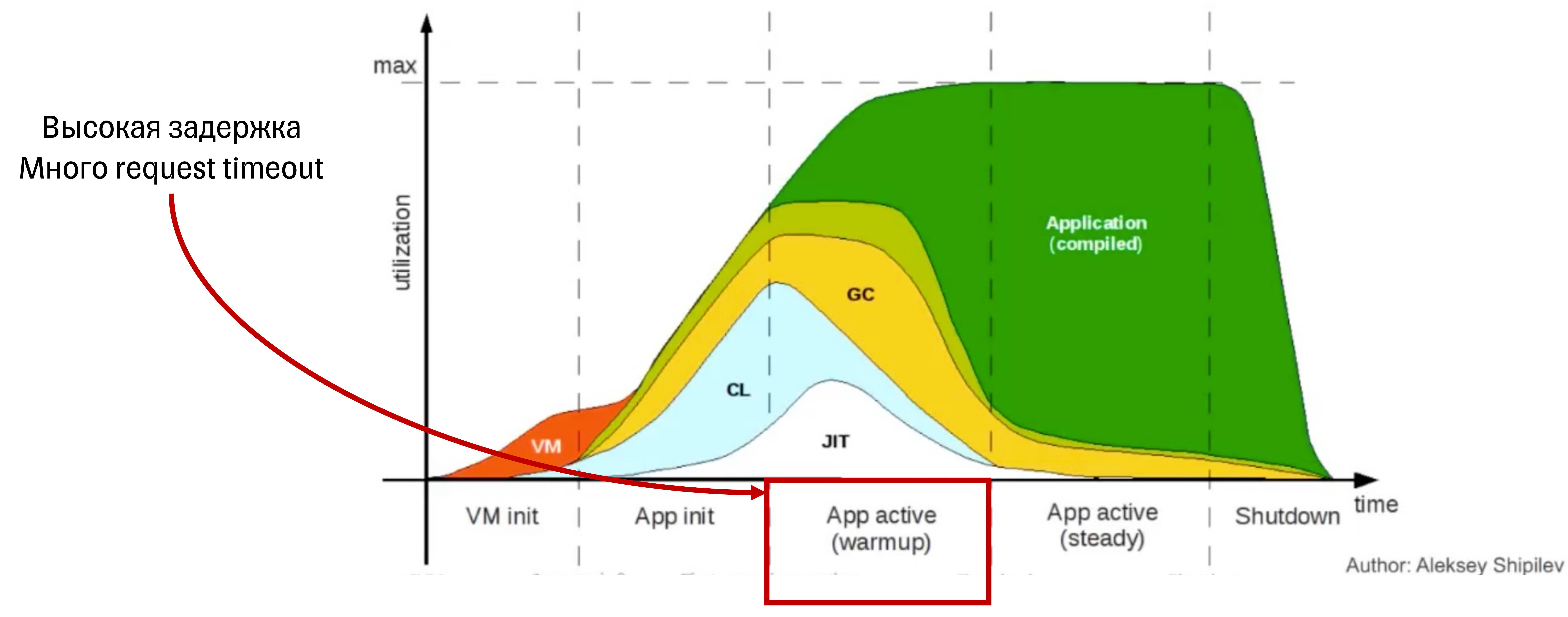
https://cr.openjdk.org/~vlivanov/talks/2015_JIT_Overview.pdf

Жизненный цикл приложения JVM



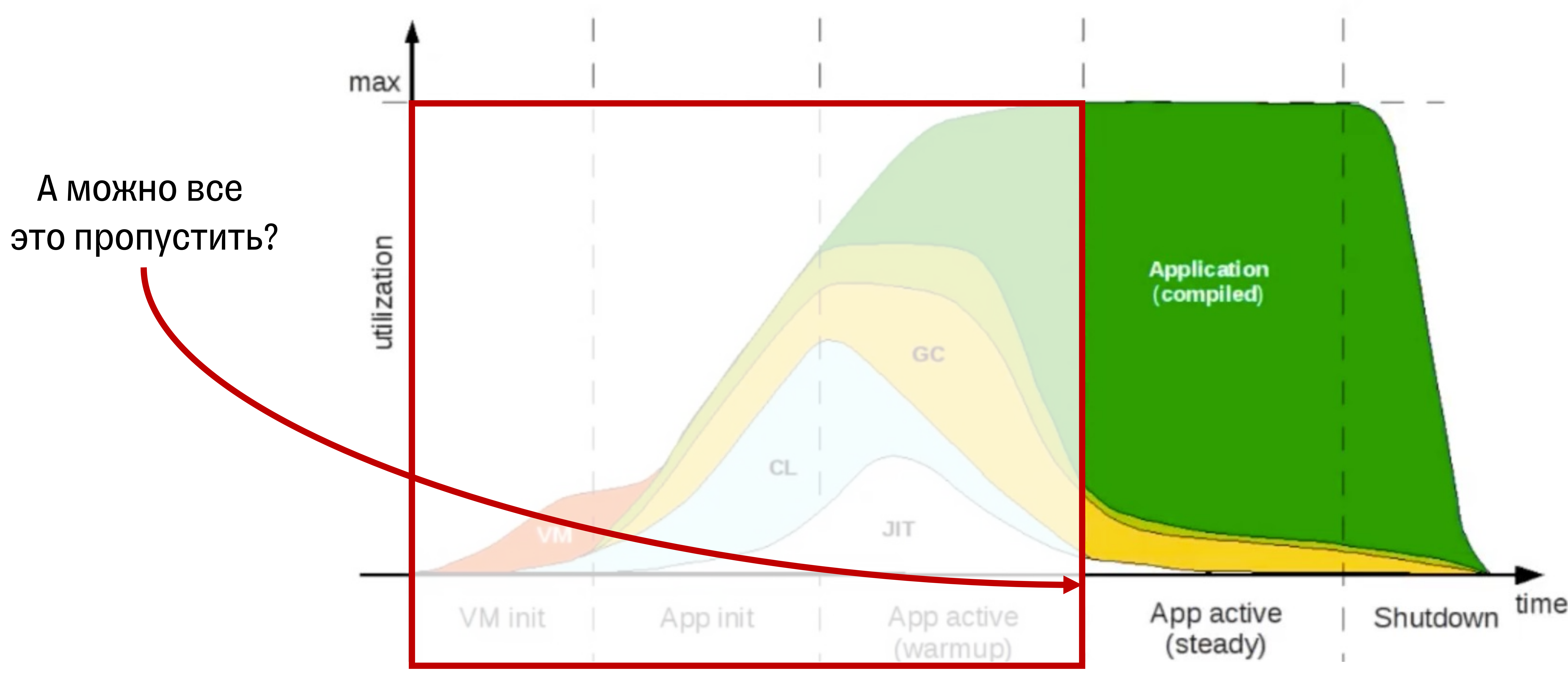
https://cr.openjdk.org/~vlivanov/talks/2015_JIT_Overview.pdf

Жизненный цикл приложения JVM



https://cr.openjdk.org/~vlivanov/talks/2015_JIT_Overview.pdf

Жизненный цикл приложения JVM



https://cr.openjdk.org/~vlivanov/talks/2015_JIT_Overview.pdf

Многоуровневая Компиляция

Сборка Мусора и Загрузка Клаасов вне скоупа доклада

Сосредоточимся на работе JIT компилятора

- Level 0 – Interpreted Code
- Level 1 – Simple C1 Compiled Code
- Level 2 – Limited C1 Compiled Code
- Level 3 – Full C1 Compiled Code
- Level 4 – C2 Compiled Code

Многоуровневая Компиляция

Позволяет использовать C1 и C2 для достижения как быстрого запуска, так и хорошей долгосрочной производительности

- Level 0 – Interpreted Code
- Level 1 – Simple C1 Compiled Code
- Level 2 – Limited C1 Compiled Code
- Level 3 – Full C1 Compiled Code
- Level 4 – C2 Compiled Code

Многоуровневая Компиляция

В многоуровневой компиляции много нюансов!

Уровень	Скорость
Level 0 – Interpreted Code	Медленно
Level 1 – Simple C1 Compiled Code	Побыстрее
Level 2 – Limited C1 Compiled Code	Средне
Level 3 – Full C1 Compiled Code	Медленно
Level 4 – C2 Compiled Code	Быстро

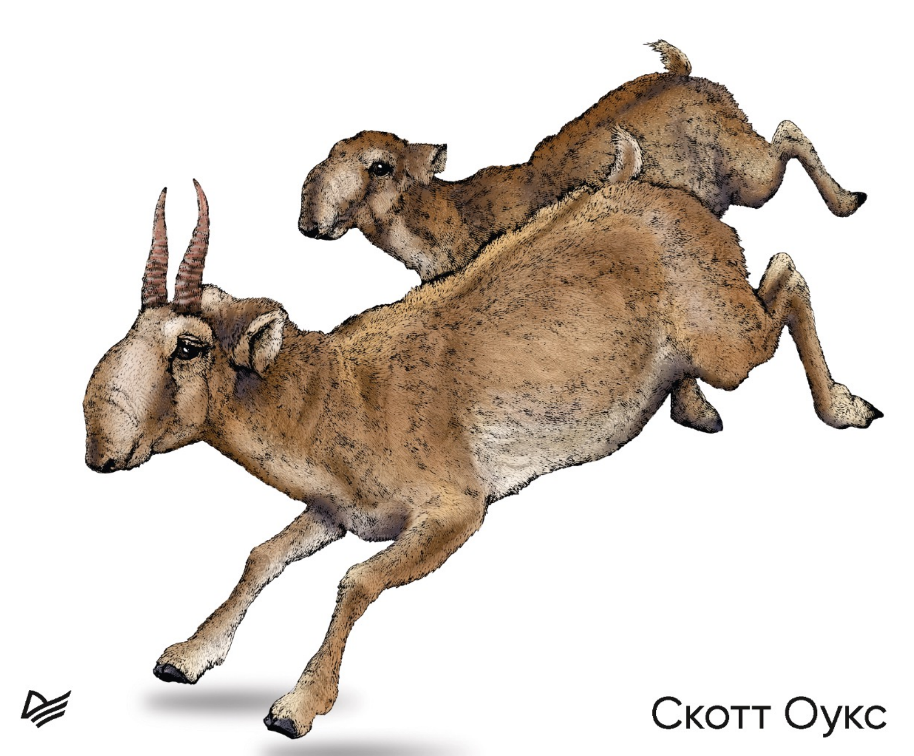
Код скомпилированный на уровне 3 будет медленнее чем на уровне 1 или 2!

Компиляция работает не по порядку!

Многоуровневая Компиляция

Более подробнее читаем тут:

- Level 0 – Interpreted Code
- Level 1 – Simple C1 Compiled Code
- Level 2 – Limited C1 Compiled Code
- Level 3 – Full C1 Compiled Code
- Level 4 – C2 Compiled Code



Многоуровневая Компиляция

Мы хотим, чтобы горячие точки наших приложений были скомпилированы на Level 4 - C2

- Level 0 – Interpreted Code
- Level 1 – Simple C1 Compiled Code
- Level 2 – Limited C1 Compiled Code
- Level 3 – Full C1 Compiled Code
- **Level 4 – C2 Compiled Code**

Многоуровневая Компиляция

Порог компиляции — это количество вызовов метода до того, как код будет скомпилирован с применением конкретного компилятора и оптимизаций

```
~/workspace/education/warmup/demo-spring | on main +81 !13 ?23  java -XX:+PrintFlagsFinal -version | grep -E 'Tier.*CompileThreshold'
    uintx IncreaseFirstTierCompileThresholdAt      = 50                                        {product} {default}
     intx Tier2CompileThreshold                    = 0                                         {product} {default}
     intx Tier3CompileThreshold                    = 2000                                      {product} {default}
     intx Tier4CompileThreshold                    = 15000                                     {product} {default}
openjdk version "21" 2023-09-19
OpenJDK Runtime Environment (build 21+35-2513)
OpenJDK 64-Bit Server VM (build 21+35-2513, mixed mode, sharing)
```

О, тепленькая пошла!

Окружение

Окружение

Наши сервисы развернута в следующем окружении:

Кластер K8s

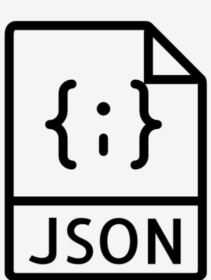
Istio Service Mesh

OpenJDK 17 - 21

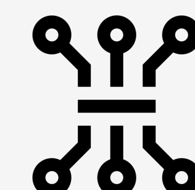
Что под капотом?

Какой стек мы используем:

Spring Boot



Jackson



Http Clients &
Connection Pools

О, тепленькая пошла!

Методы прогрева

Методы прогрева

Использование возможностей инфраструктуры

Плавный ввод приложения в эксплуатацию может снизить пики latency на старте, для этого можем использовать:

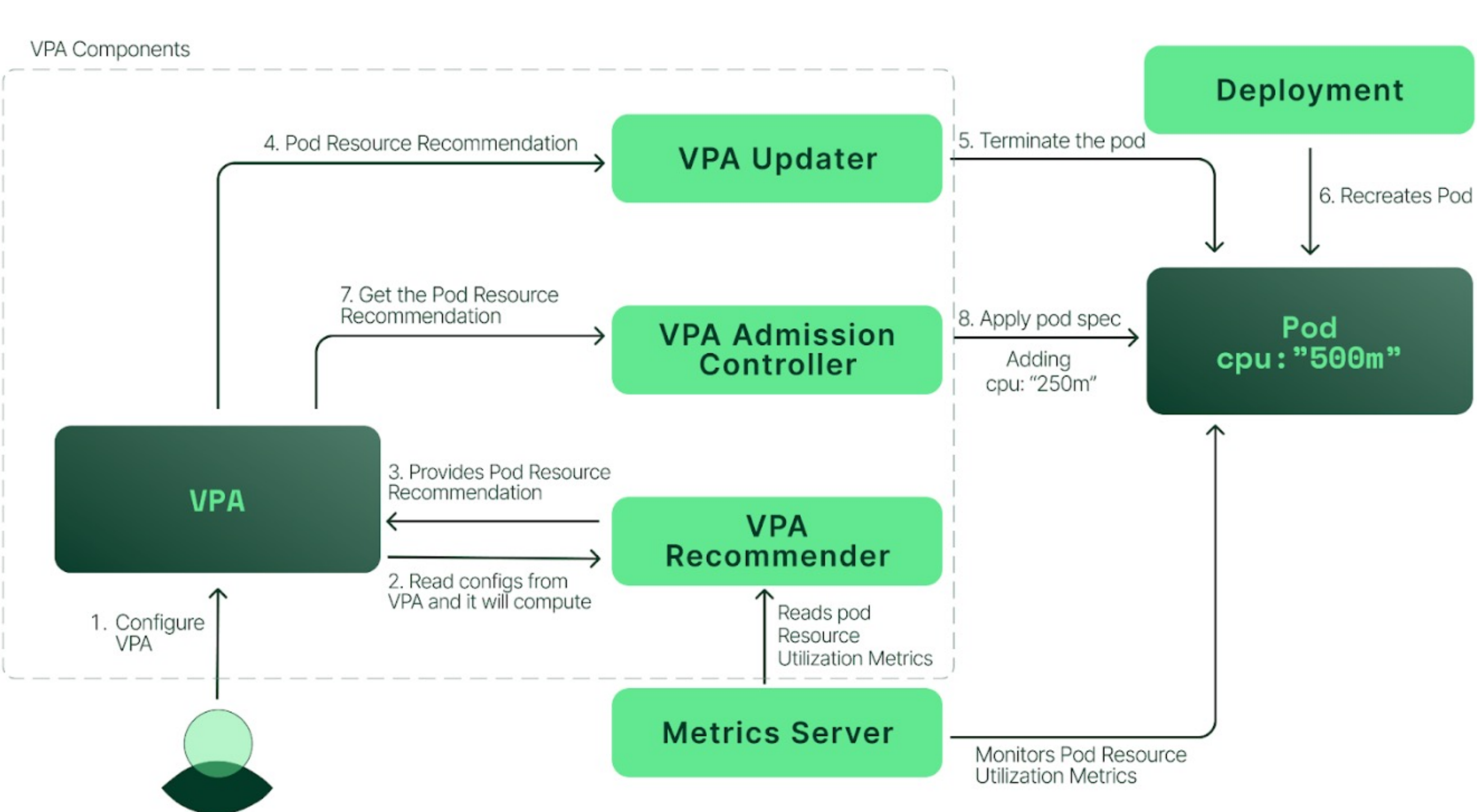
1. kubernetes VPA
2. Istio Traffic Management

Vertical Pod Autoscaler (VPA)

VPA описывает механизм оптимального определения требований к CPU и RAM

Нам интересна отдельная фича

в рамках кластера k8s в TBank



How Kubernetes VPA allocates resources.

[How K8s allocates resources]

VPA Burst

После подключения VPA в отдельный namespace, сервисам в нем становится доступен burst limits

Burst limits позволяет использовать 4 CPU на короткое время

CPU не гарантированы

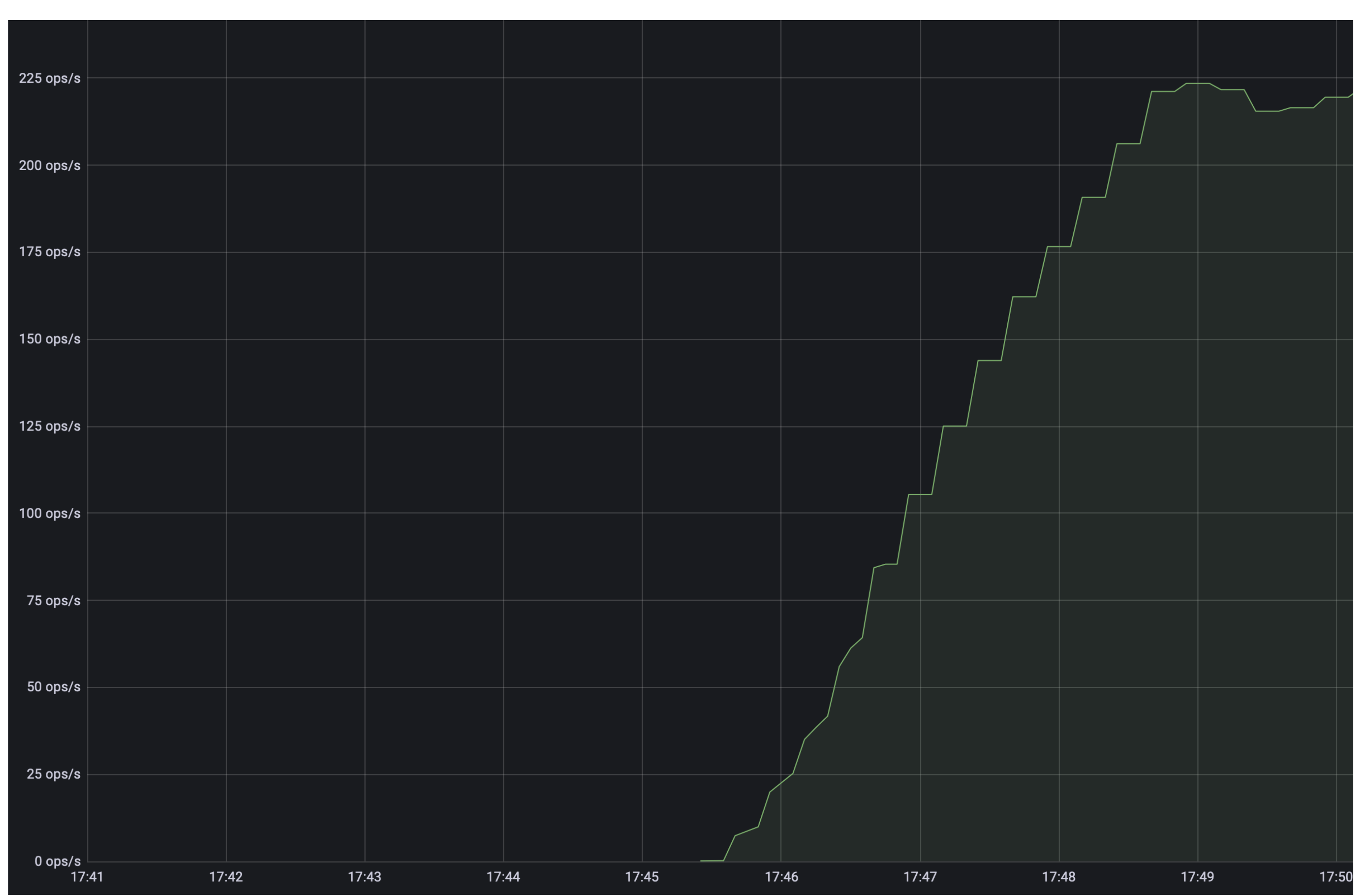
Kubernetes VPA

Istio WarmupDurationSecs

Istio предлагает широкие возможности для маршрутизации входящего трафика:

Параметр **WarmupDurationSecs:**

В течении N секунд Istio будет плавно увеличивать объем трафика на новый под



https://istio.io/latest/docs/reference/config/networking/destination-rule/#LoadBalancerSettings

Istio WarmupDurationSecs

Параметр **WarmupDurationSecs:**

В течении N секунд Istio будет плавно увеличивать объем трафика на новый под

Просто подключить

Слабо влияем на процесс

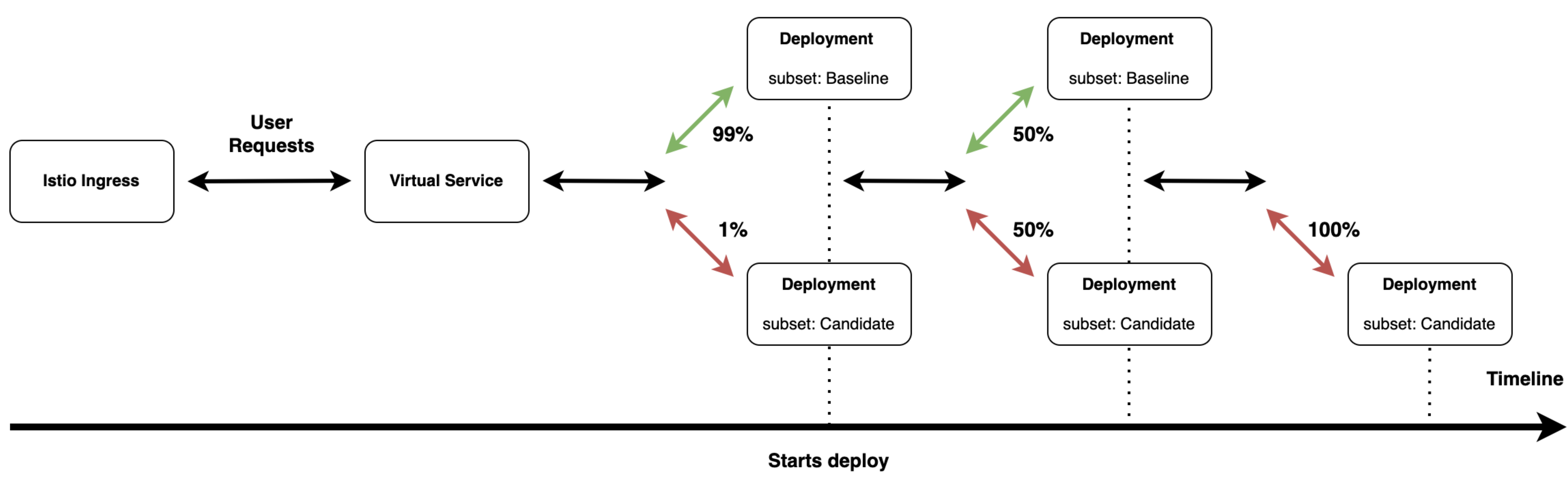
DestinationRule.yaml

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: data-enrichment-service-destination-rule
spec:
  host: data-enrichment-service
   trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
      warmupDurationSecs: 180s
```

https://istio.io/latest/docs/reference/config/networking/destination-rule/#LoadBalancerSettings

Istio Traffic Shifting

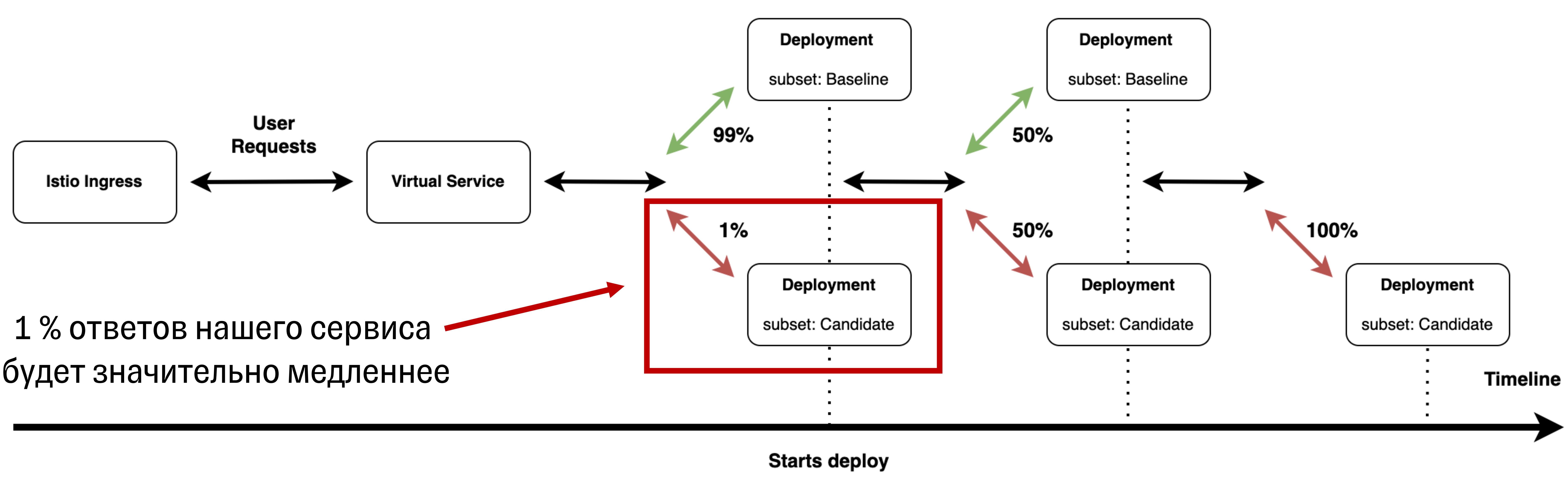
Механизм Traffic Shifting позволяют постепенно перемещать трафик между подмножествами deployment вашего сервиса



https://istio.io/latest/docs/tasks/traffic-management/traffic-shifting/

Istio Traffic Shifting

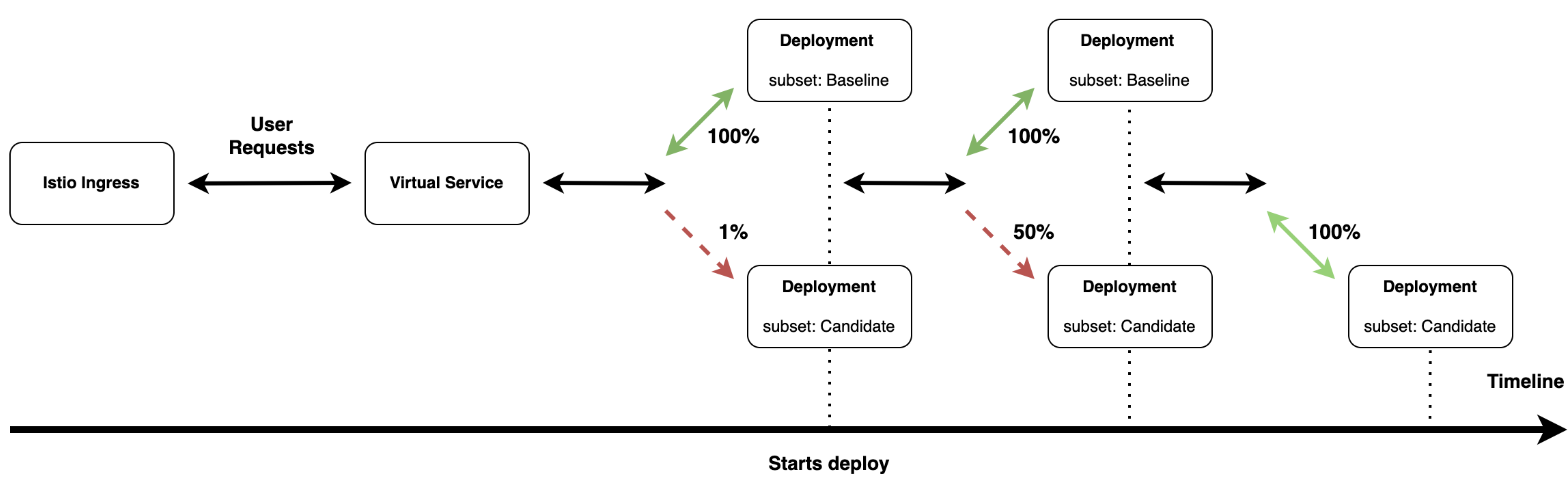
Механизм Traffic Shifting позволяют постепенно перемещать трафик между подмножествами deployment вашего сервиса



https://istio.io/latest/docs/tasks/traffic-management/traffic-shifting/

Istio Traffic Mirroring

Mirroring не влияет на время ответа сервиса, так как зеркалированный трафик не будет возвращен пользователям



https://istio.io/latest/docs/tasks/traffic-management/mirroring/

Istio Traffic Management

- Простое, но гибкое решение
- Нужно менять механизм деплоя
- Shifting оставляет небольшой процент медленного трафика
- Mirroring необходима модификация приложения для корректной работы

Методы прогрева

Использование возможностей инфраструктуры

Плавный ввод приложения в эксплуатацию:

1. kubernetes VPA
2. Istio Traffic Management

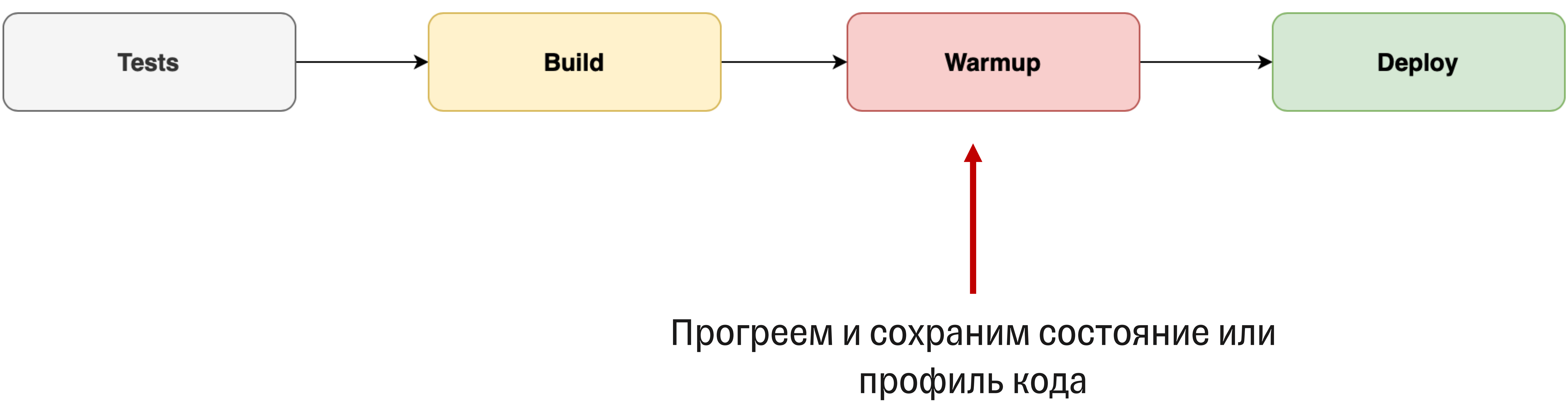
Подготовка приложения к эксплуатации

Рассмотрим перспективные решения, которые прогревают приложение до ввода его в эксплуатацию:

1. Azul Ready Now
2. Coordinated Restore at Checkpoint (CRaC)
3. Graal VM Native Image
4. Project Layden

Подготовка приложения к эксплуатации

Все перечисленные механизмы выносят прогрев на фазу сборки нашего приложения



Ready Now

Значительно улучшает поведения приложения на старте

https://docs.azul.com/prime/Use-ReadyNow

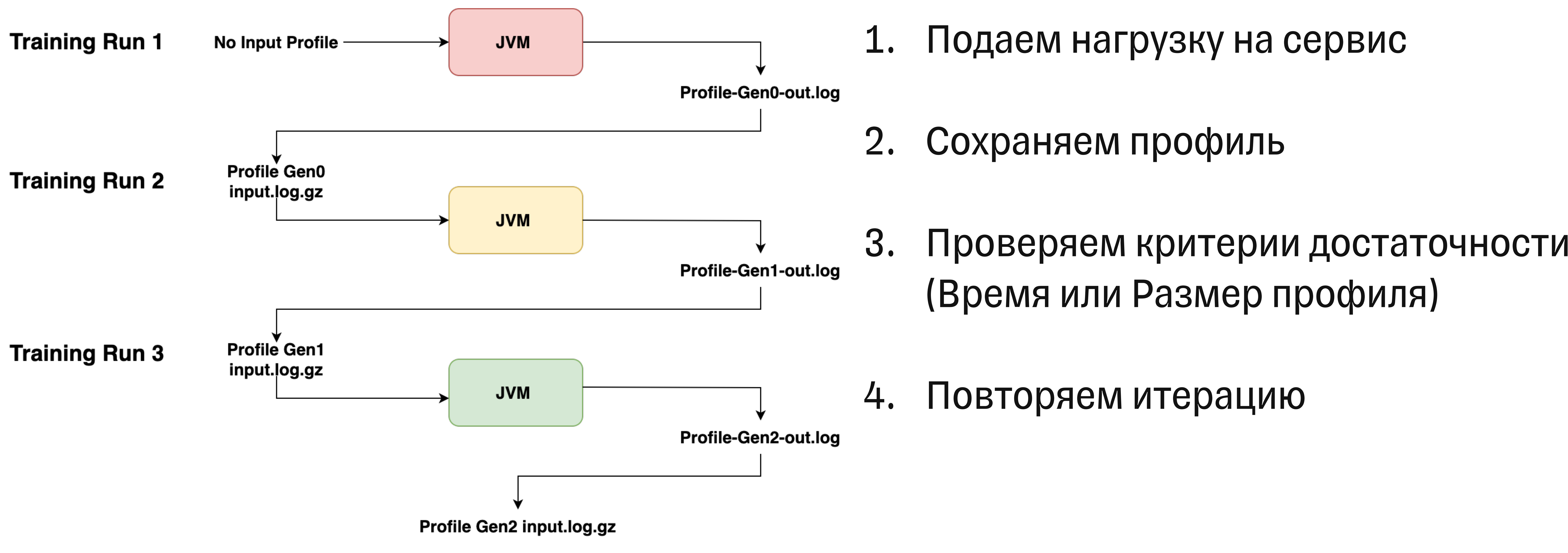
Ready Now

Значительно улучшает поведения приложения на старте

Сохраняет информацию профилирования и использует ее при следующих запусках

https://docs.azul.com/prime/Use-ReadyNow

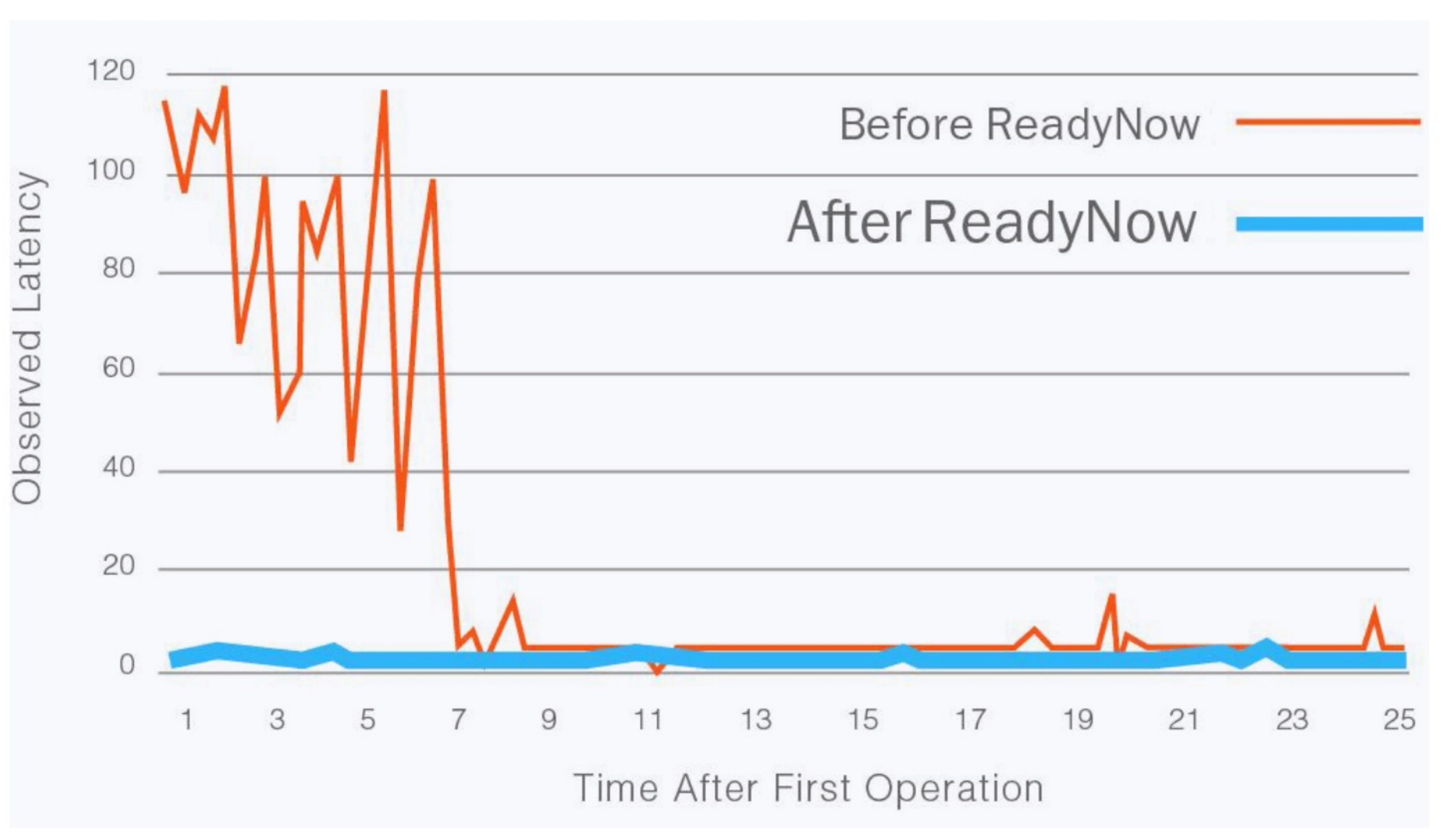
Ready Now: Принцип работы



1. Подаем нагрузку на сервис
2. Сохраняем профиль
3. Проверяем критерии достаточности (Время или Размер профиля)
4. Повторяем итерацию

https://docs.azul.com/prime/Use-ReadyNow

Ready Now: Принцип работы



Указываем профиль при старте:

-XX:ProfileLogIn=Profile_Gen2_input.log.gz

Получаем отсутствие пиков latency на старте!

https://docs.azul.com/prime/Use-ReadyNow

Ready Now

- Полностью решает проблему прогрева
- Платное решение
- Недоступно из-за санкций

Coordinated Restore at Checkpoint (CRaC)

Основан на технологии Linux CRIU

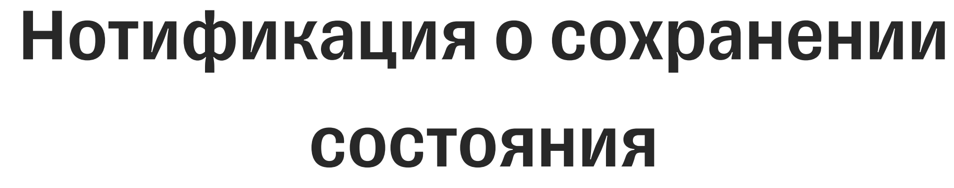
Доступен в Axiom JDK, Azul JDK, Liberica JDK

Позволяет создать снимок приложения

Использует снимок для старта приложения без фазы прогрева

https://docs.spring.io/spring-boot/reference/packaging/checkpoint-restore.html#packaging.checkpoint-restore

CraC: Принцип работы

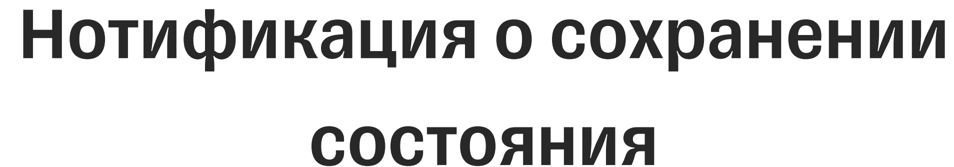


Указываем параметры
-XX:CRaCCheckpointTo=PATH

timeline

Startup + Warmup Phase

CraC: Принцип работы



Указываем параметры
-XX:CRaCCheckpointTo=PATH

timeline

Startup + Warmup Phase

Image
Registry

CraC: Принцип работы

Нотификация Restore Checkpoint

Указываем параметры
-XX:CRaCCheckpointFrom=PATH

timeline

Startup at peak performance

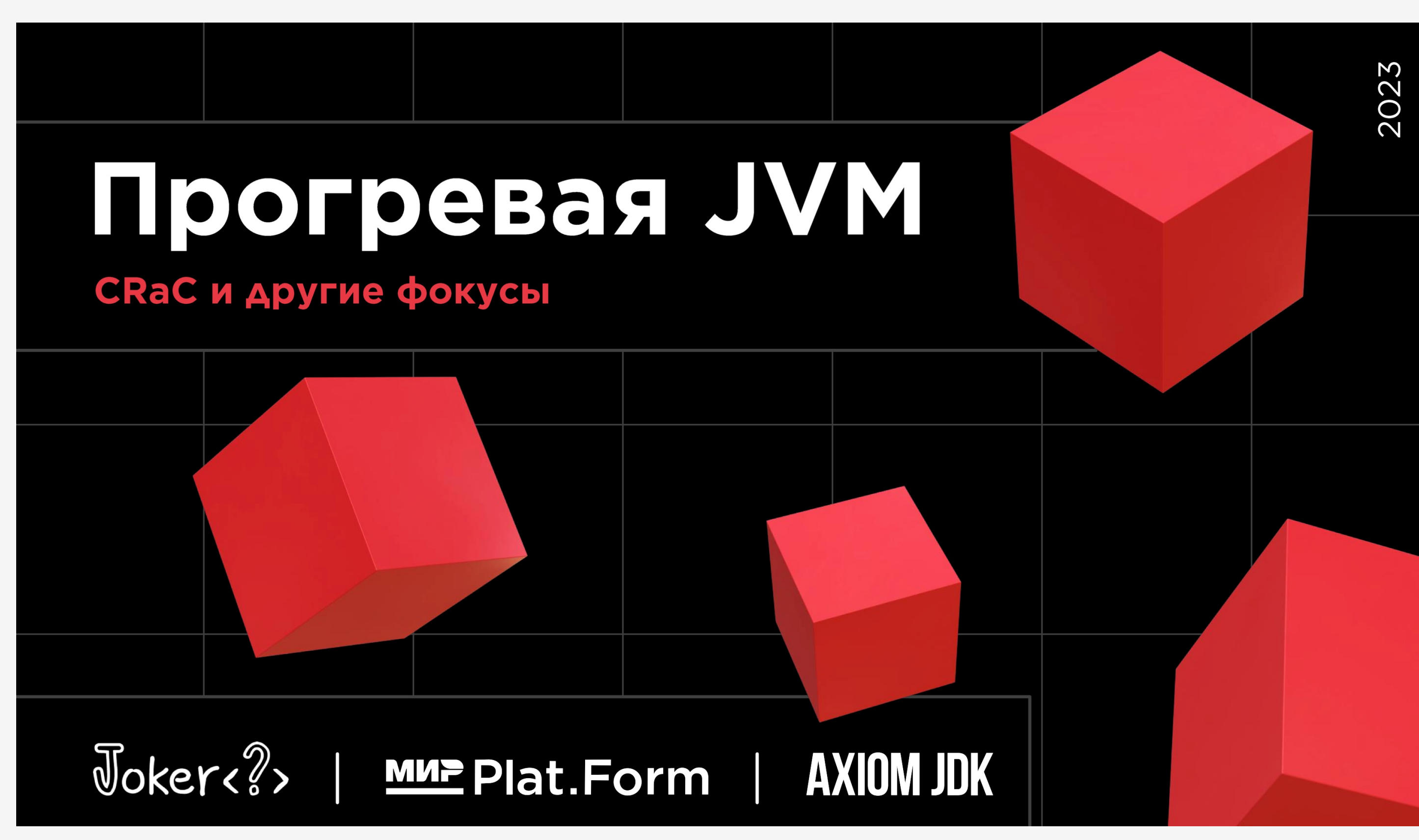
Image Registry

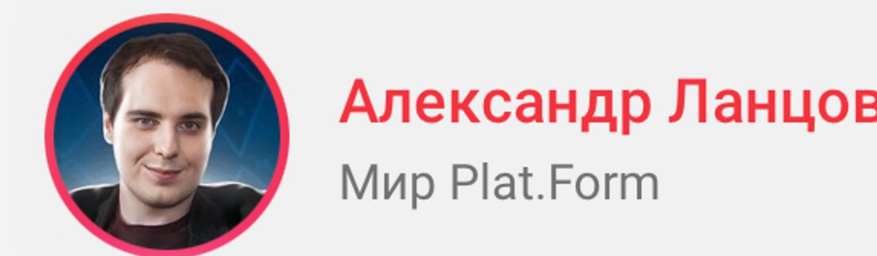
Coordinated restore at Checkpoint (CraC)

- Мгновенный старт с пиковой производительностью
- CRIU требует root доступа на машине для создания checkpoint’ов
- Неаккуратно созданные Checkpoint’ы могут хранить секреты
- Не нравится отделу безопасности

Coordinated restore at Checkpoint (CraC)

- Мгновенный старт с пиковой производительностью
- CRIU требует root доступа на машине для создания checkpoint'ов
- Неаккуратно созданные Checkpoint'ы могут хранить секреты
- Не нравится отделу безопасности





https://docs.spring.io/spring-boot/reference/packaging/checkpoint-restore.html#packaging.checkpoint-restore

Graal VM Native Image

Позволяет компилировать Java-приложения в нативный исполняемый файл

Плюсы:

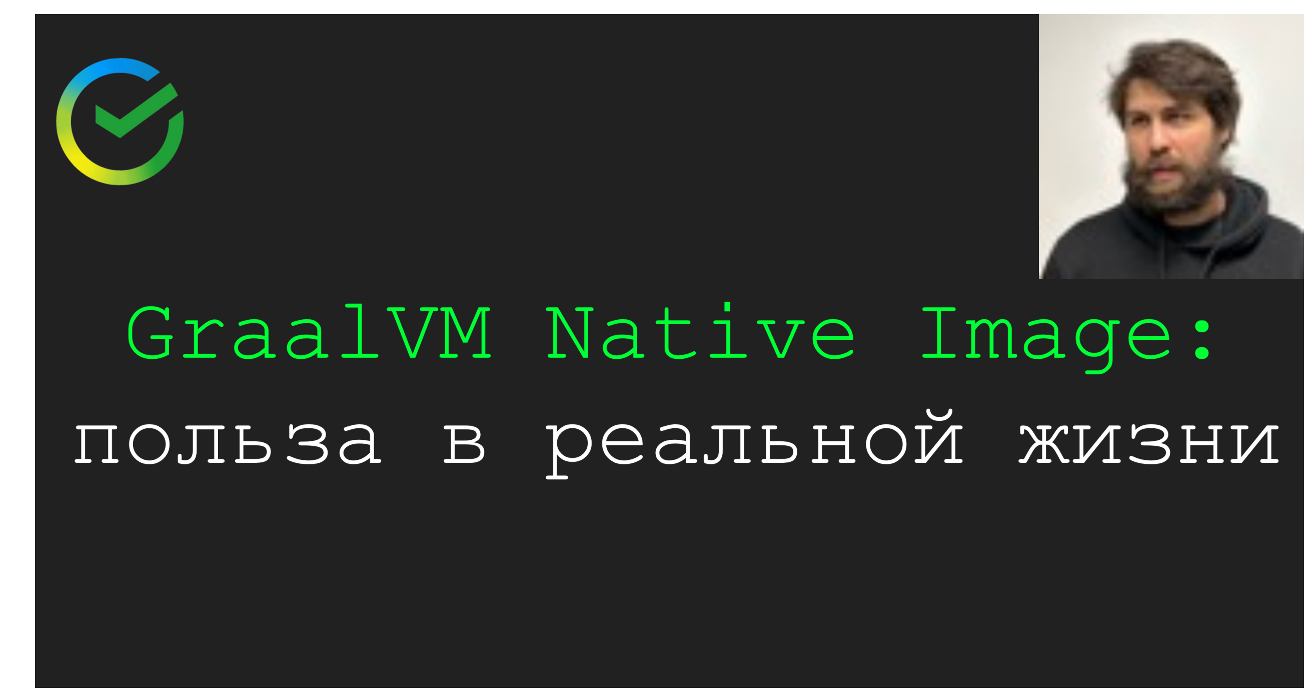
- Быстрый старт
- Низкое потребление ресурсов
- Меньший размер docker образов

https://www.graalvm.org/latest/reference-manual/native-image

Native Image

Graal VM Native Image

Из коробки получаем ahead-of-time компиляцию, нативный образ, и JVM тащить не надо?!



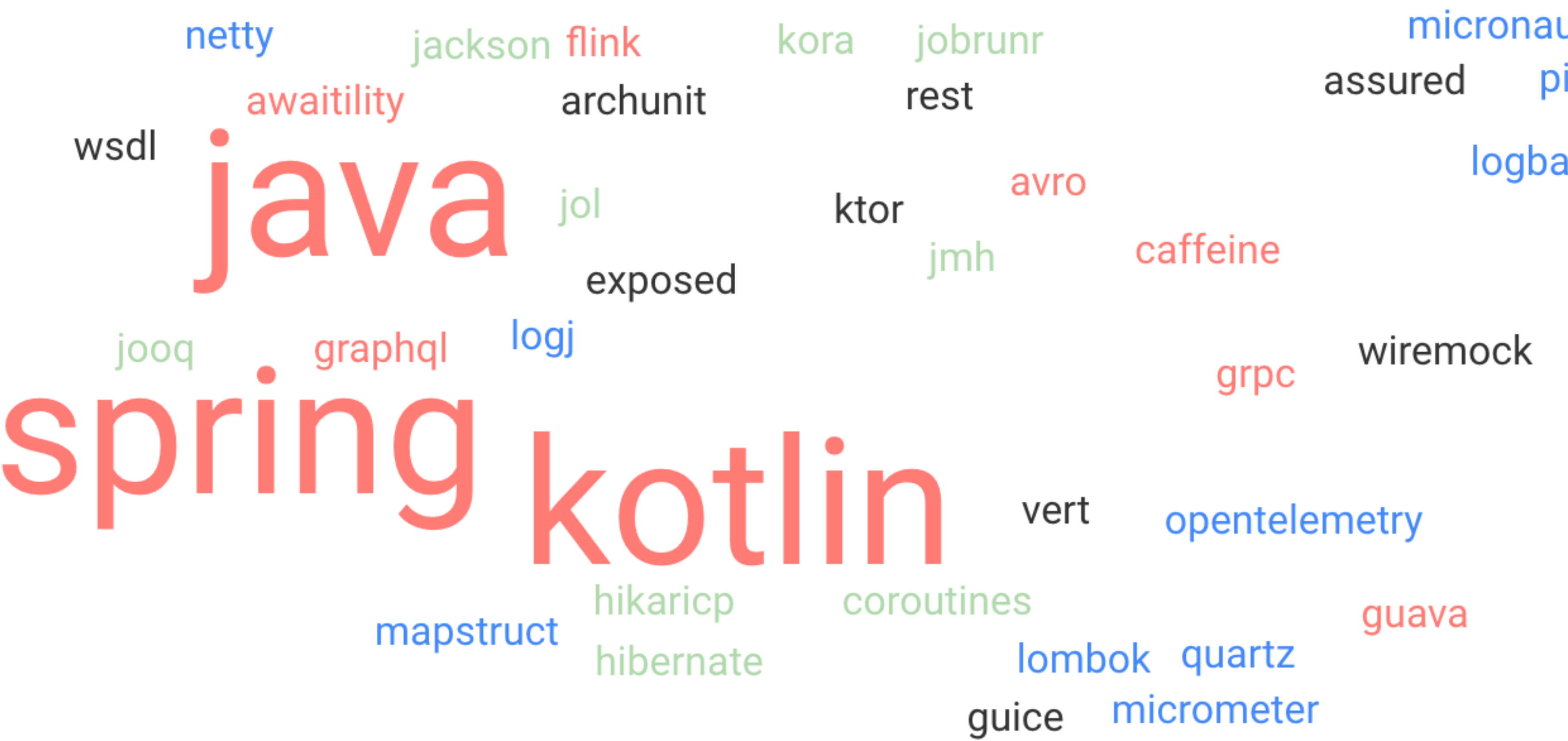
Андрей Чухлебов: GraalVM Native Image: польза в реальной жизни

Native Image

Graal VM Native Image

Можно ли использовать Graal VM если у тебя зоопарк технологий?

Сколько будет стоить внедрить GraalVM в команды?



Будут ли команды рады всем нюансам работы с Native Image?

Как много времени займет обучение команд?

radar.tcsbank.ru/java/

Graal VM Native Image

Почему не взяли Native Image?

- Большая стоимость внедрения и поддержки
- JIT + C2 быстрее native image
- У команд нет ресурсов на переезд

Native Image

https://www.graalvm.org/latest/reference-manual/native-image/

Project Leyden

Основная цель этого проекта — сократить время запуска, время достижения пиковой производительности и уменьшить объем памяти программ Java

OpenJDK

Project Leyden

Project Leyden

Основная цель этого проекта — сократить время запуска, время достижения пиковой производительности и уменьшить объем памяти программ Java

Уже можем использовать:

JEP 483: Ahead-of-Time Class Loading & Linking

Draft:

JEP draft: Ahead-of-Time Code Compilation

JEP draft: Ahead-of-Time Method Profiling

OpenJDK

Project Leyden

Методы прогрева

Использование возможностей инфраструктуры

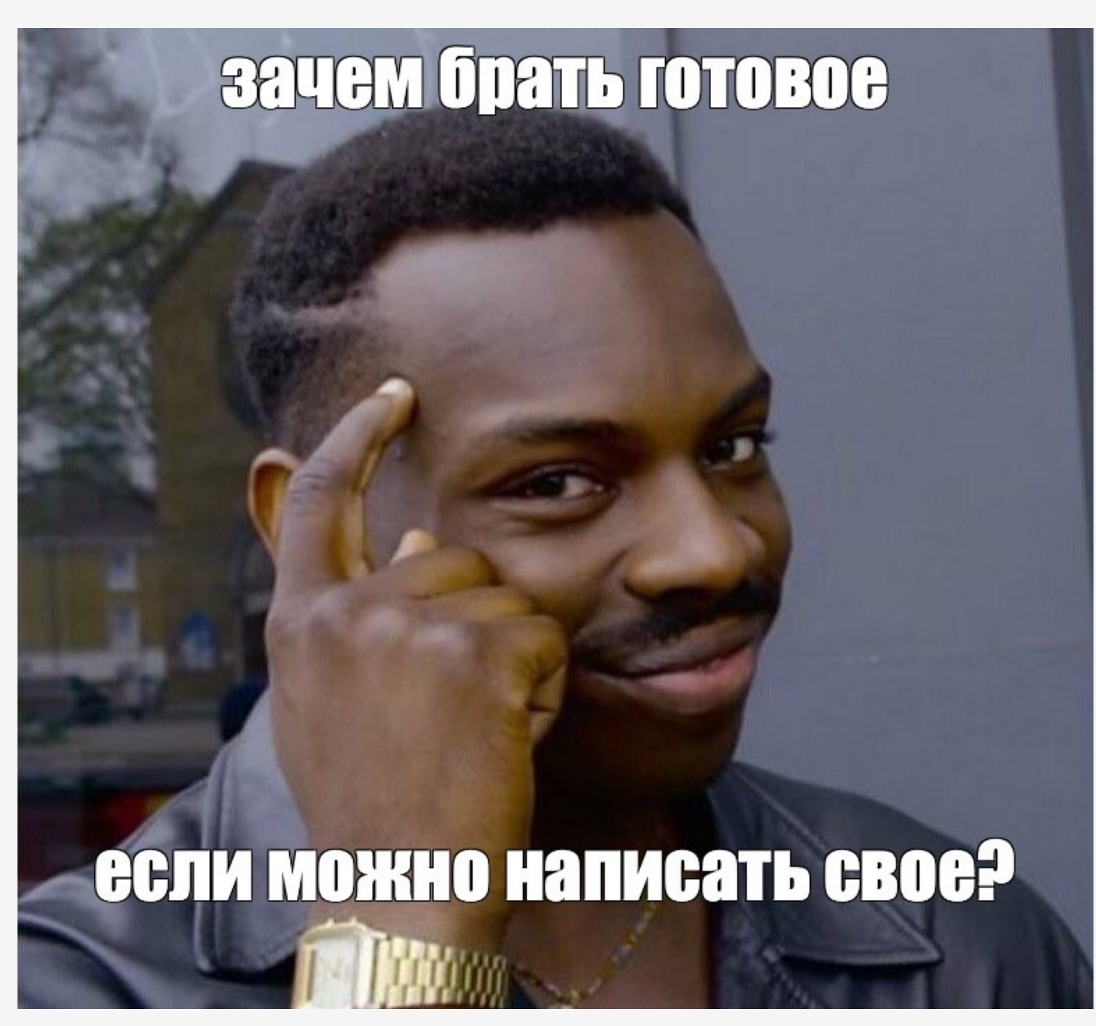
1. kubernetes VPA
2. Istio Traffic Management

Подготовка приложения к эксплуатации

1. Azul Ready Now
2. Coordinated Restore at Checkpoint (CRaC)
3. Graal VM Native Image
4. Project Layden

Собственное решения?!

Методы прогрева



Использование возможностей инфраструктуры

1. kubernetes VPA
2. Istio Traffic Management

Подготовка приложения к эксплуатации

1. Azul Ready Now
2. Coordinated Restore at Checkpoint (CRaC)
3. Graal VM Native Image
4. Project Layden

Собственное решения!

1. Warmup Library

Библиотека прогрева

Какие требования выдвигали:

Простое внедрение

Чем меньше строк кода придется написать, тем лучше - подключаем автоконфигурацию и готово!

Гибкая настройка

Конфигурация Yaml, должна покрывать большинство сценариев использования прогрева в командах

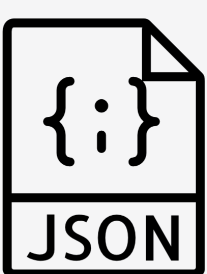
Не трогаем сборку

Код прогрева должен выполняется как часть инициализации приложения, процессы сборки и развертывания не должны усложняться

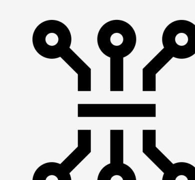
Что будем греть внутри микросервисов?

Какой стек мы используем:

Spring Boot



Jackson



Http Clients &
Connection Pools

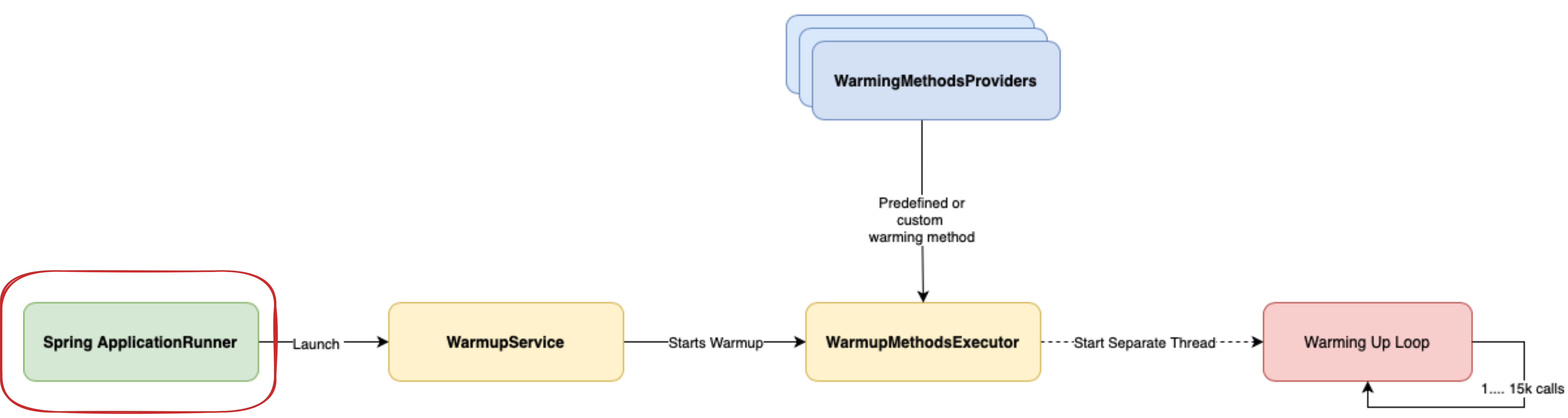
Библиотека прогрева

Что внутри?

- Стартер с автоконфигурацией
- Конфигурация через application.yml
- Примеры с базовыми кейсами

Механизм работы

Прогрев всегда начинается со старта Spring Application Runner



Механизм работы

Интерфейс ApplicationRunner в Spring Boot используется для выполнения кода при запуске приложения

```java
@FunctionalInterface
public interface ApplicationRunner extends Runner {

    /**
     * Callback used to run the bean.
     * Params: args – incoming application arguments
     * Throws: Exception – on error
     */
    void run(ApplicationArguments args) throws Exception;

}
```

Механизм работы

Приложение не будет принимать пользовательский трафик пока все ApplicationRunner'ы не будут завершены

```kotlin
open class WarmupAutoConfiguration {

    /**   You, 9 minutes ago • Uncommitted changes
     * Starting a warming-up process.
     * Readiness probe will be [ReadinessState.REFUSING_TRAFFIC]
     * until this ApplicationRunner is completed
     */
    @Bean
    open fun warmUpRunner(
        warmUpService: WarmupService
    ): ApplicationRunner = ApplicationRunner {
        warmUpService.warmUp()
    }
```

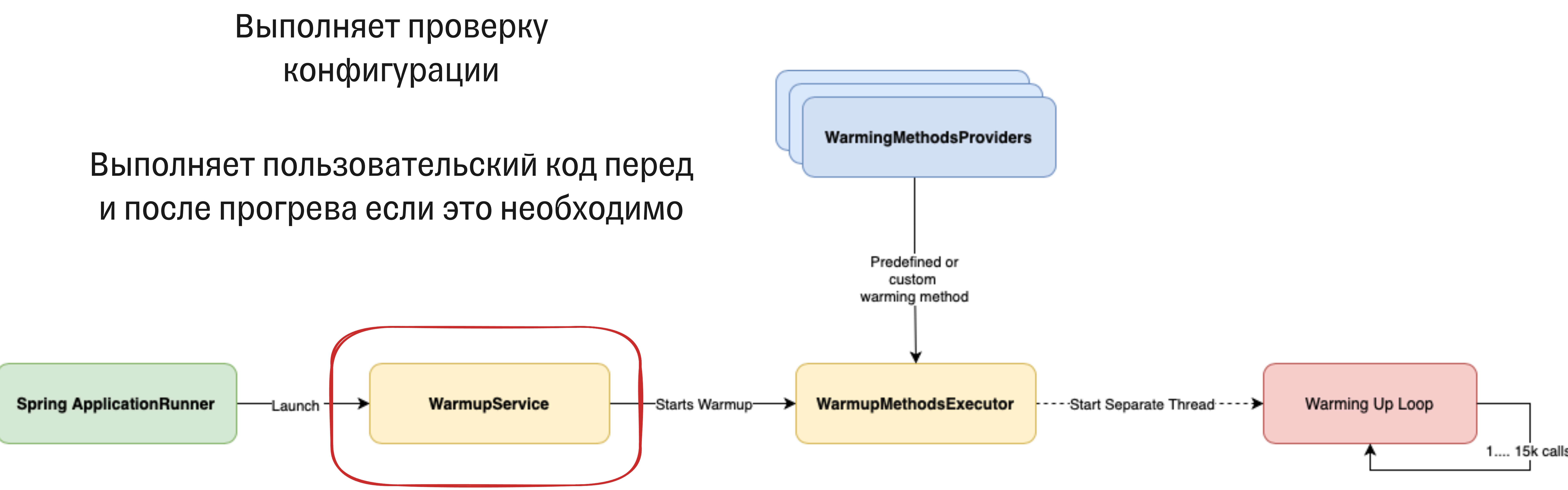
Запускаем прогрев

Механизм работы

Warmup Service инициализирует процесс прогрева

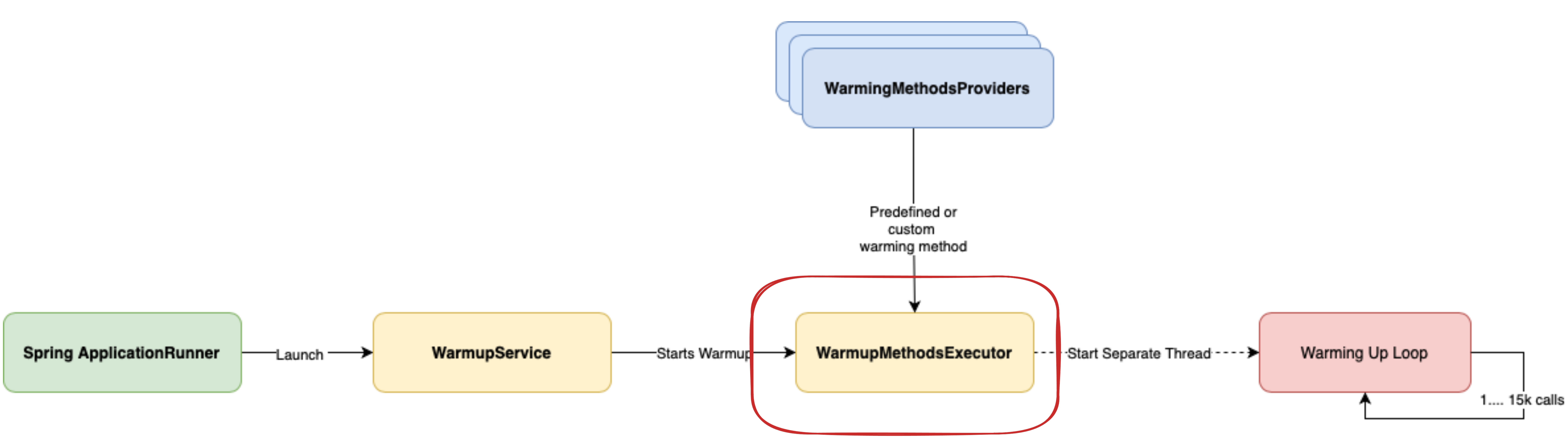
Выполняет проверку конфигурации

Выполняет пользовательский код перед и после прогрева если это необходимо



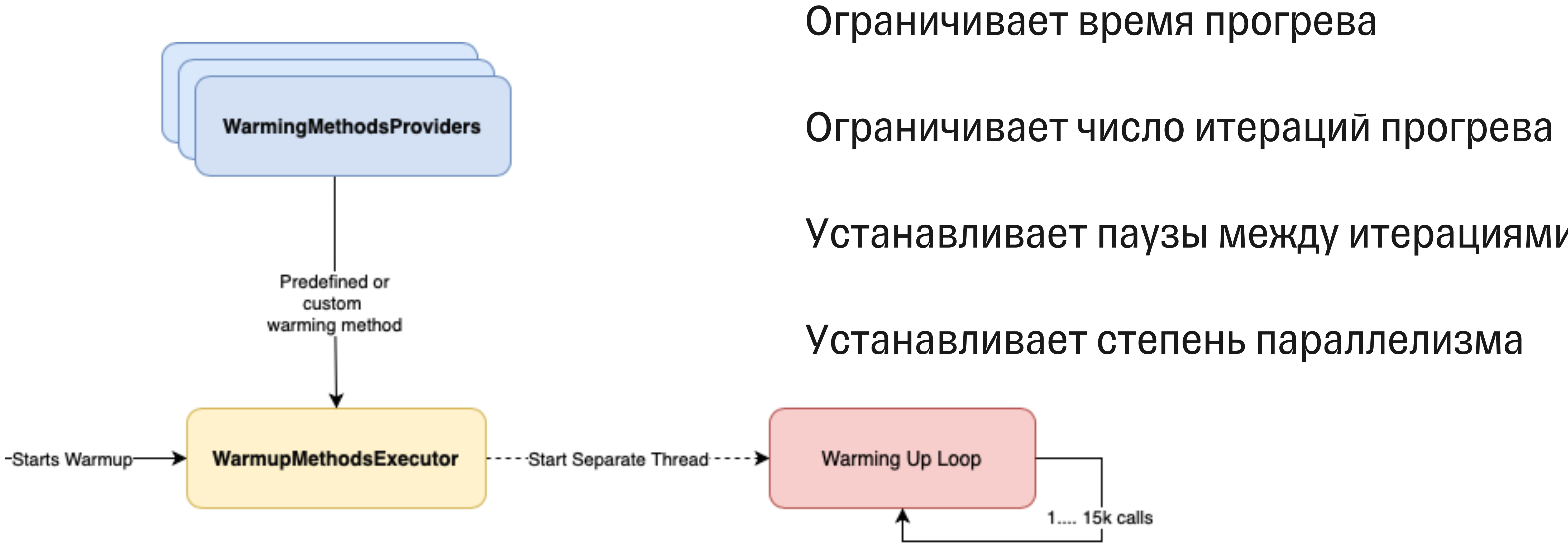
Механизм работы

WarmupMethodExecutor отвечает за многократный вызов прогревающих методов



Механизм работы

WarmupMethodExecutor отвечает за многократный вызова прогревающих методов



Ограничивает время прогрева

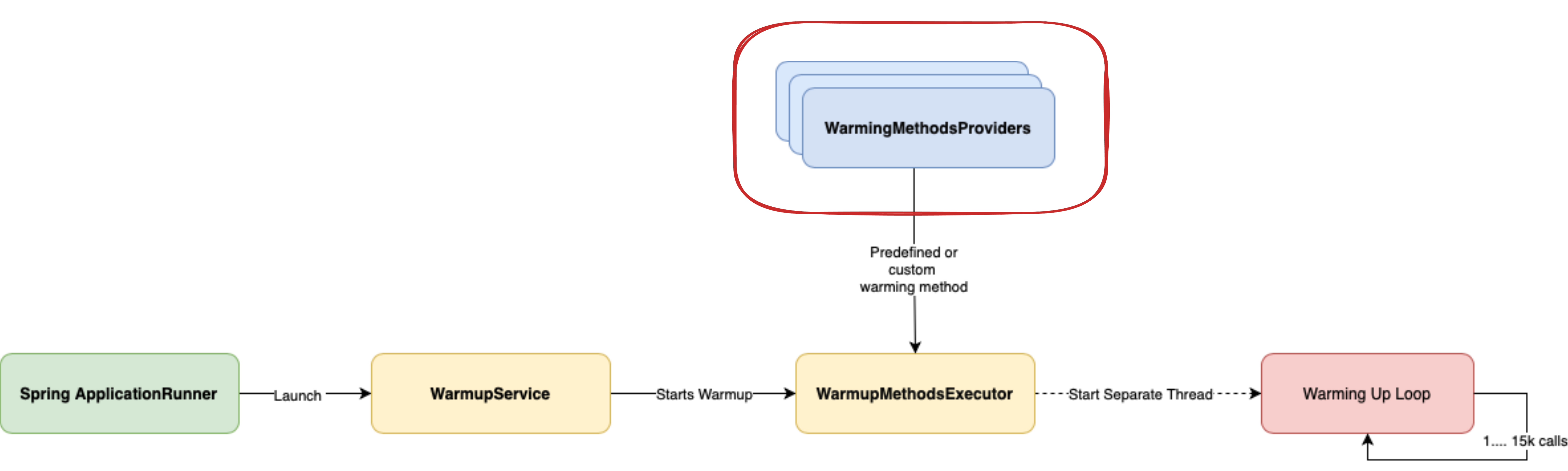
Ограничивает число итераций прогрева

Устанавливает паузы между итерациями

Устанавливает степень параллелизма

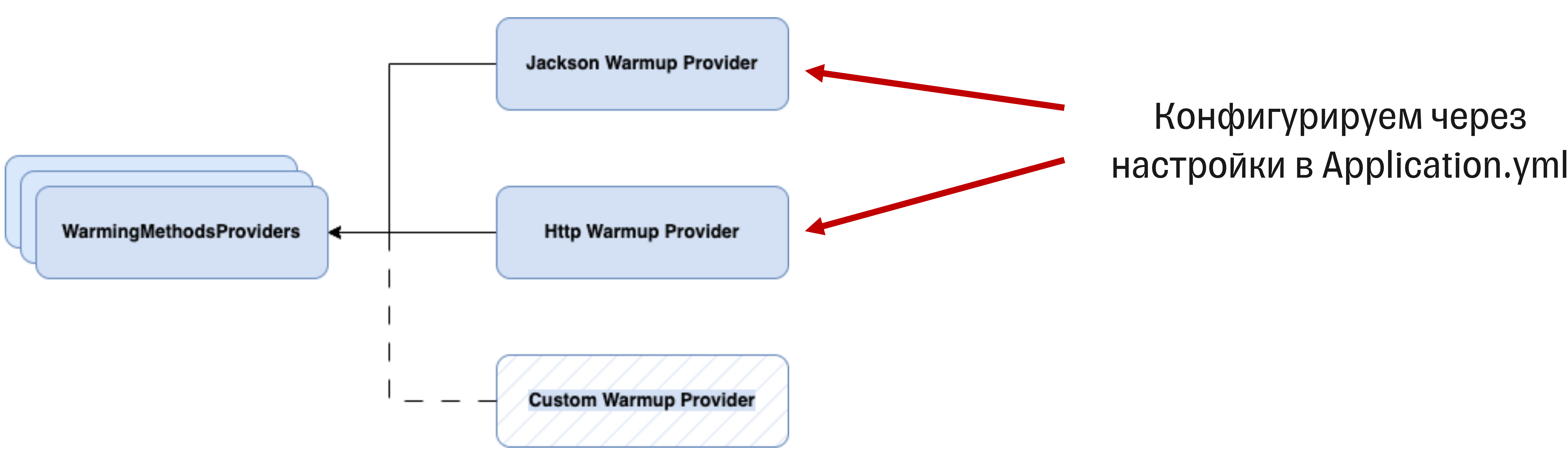
Механизм работы

Провайдеры определяют участки кода, которые будет вызывать Executor для прогрева



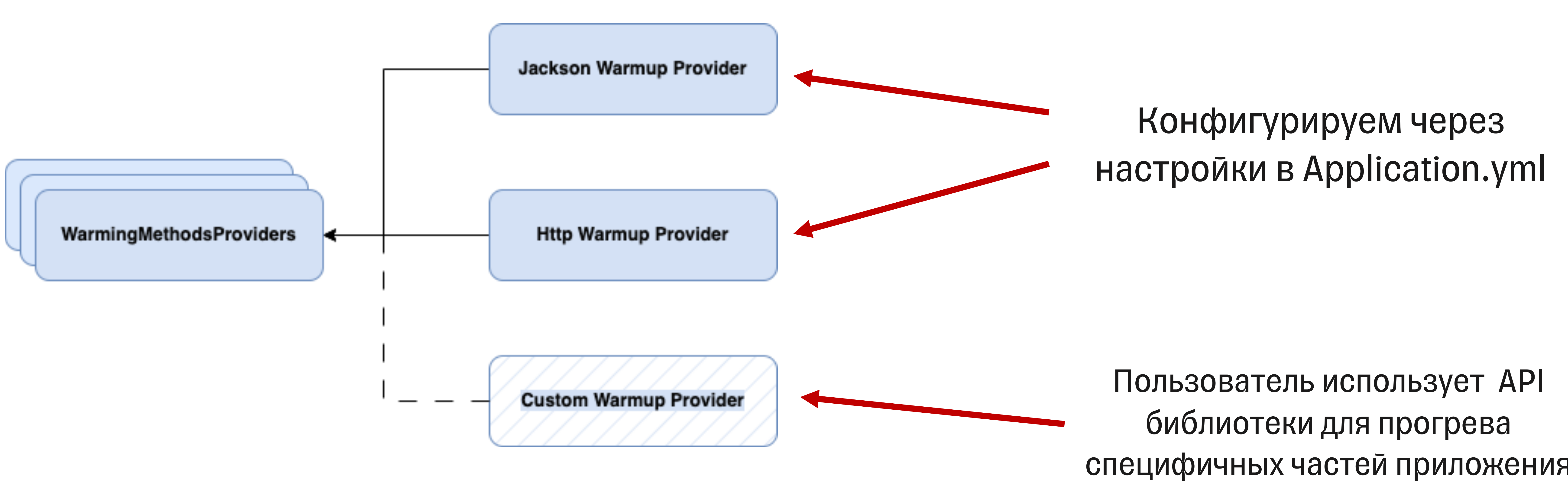
Механизм работы

Библиотека покрывает типичные кейсы прогрева



Механизм работы

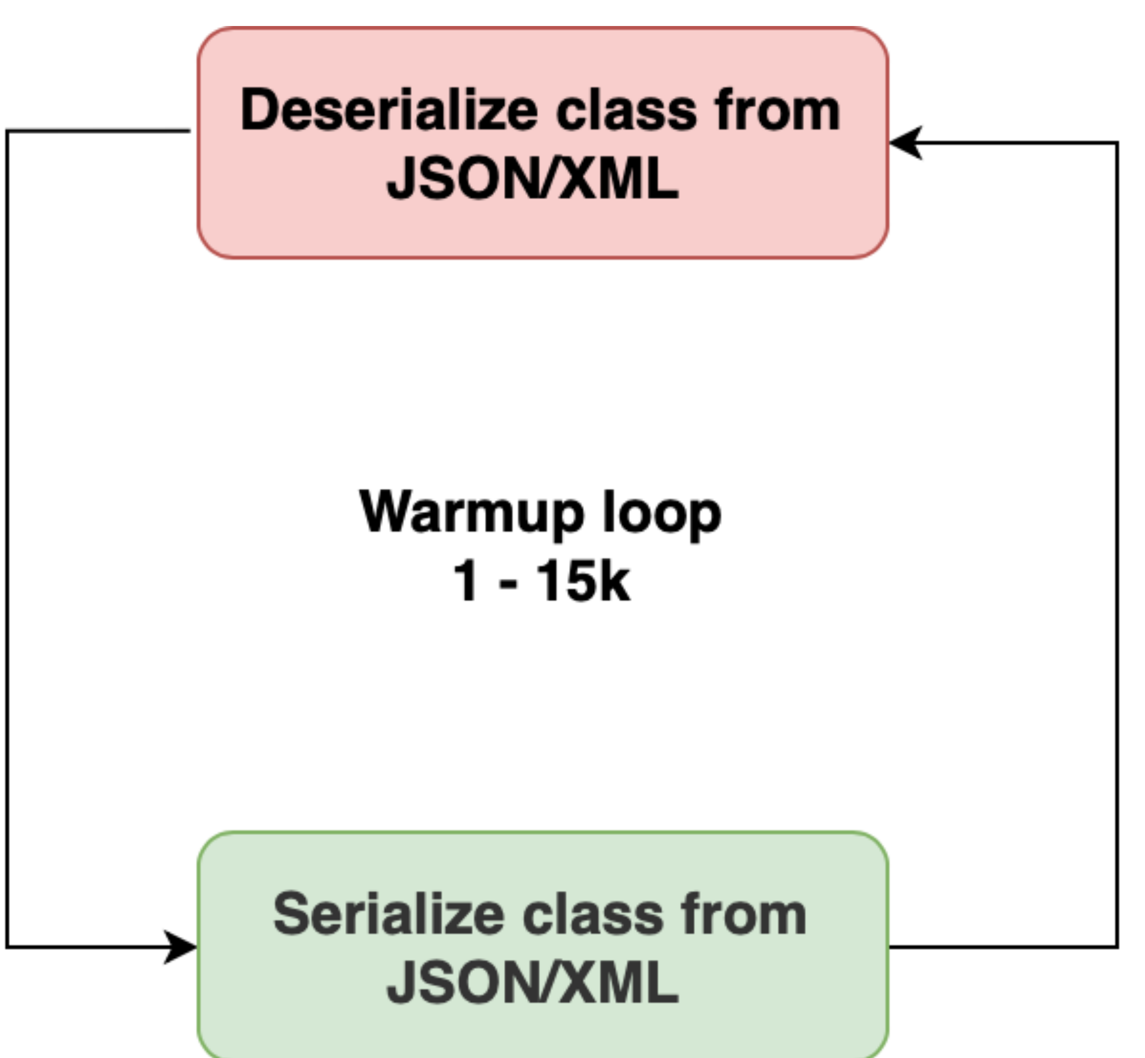
Пользователь может расширить функционал описав свой
Custom Warmup Provider



Прогрев Jackson Serializer

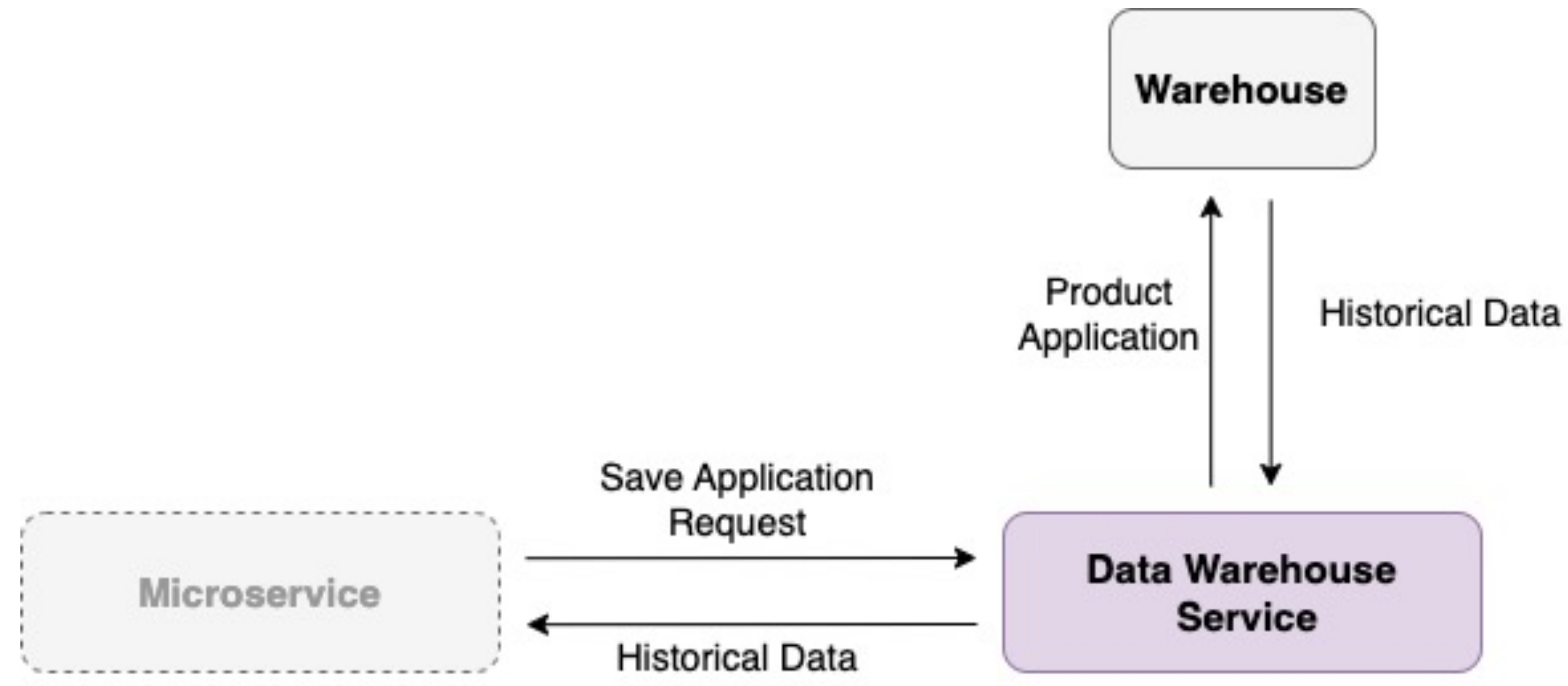
Сериализация и десериализация модели в цикле = Прогрев Serializer

```
fun warmupJackson(
    objectMapper: ObjectMapper,
    jsonContent: InputStream,
    type: KClass<*>
) {
    val obj = readValue(objectMapper, jsonContent, type)
    writeObject(objectMapper, obj)
}
```



Сколько времени занимает десериализация?

Сервис хранения заявок работает с большими JSON моделями



Сколько времени занимает десериализация?

Наш случай - большие модели и простая бизнес логика в сервисах

Сколько времени занимает десериализация?

Простой бенчмарк иллюстрирующий проблему

```kotlin
@State(Scope.Benchmark)
open class DeserializeHugeModel {

    private lateinit var objectMapper: ObjectMapper
    private lateinit var serializedHugeModel: String

    @Setup
    fun prepare() {
        objectMapper = objectMapperConfiguration.objectMapper()
        serializedHugeModel = String(ResourceLoader.readResource( resourcePath: "/sample/extremely_huge_model.json").readAllBytes())
    }

    @Benchmark
    @OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
    @BenchmarkMode(SingleShotTime)
    @Warmup(iterations = 0)
    @Fork(10)
    open fun deserializeModelOf22KLines() {
        objectMapper.readValue<Application>(serializedHugeModel)
    }

    @Benchmark
    @OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
    @BenchmarkMode(SingleShotTime)
    @Warmup(iterations = 1)
    @Fork(10)
    open fun deserializeModelOf22KLinesWith1WarmupIteration() {
        objectMapper.readValue<Application>(serializedHugeModel)
    }
}
```

Сколько времени занимает десериализация?

Инициализируем ObjectMapper и загружаем модель из ресурсов

```
private lateinit var objectMapper: ObjectMapper
private lateinit var serializedHugeModel: String

@Setup
fun prepare() {
    objectMapper = objectMapperConfiguration.objectMapper()
    serializedHugeModel = String(ResourceLoader.readResource( resourcePath: "/sample/extremely_huge_model.json").readAllBytes())
}
```

Сколько времени занимает десериализация?

Десериализуем модель без прогрева:

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 0)
@Fork(10)
open fun deserializeModelOf22KLines() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Используем @Fork вместо @Measurement, чтобы избежать влияния JIT компиляции на отдельный тест

Всего проводим 10 измерений

Сколько времени занимает десериализация?

Десериализуем модель с прогревов:

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 1)
@Fork(10)
open fun deserializeModelOf22KLinesWith1WarmupIteration() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
}
```

Используем @Warmup, чтобы 1 раз выполнить тест перед измерением

Всего проводим 10 измерений

Сколько времени занимает десериализация?

Прогрева нет:

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 0)
@Fork(10)
open fun deserializeModelOf22KLines() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

1268,318 ± 73,862 ms

95 Процентиль:

1359 ms

Прогреваем 1 раз:

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 1)
@Fork(10)
open fun deserializeModelOf22KLinesWith1WarmupIteration() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

37,717 ± 6,814 ms

95 Процентиль:

45,868 ms

Сколько времени занимает десериализация?

Прогрева нет:

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 0)
@Fork(10)
open fun deserializeModelOf22KLines() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

1268,318 ± 73,862 ms

95 Процентиль:

1359 ms

Прогреваем 1 раз:

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 1)
@Fork(10)
open fun deserializeModelOf22KLinesWith1WarmupIteration() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

37,717 ± 6,814 ms

95 Процентиль:

45,868 ms

Откуда разница в 30 раз?

Сколько времени занимает десериализация?

Запустим профайлер и посмотрим на работу ObjectMapper.readValue...

Call Tree десериализации без прогрева:

```
100.0%  ru.tbank.example.warmup.MainKt.main()
 74.9%  com.fasterxml.jackson.databind.ObjectMapper.readValue(InputStream, Class)
  100.0%  com.fasterxml.jackson.databind.ObjectMapper._readMapAndClose(JsonParser, JavaType)
   86.4%  com.fasterxml.jackson.databind.ObjectMapper._findRootDeserializer(DeserializationContext, JavaType)
    100.0%  com.fasterxml.jackson.databind.DeserializationContext.findRootValueDeserializer(JavaType)
     100.0%  com.fasterxml.jackson.databind.deser.DeserializerCache.findValueDeserializer(DeserializationContext, DeserializerFactory, JavaType)
   13.6%  com.fasterxml.jackson.databind.deser.DefaultDeserializationContext.readRootValue(JsonParser, JavaType, JsonDeserializer, Object)
```

Примерно 86% процентов времени выполнения readValue мы тратим на инициализацию дезериализатор для модели и их кеширование в методе **findRootDeserializer**

Примерно 14% процентов времени мы парсим модель

Сколько времени занимает десериализация?

Запустим профайлер и посмотрим на работу ObjectMapper.readValue…

Call Tree десериализации с прогревом:

```
100.0%  ru.tbank.example.warmup.MainKt.main()
  90.0%  com.fasterxml.jackson.databind.ObjectMapper.readValue(InputStream, Class)
    100.0%  com.fasterxml.jackson.databind.ObjectMapper._readMapAndClose(JsonParser, JavaType)
      100.0%  com.fasterxml.jackson.databind.deser.DefaultDeserializationContext.readRootValue(JsonParser, JavaType, JsonDeserializer, Object)
        100.0%  com.fasterxml.jackson.databind.deser.BeanDeserializer.deserialize(JsonParser, DeserializationContext)
```

Все 100% семплов профайлера попадают только в ветку парсинга модели

Сколько времени занимает десериализация?

Подожди! А где JIT - компиляция?
Ты же просто кеши заполнил...

Сколько времени занимает десериализация?

А что там про уровни компиляции?

Снова вспомним пороги компиляции... Нам нужен Tier4CompileThreshold, что будет соответствовать C2

```
~/workspace/education/warmup/demo-spring | on main +81 !13 ?23  java -XX:+PrintFlagsFinal -version | grep -E 'Tier.*CompileThreshold'
    uintx IncreaseFirstTierCompileThresholdAt      = 50                                      {product} {default}
     intx Tier2CompileThreshold                    = 0                                       {product} {default}
     intx Tier3CompileThreshold                    = 2000                                    {product} {default}
     intx Tier4CompileThreshold                    = 15000                                   {product} {default}
openjdk version "21" 2023-09-19
OpenJDK Runtime Environment (build 21+35-2513)
OpenJDK 64-Bit Server VM (build 21+35-2513, mixed mode, sharing)
```

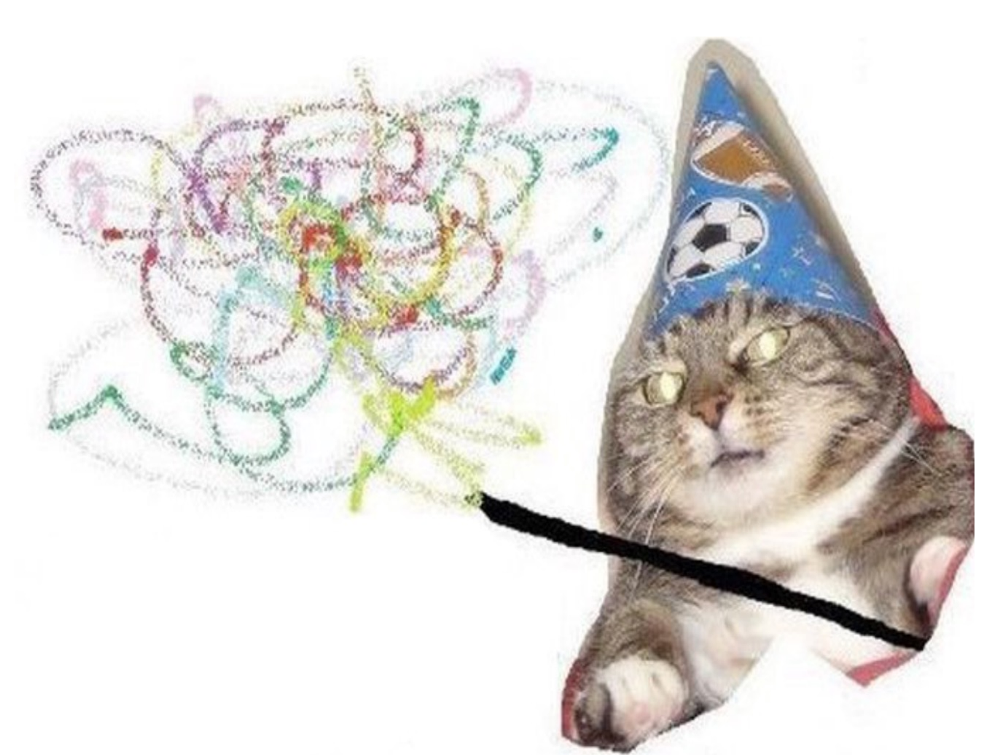
15 тысяч раз выполнить...

Сколько времени занимает десериализация?

Давайте добавим немного JIT компиляции…

Используем @Warmup, чтобы 15к раз выполнить тест перед замерами

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 15000)
@Measurement(iterations = 10)
@Fork(1)
open fun deserializeModelOf22KLinesWith15kWarmupIteration() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```



Сколько времени занимает десериализация?

Прогрева нет:

```kotlin
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 0)
@Fork(10)
open fun deserializeModelOf22KLines() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

1268,318 ± 73,862 ms

95 Процентиль:

1359 ms

Прогреваем Кеши:

```kotlin
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 1)
@Fork(10)
open fun deserializeModelOf22KLinesWith1WarmupIteration() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

37,717 ± 6,814 ms

95 Процентиль:

45,868 ms

Прогреваем с JIT:

```kotlin
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 15000)
@Measurement(iterations = 10)
@Fork(1)
open fun deserializeModelOf22KLinesWith15kWarmupIteration() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

7,285 ± 0,649 ms

95 Процентиль:

7,935 ms

Сколько времени занимает десериализация?

Прогрева нет:

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 0)
@Fork(10)
open fun deserializeModelOf22KLines() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

1268,318 ± 7[illegible] ms

95 Процентиль:

1359 ms

Прогреваем Кеши:

APPROVED

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 1)
@Fork(10)
open fun deserializeModelOf22KLinesWith1WarmupIteration() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

[illegible] ± 6,314 ms

95 Процентиль:

45,868 ms

Прогреваем с JIT:

```
@Benchmark
@OutputTimeUnit(java.util.concurrent.TimeUnit.MILLISECONDS)
@BenchmarkMode(SingleShotTime)
@Warmup(iterations = 15000)
@Measurement(iterations = 10)
@Fork(1)
open fun deserializeModelOf22KLinesWith15kWarmupIteration() {
    objectMapper.readValue<Application>(serializedHugeModel)
}
```

Среднее время выполнения:

[illegible] ± 0,649 ms

95 Процентиль:

7,935 ms

С Бенчмарками понятно, а что там с библиотекой?

Прогрев Jackson Serializer

Простейшая конфигурация:

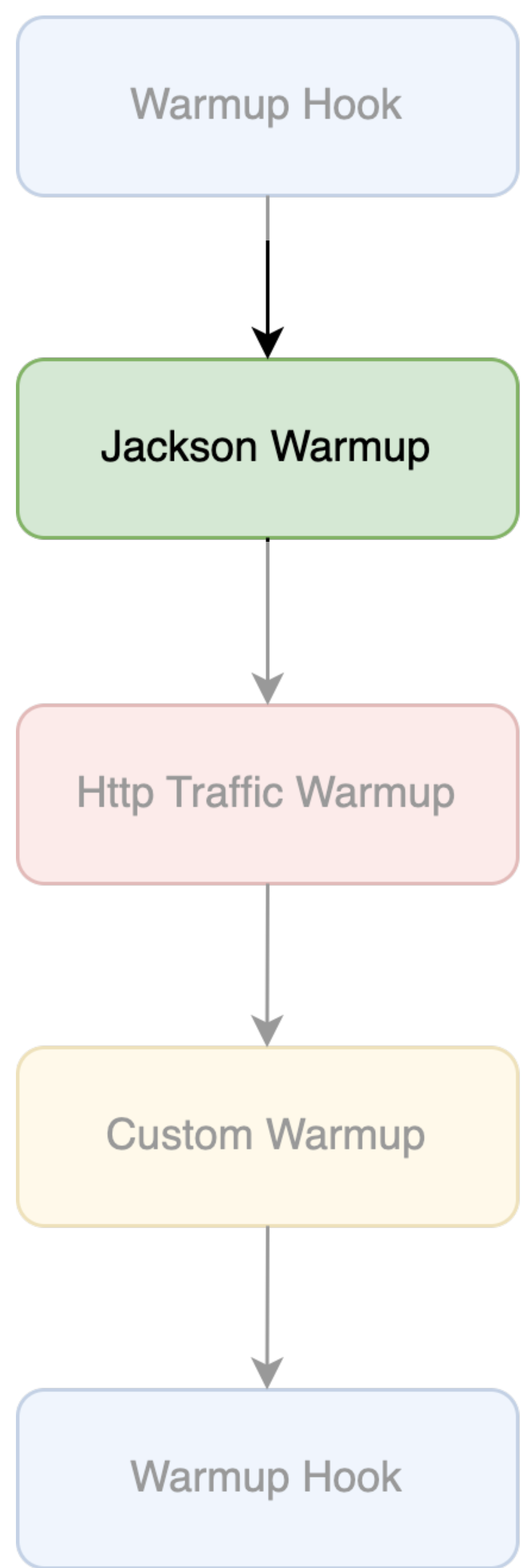
```yaml
top:
  warmup:
    enabled: true
    jackson:
      enabled: true
      maximumWarmupTime: 20s # Время, которое прогрев не будет превышать
      classes:
        - className: ru.tbank.example.HugeModel  # FQN класса
          json-file: sample/extremely_huge_model.json  # JSON, который будем мапить на наш класс
          warmup-iterations: 1 # кол-во итераций прогрева
```

Прогреем Кеши Jackson Serializer

Путь до нашего json в директории с ресурсами

- resources
  - sample
    - extremely_huge_model.json

Фазы Прогрева



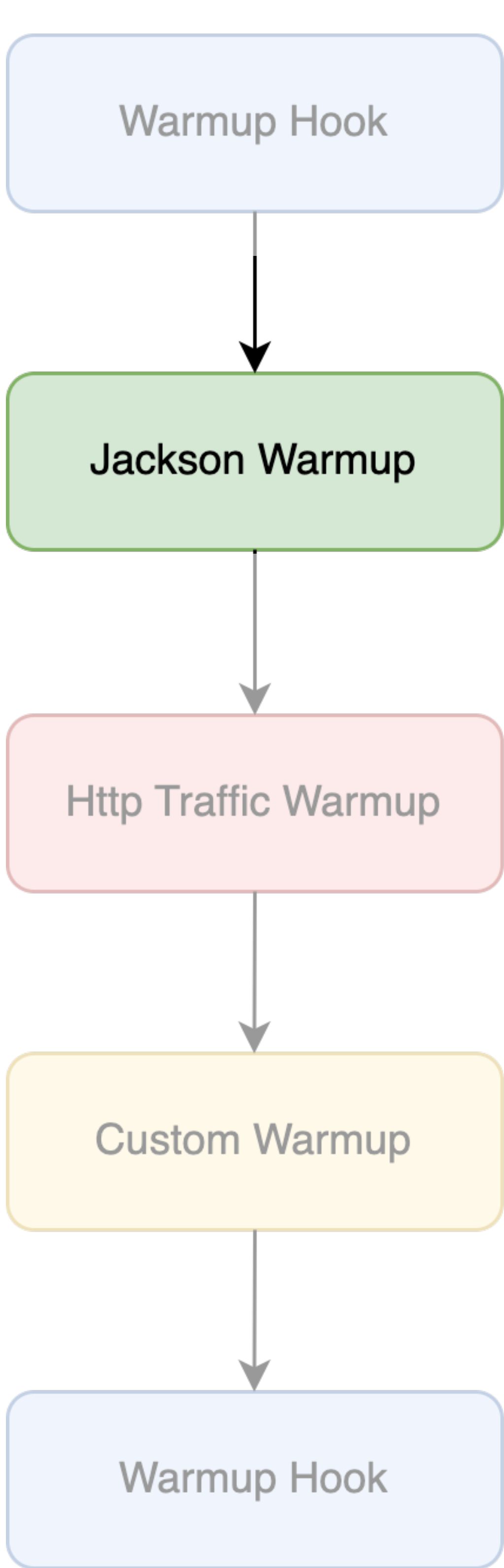
Прогрев Jackson Serializer

Простейшая конфигурация:

```yaml
top:
  warmup:
    enabled: true
    jackson:
      enabled: true
      maximumWarmupTime: 20s # Время, которое прогрев не будет превышать
      classes:
        - className: ru.tbank.example.HugeModel  # FQN класса
          json-file: sample/extremely_huge_model.json  # JSON, который будем мапить на наш класс
          warmup-iterations: 15000 # кол-во итераций прогрева
```

Изменим кол-во warmup-iteration до 15к,
чтобы C2 применил агрессивные оптимизации

Фазы Прогрева



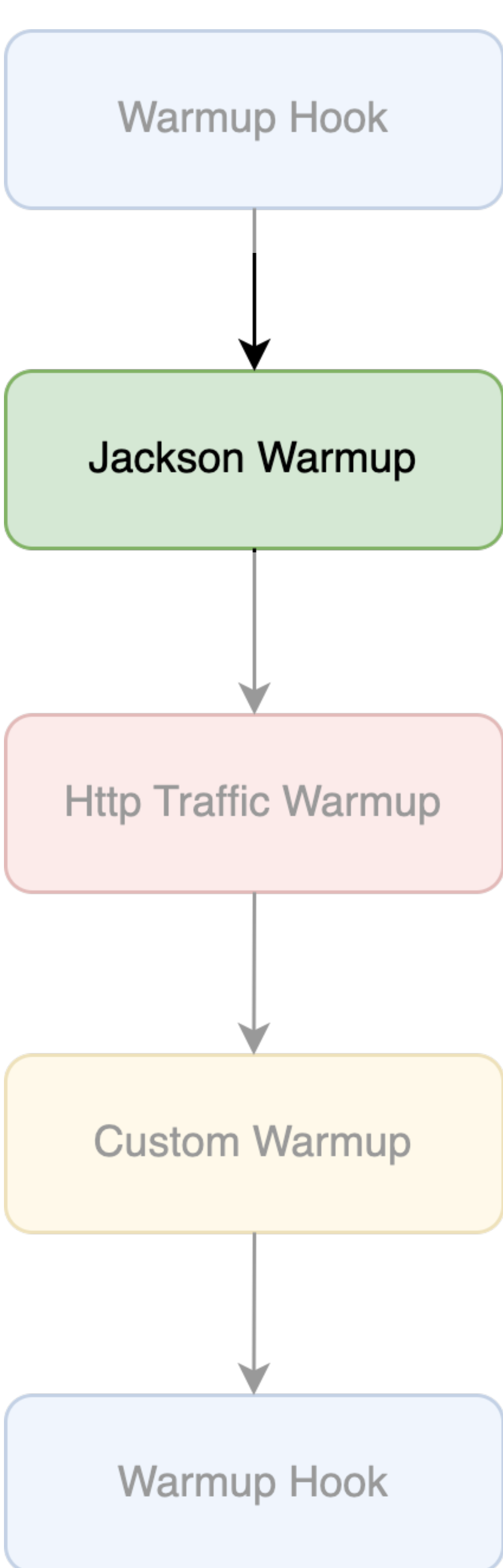
Прогрев Jackson Serializer

Простейшая конфигурация:

```
top:
  warmup:
    enabled: true
    jackson:
      enabled: true
      maximumWarmupTime: 20s # Время, которое прогрев не будет превышать
      classes:
        - className: ru.tbank.example.HugeModel  # FQN класса
          json-file: sample/extremely_huge_model.json  # JSON, который будем мапить на наш класс
          warmup-iterations: 15000 # кол-во итераций прогрева
```

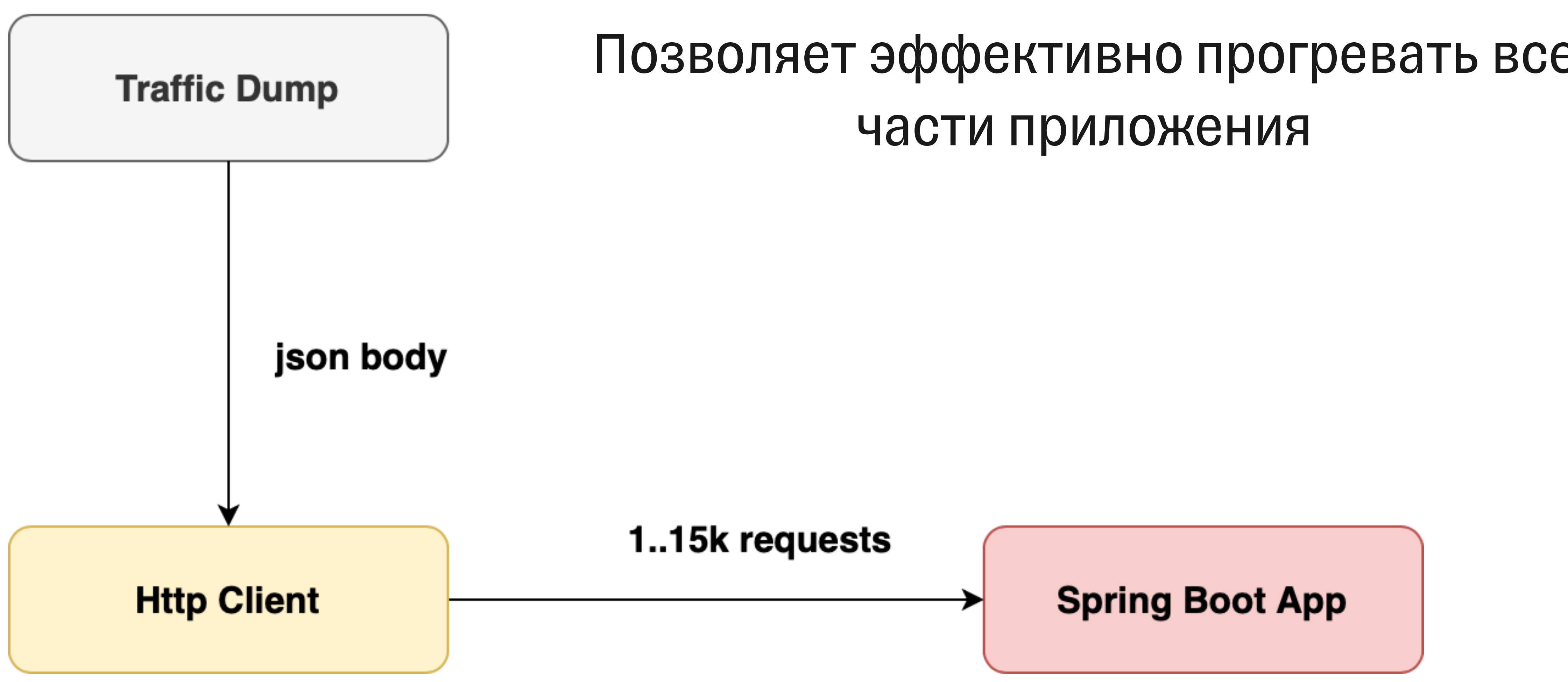
Что если мы не хотим хранить в ресурсах json на 22к строк?

Фазы Прогрева



Прогрев Spring Controllers

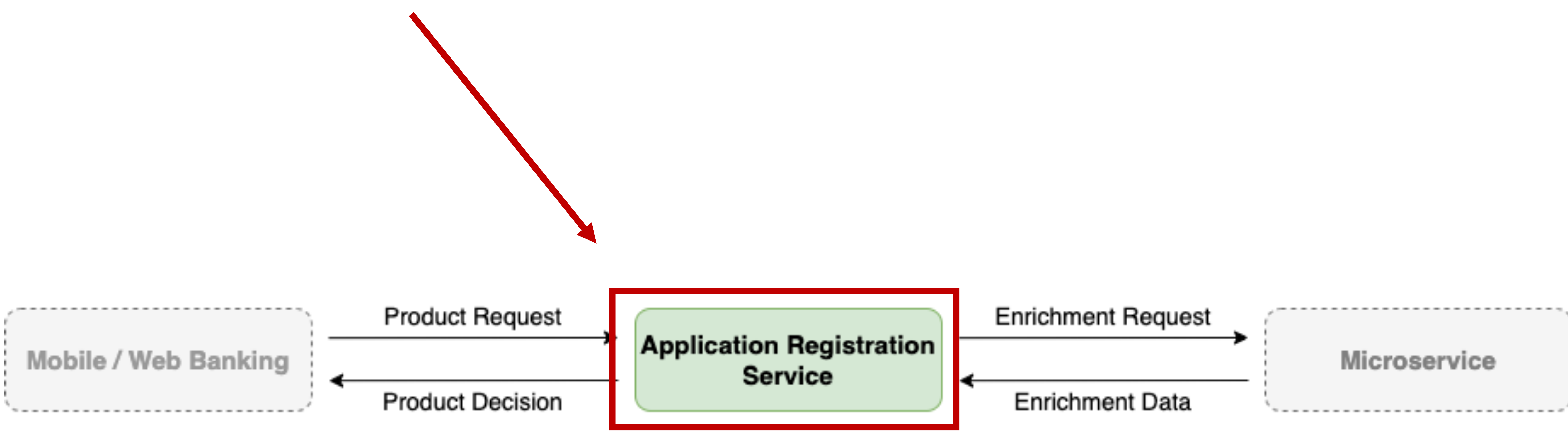
Дамп Трафика + Http Client = Прогрев Spring Controllers



Позволяет эффективно прогревать все части приложения

Прогрев Spring Controllers

Рассмотрим контроллер из Микросервиса Регистрации Заявок:



Прогрев Spring Controllers

Рассмотрим контроллер из Микросервиса Регистрации Заявок:

```
@RestController
@RequestMapping("/applications")
class ApplicationController(
    @Autowired private val dataEnrichmentService: DataEnrichmentService,
    @Autowired private val scoringService: ScoringService,
    @Autowired private val applicationStorageService: ApplicationStorageService
) {

    @GetMapping("/{id}")
    fun getApplication(@PathVariable id: String): ResponseEntity<ApplicationResponse> {...}

    @PostMapping
    fun registerApplication(@RequestBody applicationRequest: ApplicationRequest): ResponseEntity<ApplicationResponse> {...}
}
```

GET запрос для получения исторических заявок из БД

POST запрос валидирует заявку от мобильного приложения и пробрасывает ее в микросервисы для скоринга

Прогрев Spring Controllers

Замерим время выполнения POST запроса к приложению

```
POST http://localhost:8080/applications
Content-Type: application/json

{
  "clientId": "1",
  "clientName": "Sergey B",
  "clientEmail": "s.baldin@gmail.com",
  "products": [
    {
      "productId": "1",
      "productName": "Mortgage"
    },
    {
      "productId": "2",
      "productName": "CreditCard"
    }
  ]
}
```

Время выполнения запроса к приложению сразу после старта:

Среднее время выполнения:

498 ± 12,862 ms

95 Процентиль:

530 ms

Прогрев Spring Controllers

Добавим конфигурацию прогрева и снова отправим запрос

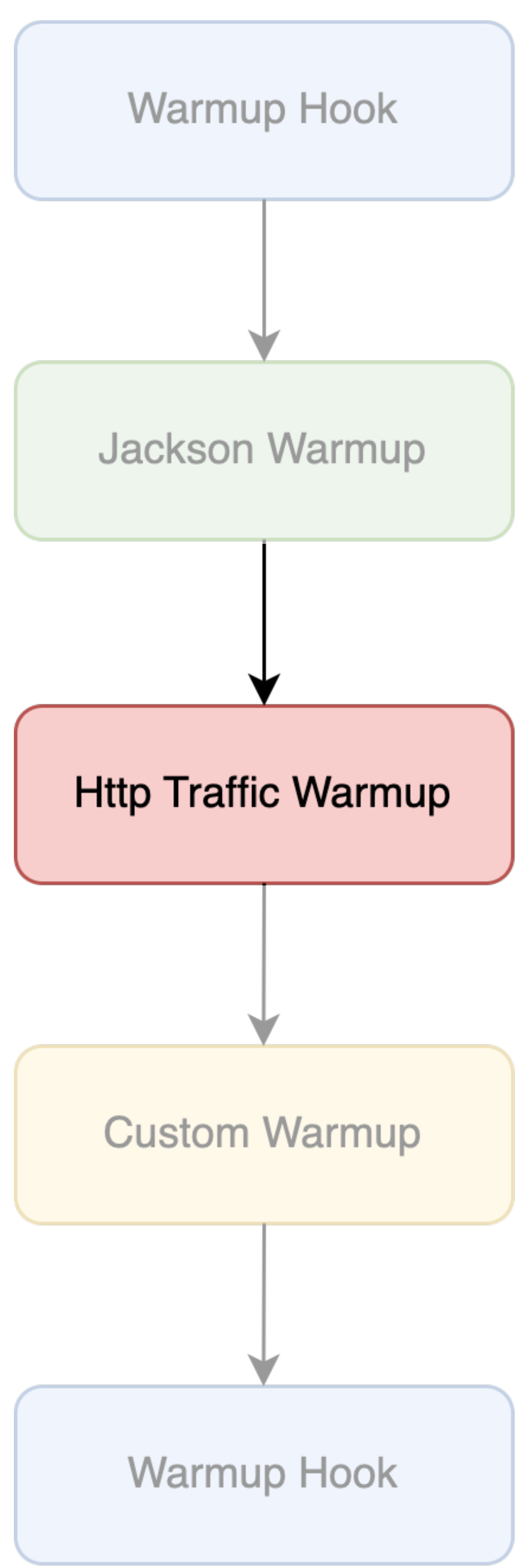
```yaml
top:
  warmup:
    http:
      enabled: true
      maximumWarmupTime: 20s  # задаем максимальное время прогрева
      endpoints:
        - path: /applications   # HTTP path метода регистрации заявок
          method: POST # Http method
          headers:
            Content-Type: application/json
          body-file: warmup/test-application.json
          warmup-iterations: 1
        - path: /applications/app-1  # HTTP path метода получения заявок
          method: GET # Http method
          warmup-iterations: 1
```

Укажем пути до методов, которые будем прогревать

Для POST запроса укажем json payload

Замерим, что даст 1 итерация прогрева

Фазы Прогрева

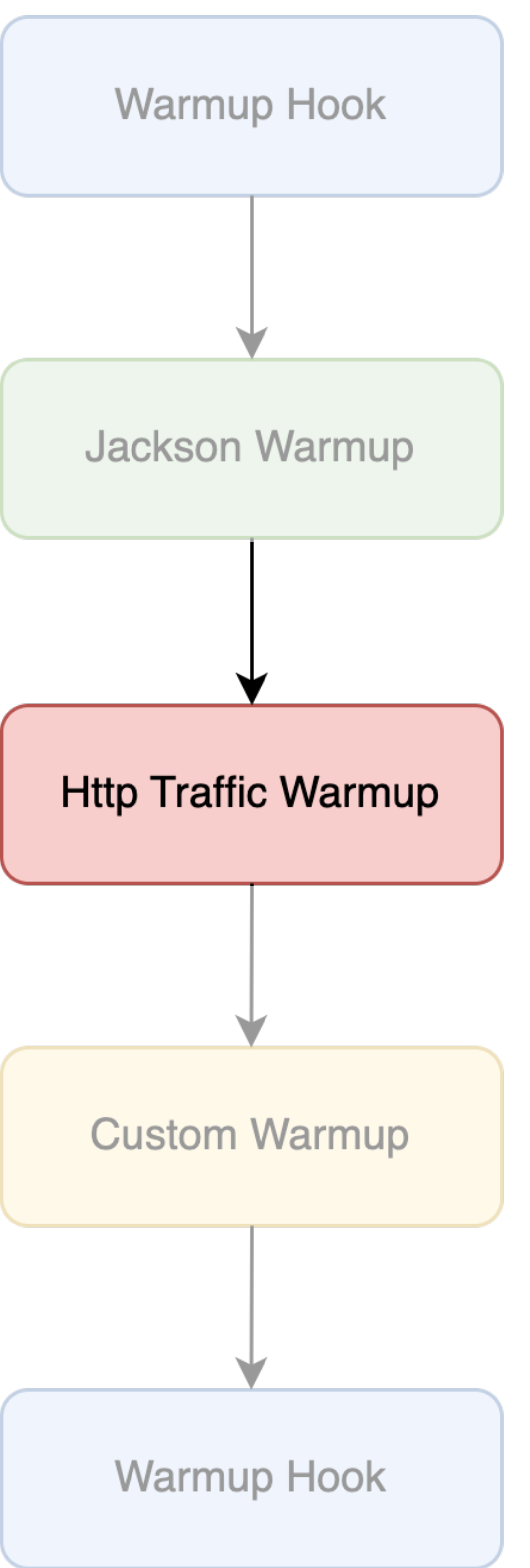


Прогрев Spring Controllers

Даже одна итерация прогрева значительно сокращает latency

Прогрева нет	1 итерация прогрева
Среднее время выполнения:	Среднее время выполнения:
498 ± 12,86 ms	15.8 ± 6.96 ms
95 Процентиль:	95 Процентиль:
530 ms	20.3 ms

Фазы Прогрева



Прогрев Spring Controllers

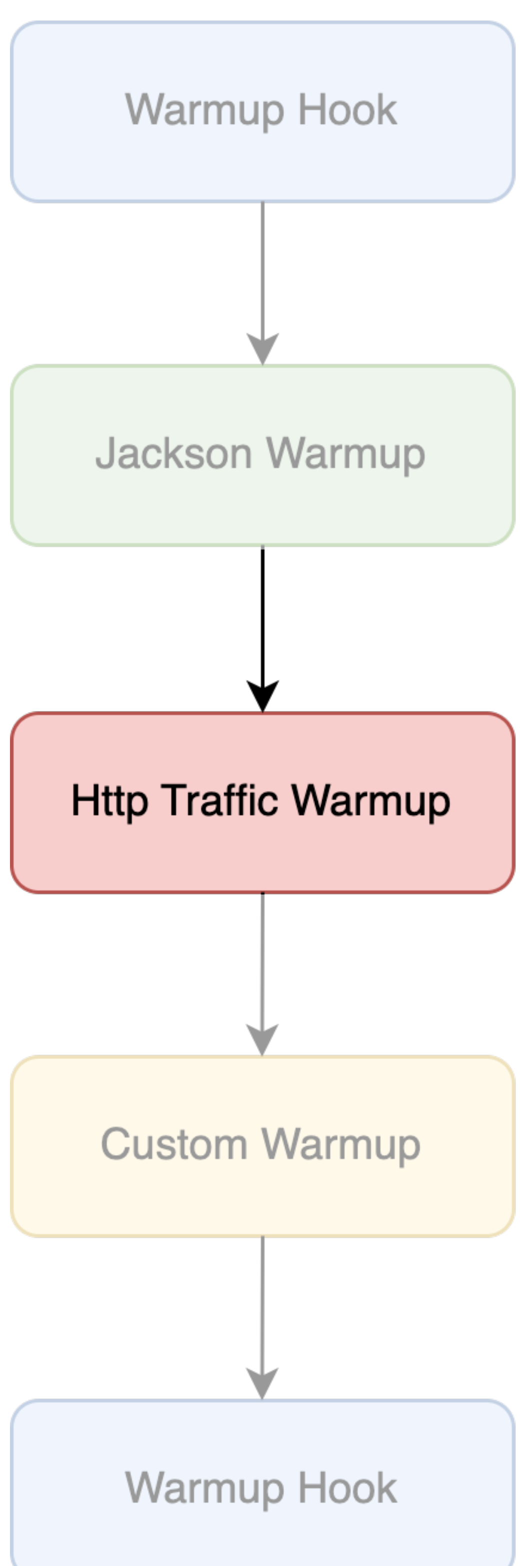
А можем прогреть еще сильнее?

Добавим итераций прогрева

```yaml
top:
  warmup:
    enabled: true
    check-configuration: true
    http:
      enabled: true
      maximumWarmupTime: 20s  # задаем максимальное время прогрева
      endpoints:
        - path: /applications   # HTTP path метода регистрации заявок
          method: POST # Http method
          headers:
            Content-Type: application/json
            Warmup: true # Добавим флаг, для переключения
          body-file: warmup/test-application.json
          warmup-iterations: 1000
        - path: /applications/app-1  # HTTP path метода получения заявок
          method: GET # Http method
          warmup-iterations: 1000
```

А почему не 15к?

Фазы Прогрева



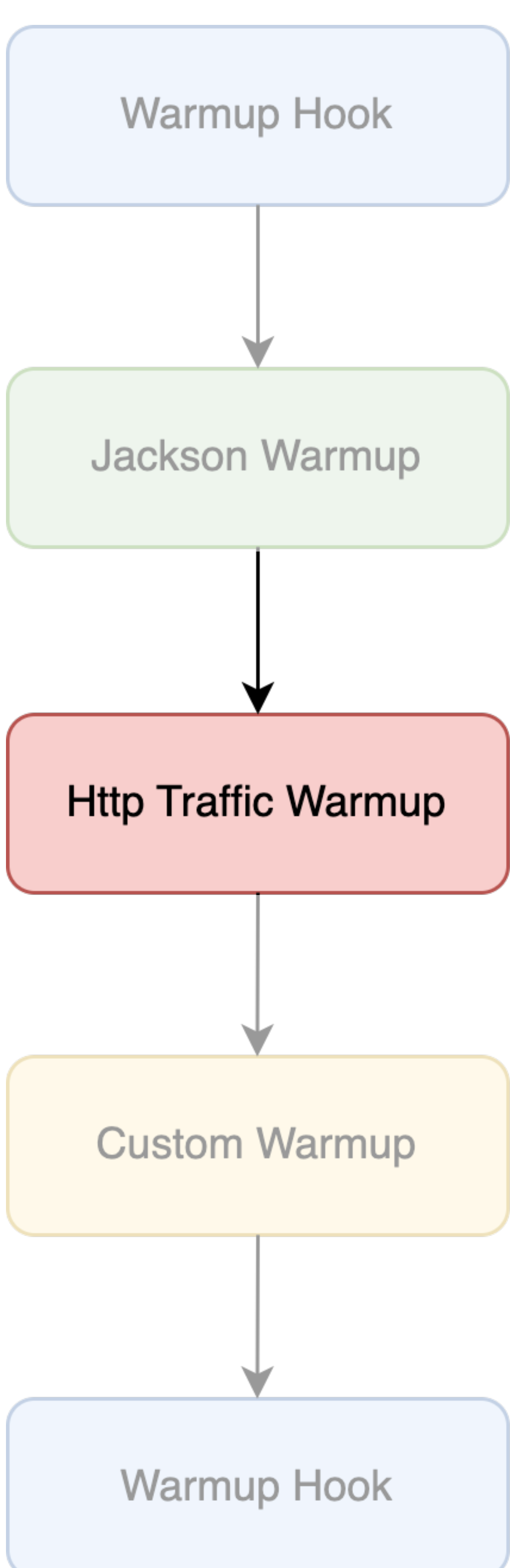
Прогрев Spring Controllers

Мы не всегда может позволить себе 15к итераций,
3х время прогрева ограничено

```yaml
top:
  warmup:
    enabled: true
    check-configuration: true
    http:
      enabled: true
      maximumWarmupTime: 20s  # задаем максимальное время прогрева
      endpoints:
        - path: /applications   # HTTP path метода регистрации заявок
          method: POST # Http method
          headers:
            Content-Type: application/json
            Warmup: true # Добавим флаг, для переключения
          body-file: warmup/test-application.json
          warmup-iterations: 1000
        - path: /applications/app-1  # HTTP path метода получения заявок
          method: GET # Http method
          warmup-iterations: 1000
```

У нас всегда есть лимит по времени

Фазы Прогрева

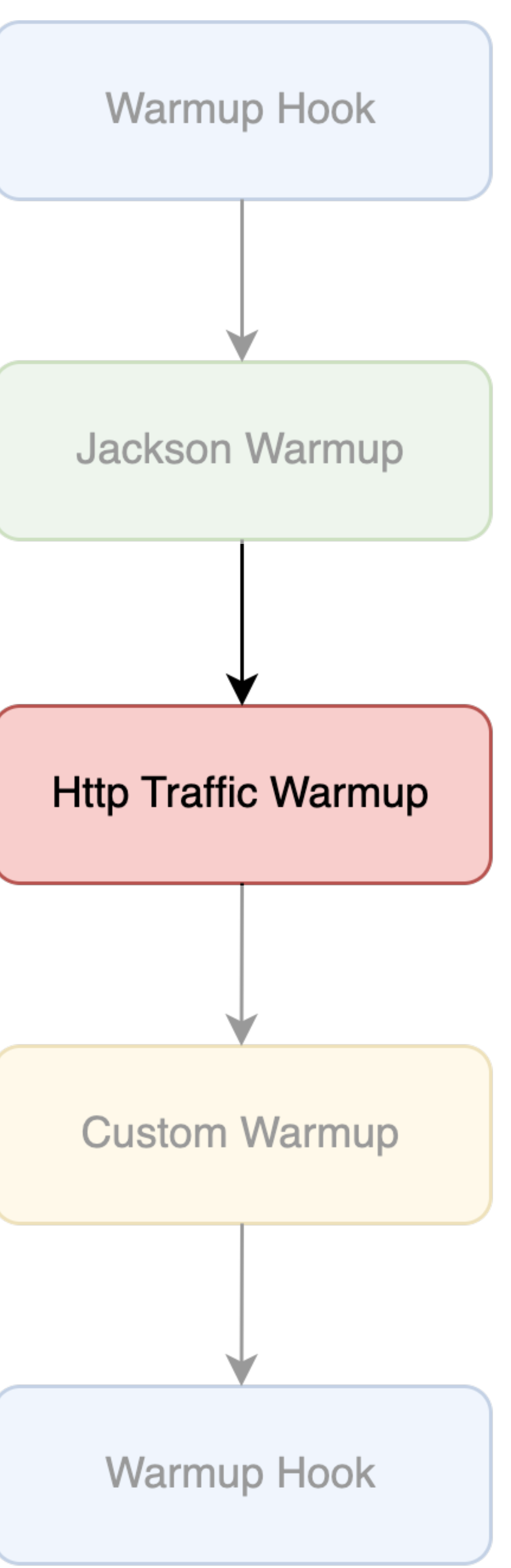


Прогрев Spring Controllers

Даже одна итерация прогрева значительно сокращает latency

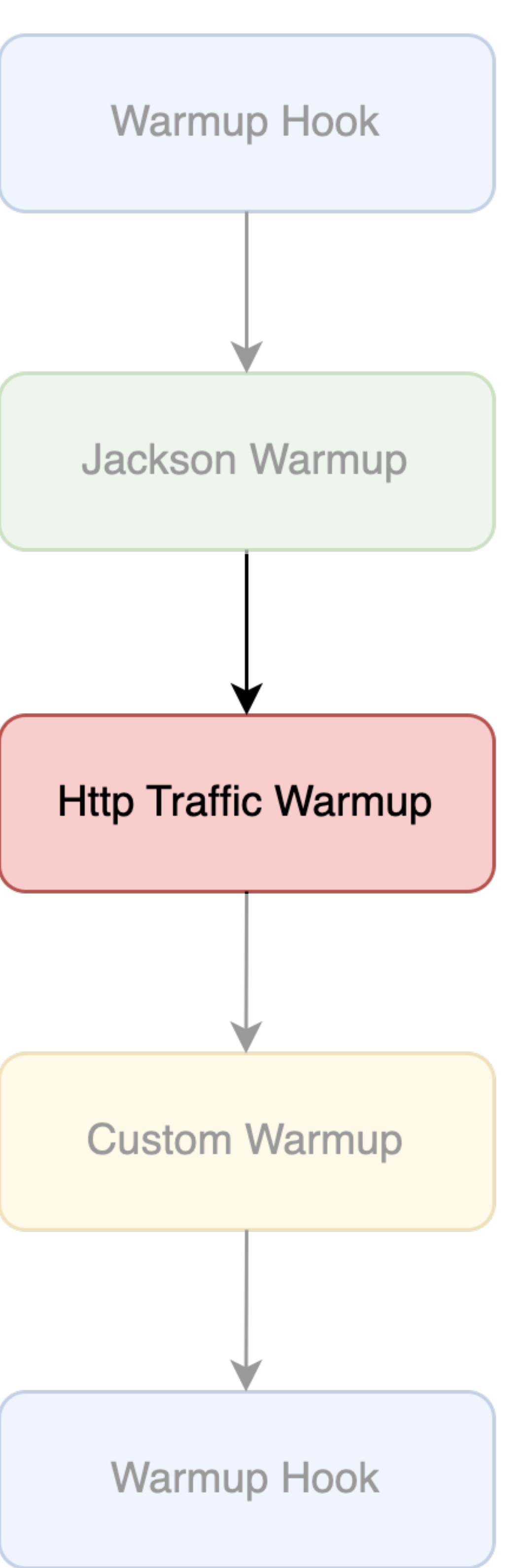
Прогрева нет	1 итерация прогрева	1k итераций прогрева
Среднее время выполнения:	Среднее время выполнения:	Среднее время выполнения:
498 ± 12,86 ms	15.8 ± 6.96 ms	10 ± 1.21 ms
95 Процентиль:	95 Процентиль:	95 Процентиль:
530 ms	20.3 ms	13.5 ms

Фазы Прогрева



Прогрев Spring Controllers

Фазы Прогрева



1k итераций сокращают latency еще на 30%!

Прогрева нет	1 итерация прогрева	1k итераций прогрева
Среднее время выполнения:	Среднее время выполнения:	Среднее время выполнения:
498 ± 12,86 ms	15.8 ± 6.96 ms	10 ± 1.21 ms
95 Процентиль:	95 Процентиль:	95 Процентиль:
530 ms	20.3 ms	13.5 ms

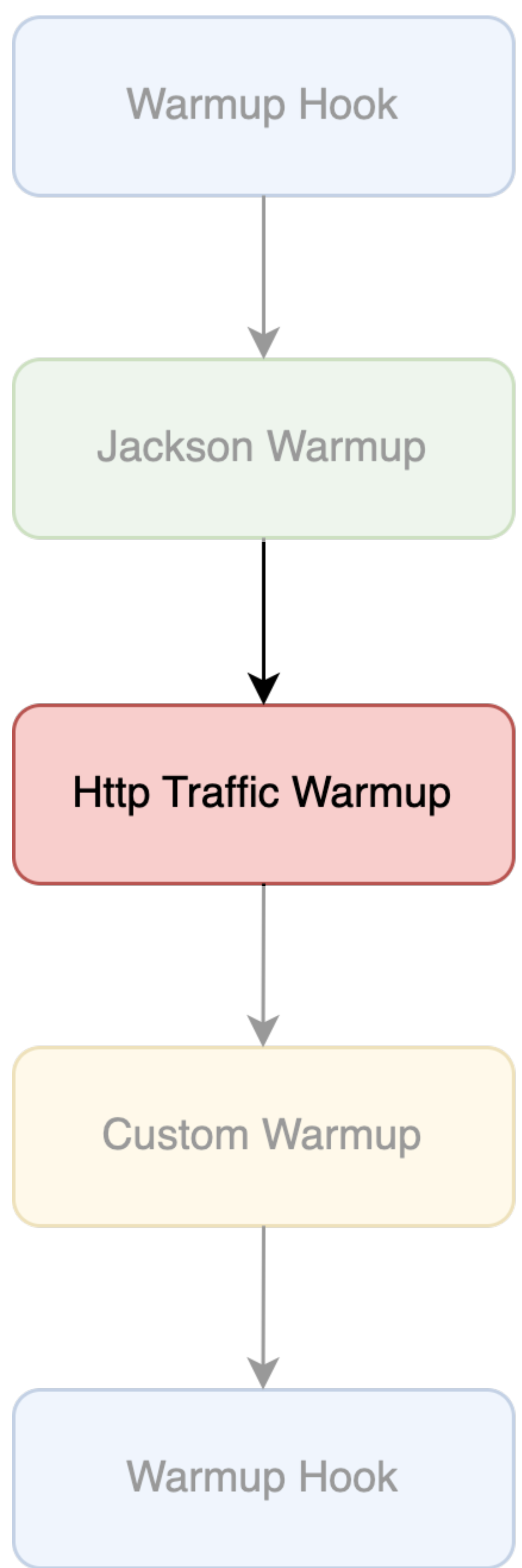
Прогрев Spring Controllers

А как же идемпотентность? С POST запросами проблема?

```yaml
top:
  warmup:
    http:
      enabled: true
      maximumWarmupTime: 20s  # задаем максимальное время прогрева
      endpoints:
        - path: /applications   # HTTP path метода регистрации заявок
          method: POST # Http method
          headers:
            Content-Type: application/json
          body-file: warmup/test-application.json
          warmup-iterations: 1
        - path: /applications/app-1  # HTTP path метода получения заявок
          method: GET # Http method
          warmup-iterations: 1
```

POST это проблема?

Фазы Прогрева



Прогрев Spring Controllers

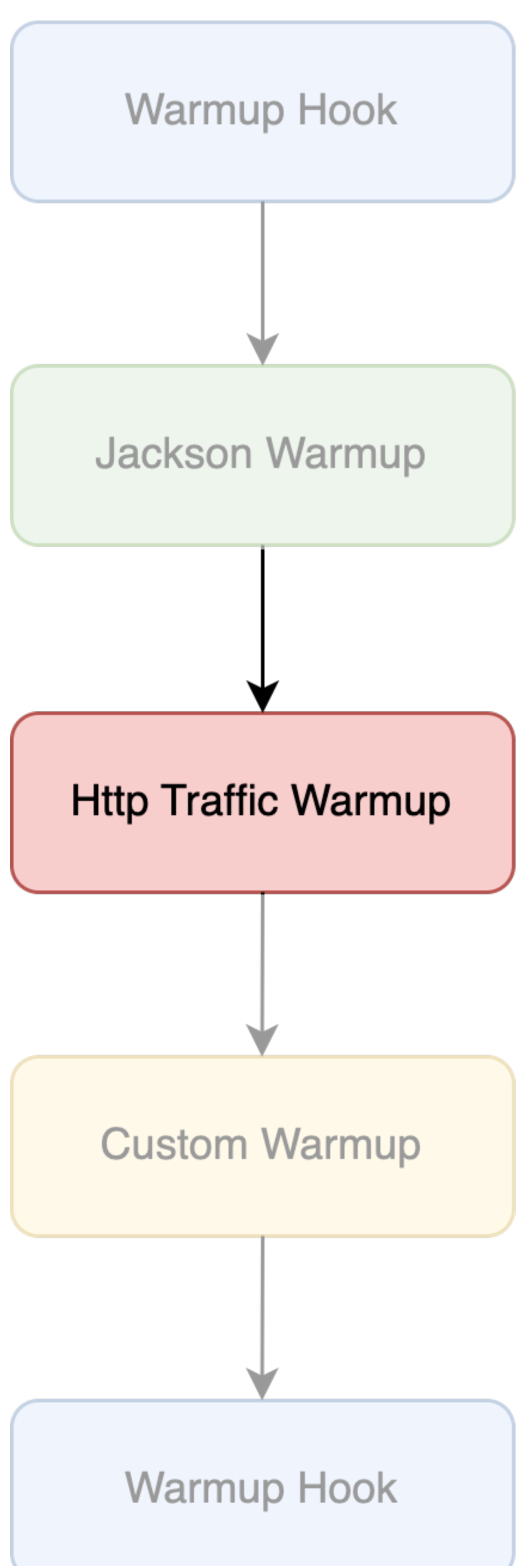
А как же идемпотентность? С POST запросами проблема?

```yaml
top:
  warmup:
    http:
      enabled: true
      maximumWarmupTime: 20s  # задаем максимальное время прогрева
      endpoints:
        - path: /applications   # HTTP path метода регистрации заявок
          method: POST # Http method
          headers:
            Content-Type: application/json
            Warmup: true # Добавим флаг, для переключения
          body-file: warmup/test-application.json
          warmup-iterations: 1
        - path: /applications/app-1  # HTTP path метода получения заявок
          method: GET # Http method
          warmup-iterations: 1
```

Да, но ее можно обойти

Перехватив флаг прогрева в кастомной логики

Фазы Прогрева



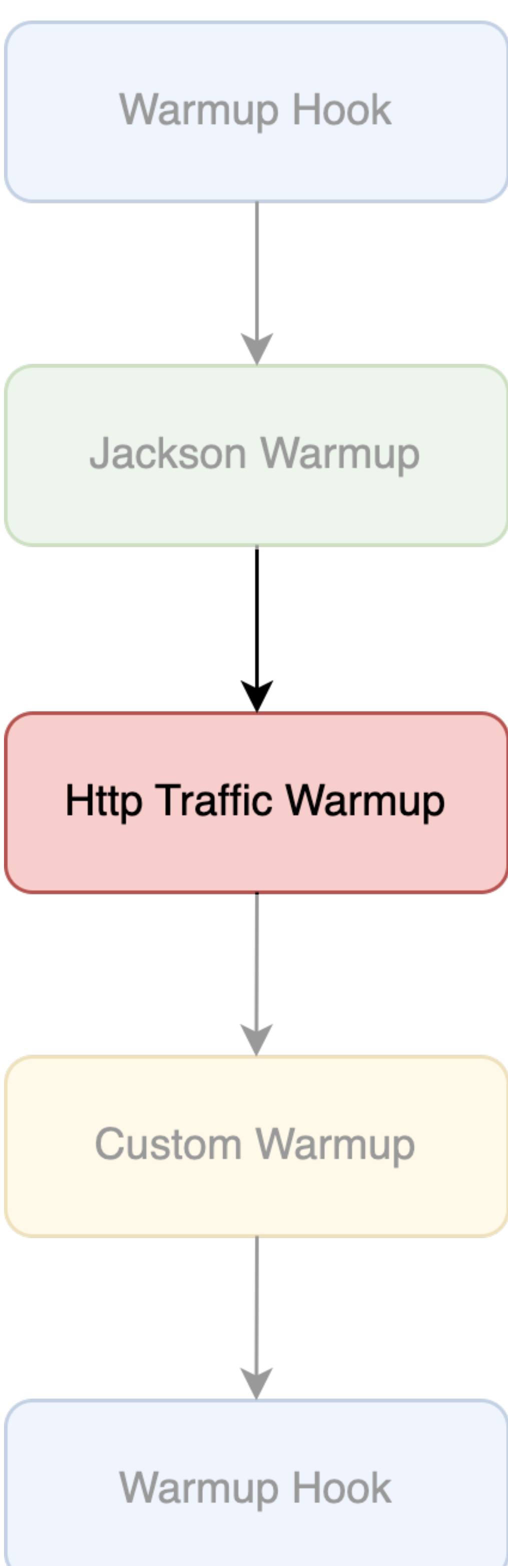
Прогрев Spring Controllers

А если я хочу греть не только happy path?

```yaml
top:
  warmup:
    http:
      enabled: true
      maximumWarmupTime: 20s  # задаем максимальное время прогрева
      endpoints:
        - path: /applications/app-1  # HTTP path метода получения заявок
          method: GET # Http method
          warmup-iterations: 1
          expectedCodes: # Можно указать список валидных response code для endpoint'a
            - 429
            - 404
            - 2xx
```

Можем указать маску либо список валидных Response Code

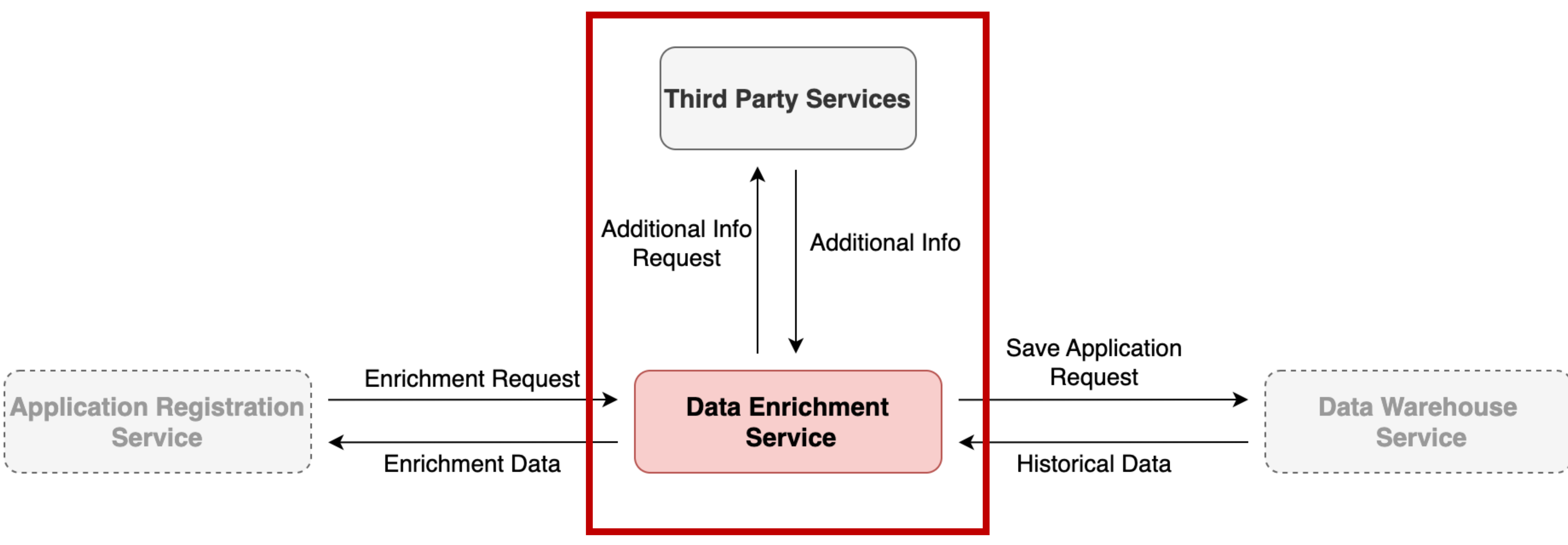
Фазы Прогрева



Пользовательский Прогрев

Сервис обогащения данных работает с большим кол-вом сторонних сервисов

Будем греть в нем Кеши



Пользовательский Прогрев

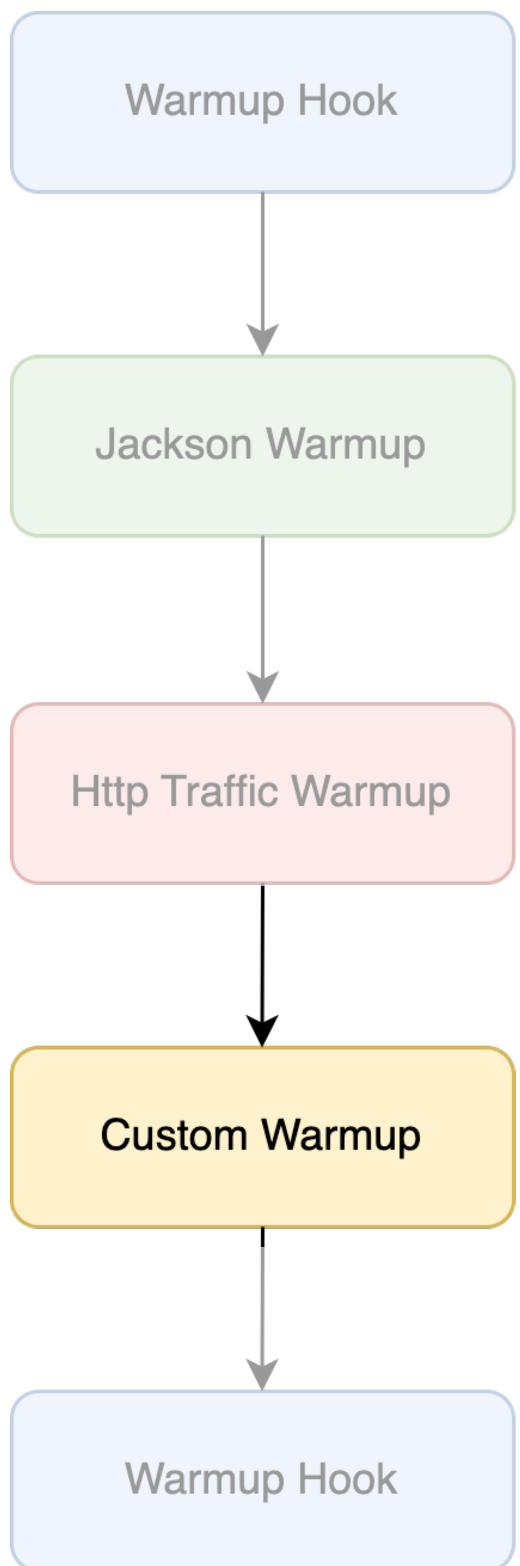
Пользователи могу реализовать свой Warmup Provider реализовав интерфейс:

```
interface WarmingUpProvider {

    val name: String

    fun getWarmingUpMethods(): List<() -> Unit>
}
```

Фазы Прогрева



Пользовательский Прогрев

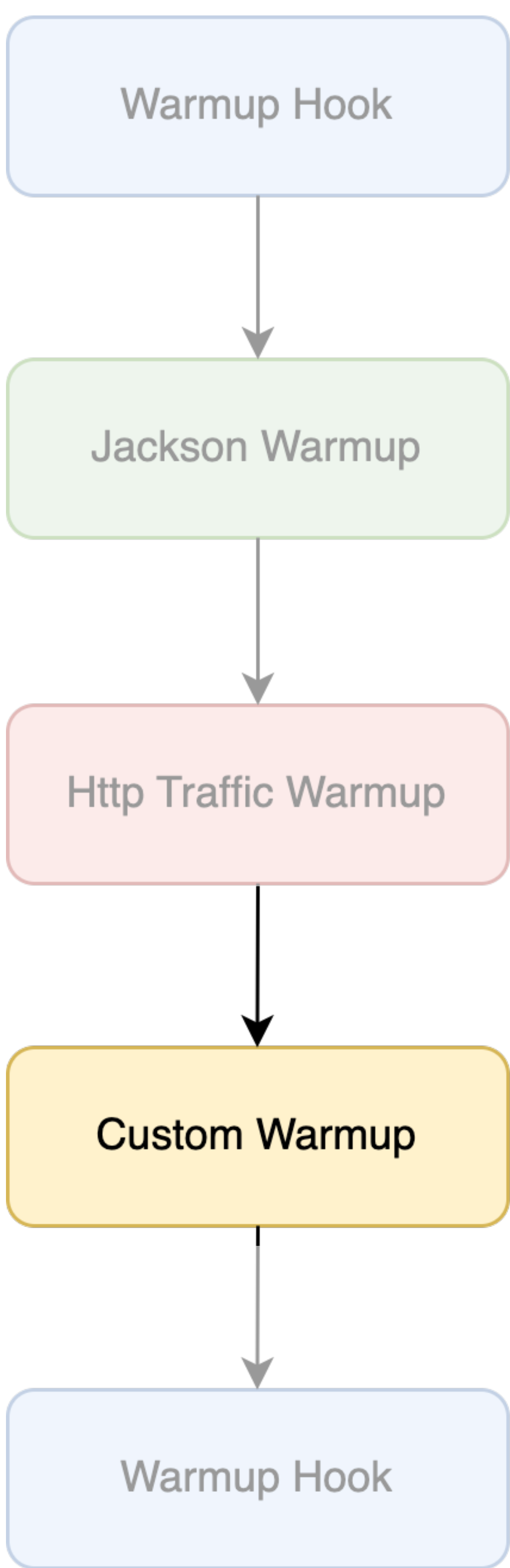
Простейшая реализация прогрева Кеша

```
@Component
class CacheManagerWarmingUpProvider(
  val cache: SimpleCache
) : WarmingUpProvider {
    override val name: String
        get() = "simpleCache"

    override fun getWarmingUpMethods(): List<() -> Unit> =
        listOf({
            getAllModelIds().forEach { id ->
              cache.put(id, loadVeryHugeModel(id))
            }
        })
}
```

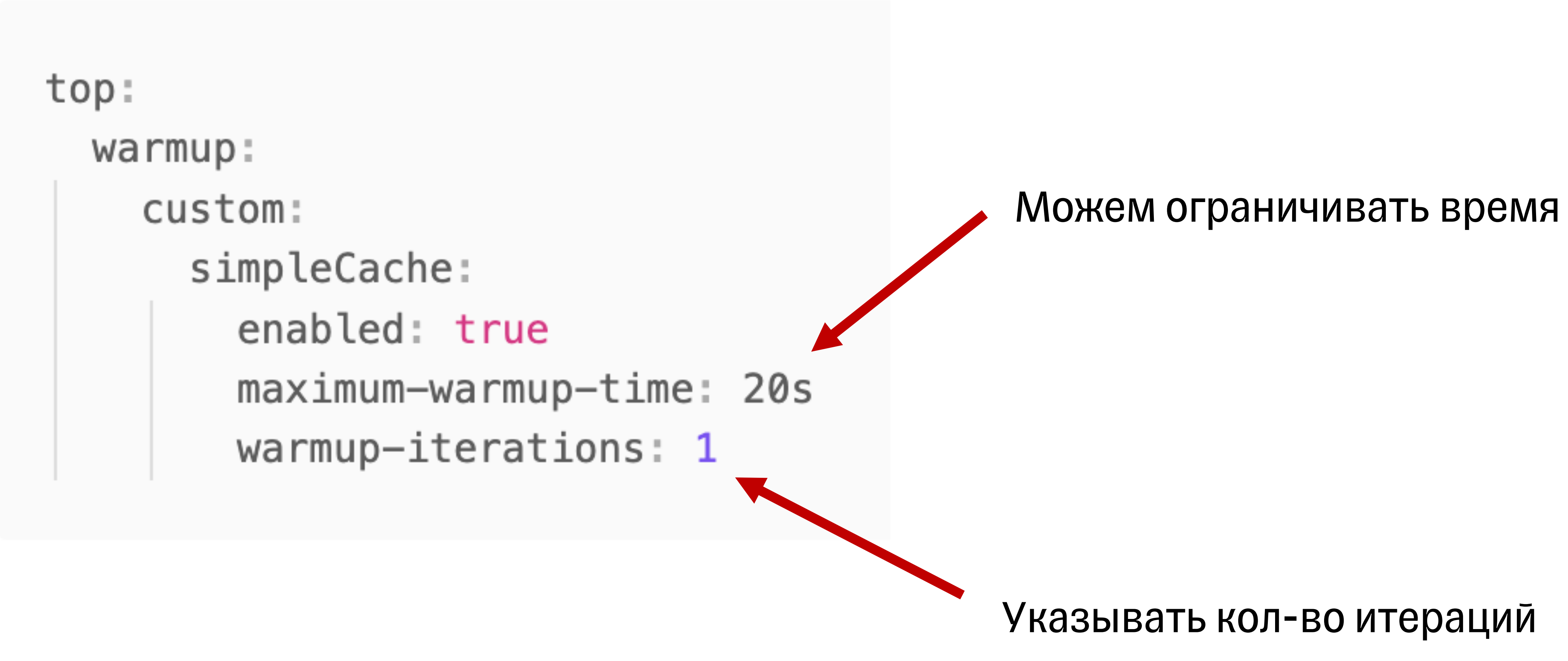
Будем ссылаться на имя в application.yaml

Фазы Прогрева



Пользовательский Прогрев

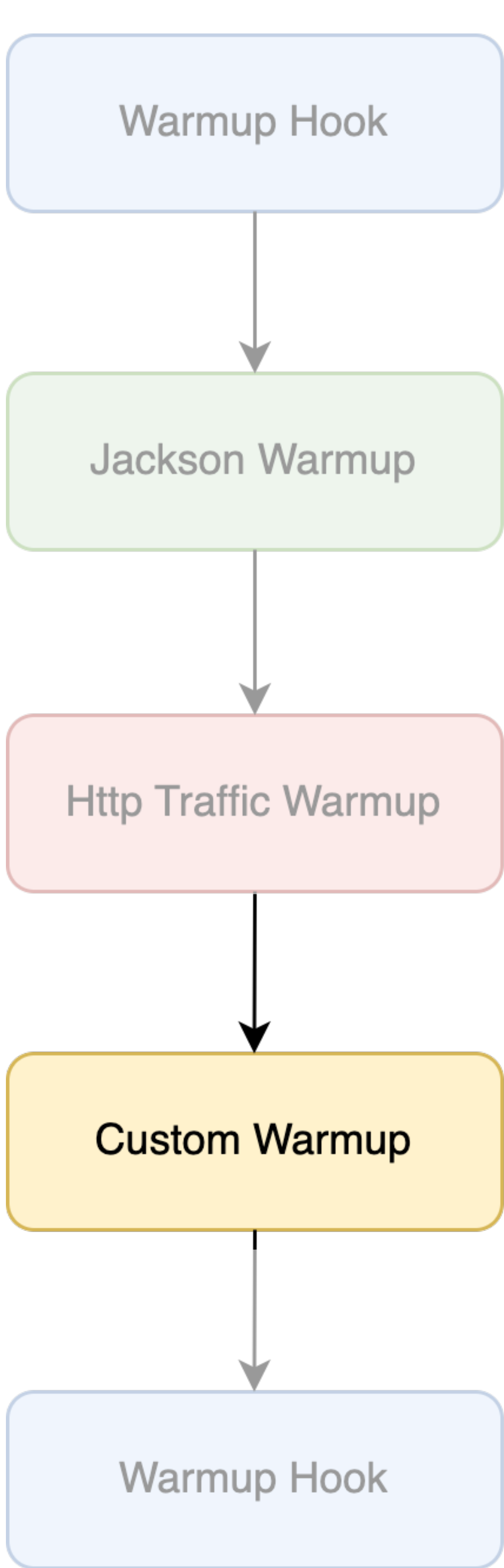
Простейшая конфигурация для пользовательского прогрева



Можем ограничивать время

Указывать кол-во итераций

Фазы Прогрева

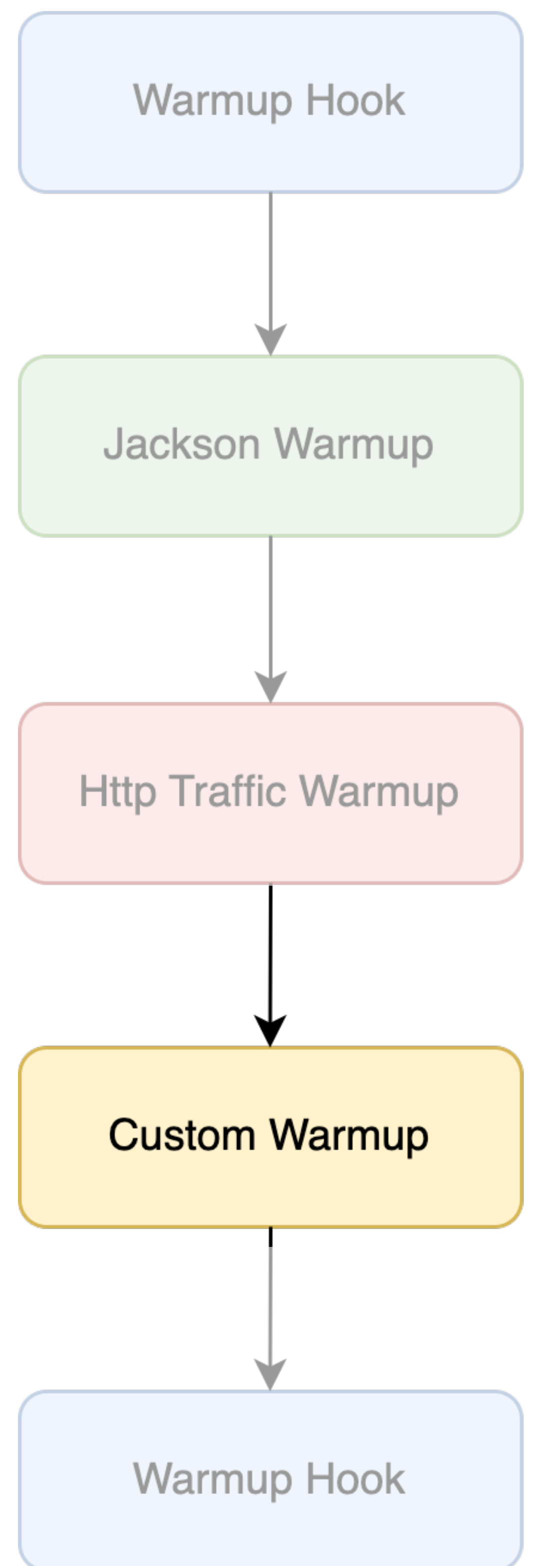


Пользовательский Прогрев

Зачем еще нужен пользовательский прогрев?

- Прогрев кэшей
- Открытие соединений к БД
- Необходимость авторизации во время запросов

Фазы Прогрева

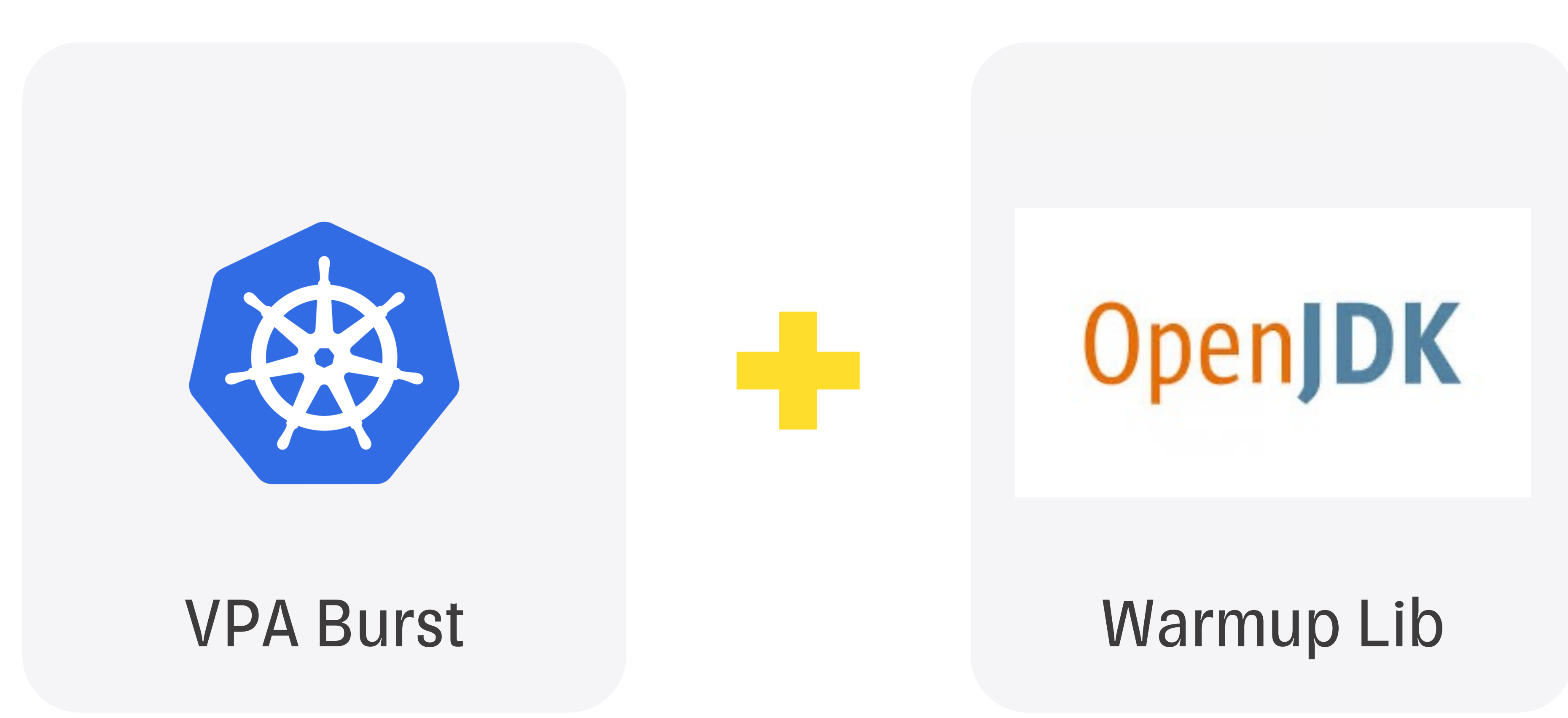


А что на проде?

Какие подходы мы используем на проде?

VPA burst позволяет сократить тротлинг отдельных подов сервиса на старте

Warmup Lib позволяет достичь стабильных показателей latency во время деплоя



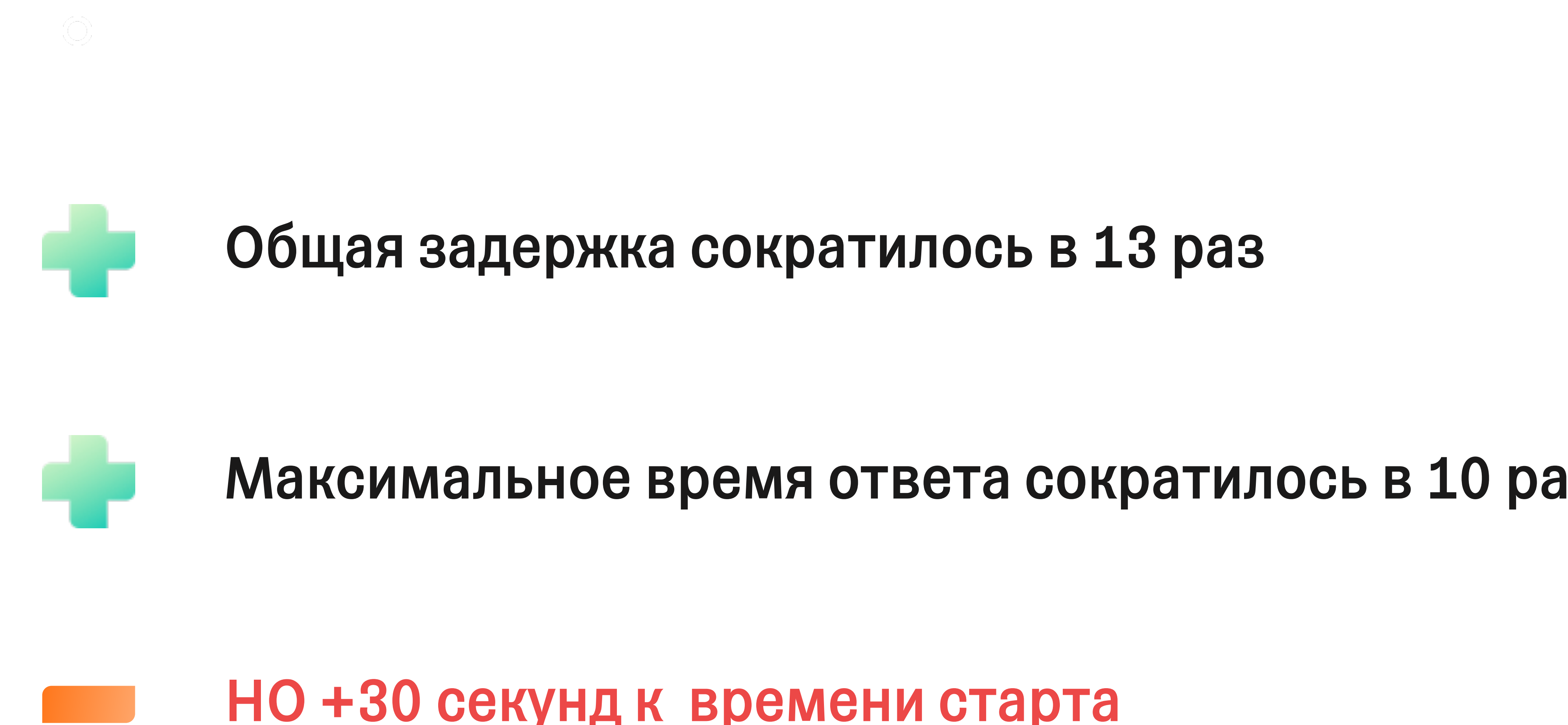
Влияние Прогрева на показатели Latency

Уменьшение максимальной задержки при старте приложения:

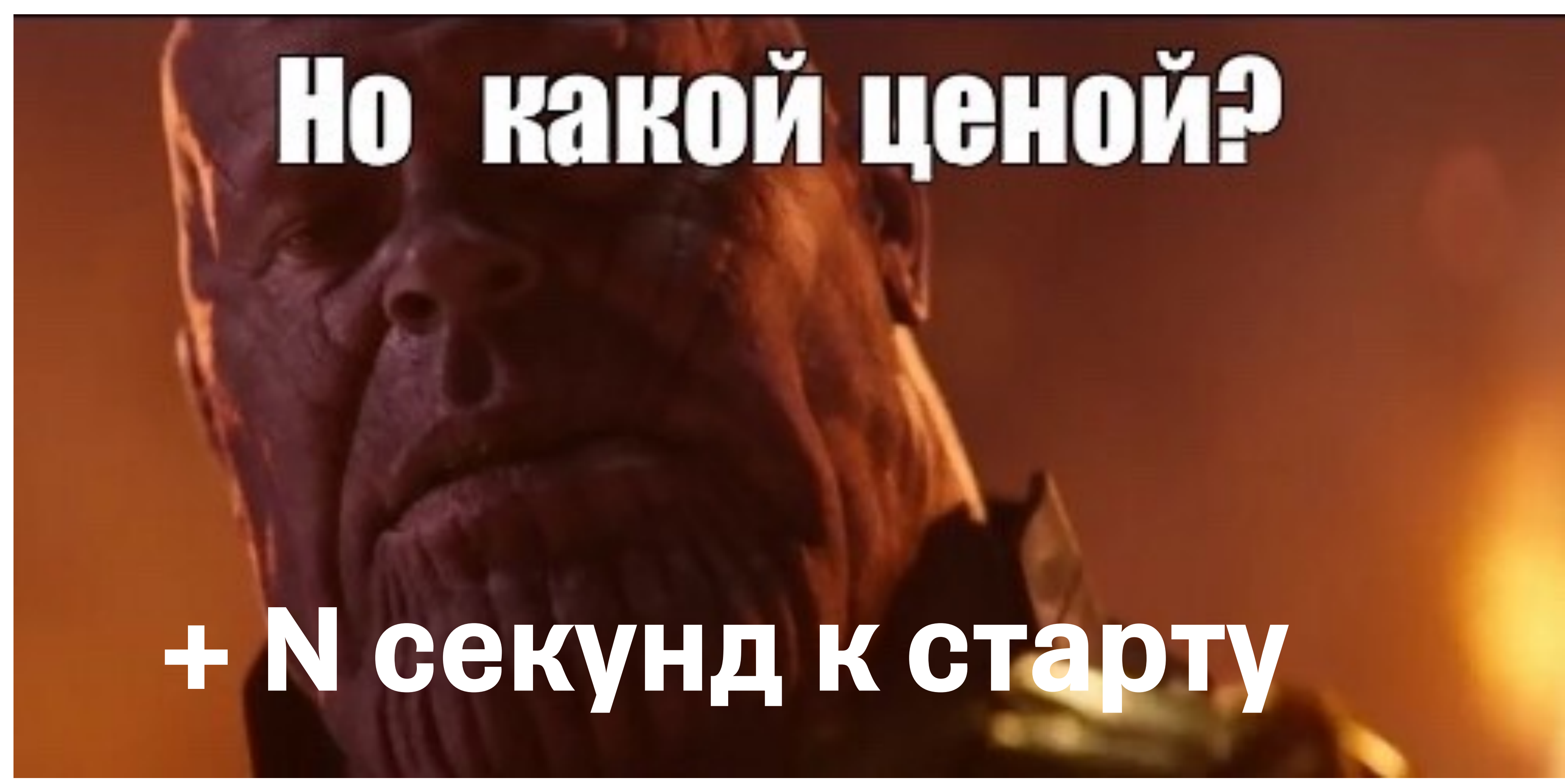
Влияние Прогрева на показатели Latency

P99 уменьшение средней задержки при старте приложения:

Что в итоге?



Проблему решили?



Проблемы решили, но время старта увеличили!

Резюмируем

Выделили плюсы каждого из методов

Снова не нашли ”Silver Bullet”

Сумели закрыть потребности своих команд

Open Source Party

Может быть вы хотите проверить сами?

Скоро на opensource.tbank.ru/top-core-libs/warmup/

Репозиторий:

Тестовый проект:

Вопросы?

Telegram:

Email: