Оптимизация работы потоков в 2k25:
почему у корутин получается лучше

Вячеслав Чернышов
СберТех

Какой стек основной на ваших проектах?

Блокирующий

Неблокирующий

Гибридный

Вячеслав Чернышов

Backend-разработчик
СберТех

Профиль на Хабре

Канал в Telegram

Я подниму 10 000 потоков и справлюсь с нагрузкой.

Зачем вообще реактивные потоки, когда есть нереактивные?

Я подниму 10 000 потоков и спі
нагрузкой.

Зачем вообще реак
есть нер

Интересно, какая операционная система сможет поддерживать 10 000 потоков и обрабатывать их на двух ядрах поды.

Но даже если и так, в чём проблема развернуть потоков сколько надо и закрыть проблему производительности железом?

Давайте посчитаем.

Из чего складывается размер оперативной памяти, потребляемой потоком?

Стековая память

32 bit: 320 KB
64 bit: 2 MB для Java 24

Из чего складывается размер оперативной памяти, потребляемой потоком?

Стековая память

32 bit: 320 KB
64 bit: 2 MB для Java 24

```
java -XX:+PrintFlagsFinal -version | grep ThreadStackSize
```

Из чего складывается размер оперативной памяти, потребляемой потоком?

Стековая память

32 bit: 320 KB
64 bit: 2 MB для Java 24

```
% java -XX:+PrintFlagsFinal -version | grep ThreadStackSize
     intx CompilerThreadStackSize                  = 2048                  {pd product} {default}
     intx ThreadStackSize                          = 2048                  {pd product} {default}
     intx VMThreadStackSize                        = 2048                  {pd product} {default}
openjdk version "24.0.1" 2025-04-15
OpenJDK Runtime Environment (build 24.0.1+9-30)
OpenJDK 64-Bit Server VM (build 24.0.1+9-30, mixed mode, sharing)
```

Из чего складывается размер оперативной памяти, потребляемой потоком?

Стековая память

32 bit: 320 KB
64 bit: 2 MB для Java 24

В Java 8 память, зарезервированная для стеков потоков, реально выделялась.

В Java 11 она уже не выделяется, а только резервируется.

Но для высоконагруженных систем, в которых потоки постоянно работают, это не так важно.

Из чего складывается размер оперативной памяти, потребляемой потоком?

Стековая память	32 bit: 320 KB 64 bit: 2 MB для Java 24
Нативные ресурсы ОС	~ 2-4 KB
Хранение состояния	~ 1-2 KB
Total	2.005 MB для Java 24

1 000 потоков потребует 2 ГБ RAM.

10 000 потоков потребует 20 ГБ RAM...

А сколько вообще потоков требует система? Может, нет поводов для паники?

Плановая нагрузка на разные типы сервисов *

Время выполнения запроса 150 ms

	RPS	Сколько потоков потребуется	Сколько RAM они займут
Один известный поисковик	30-60 K	4.5 - 9 K	9 – 18 GB
Один известный видеохостинг	1-2 M	150 – 300 K	300 – 600 GB
Одна известная социальная сеть	2-5 M	300 – 750 K	600 GB – 1.5 TB
Один известный интернет-магазин	5-10 M	750 K – 1.5 M	1.5 – 3 TB
Один известный мессенджер	10-20 M	1.5 - 3 M	3 – 6 TB
SaaS-стартап	100 – 10 K	15 – 1 500	30 MB – 3 GB
Банковское приложение	1 K – 50 K	150 – 7 500	300 MB – 15 GB
Криптобиржа	100 K – 1 M	15 – 150 K	30 – 300 GB

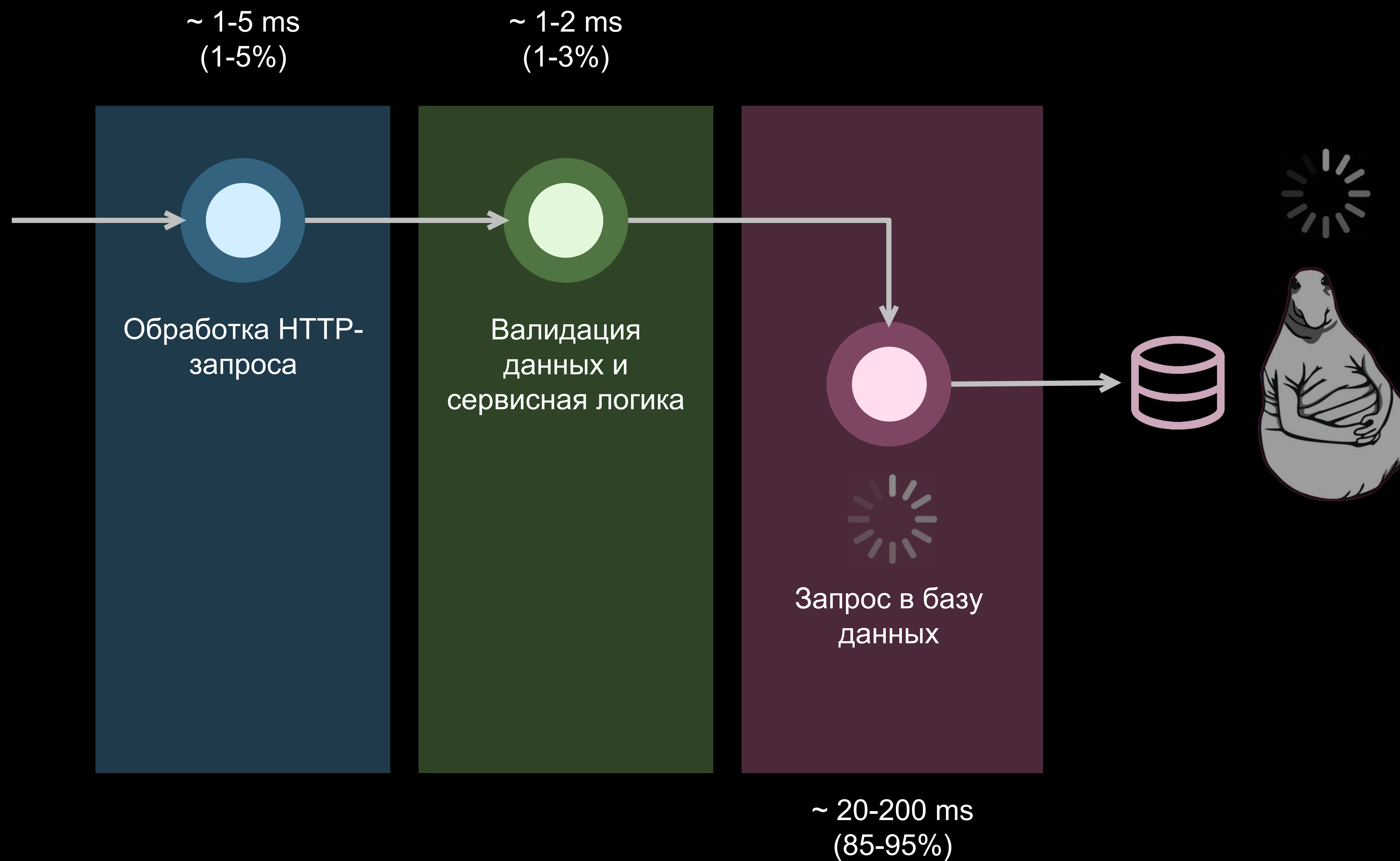
* примерные данные из открытых источников

Да, мы понимаем, что ни один сервис не потянет такого количества потоков.

Сервисы как-то выкручиваются.

Как?

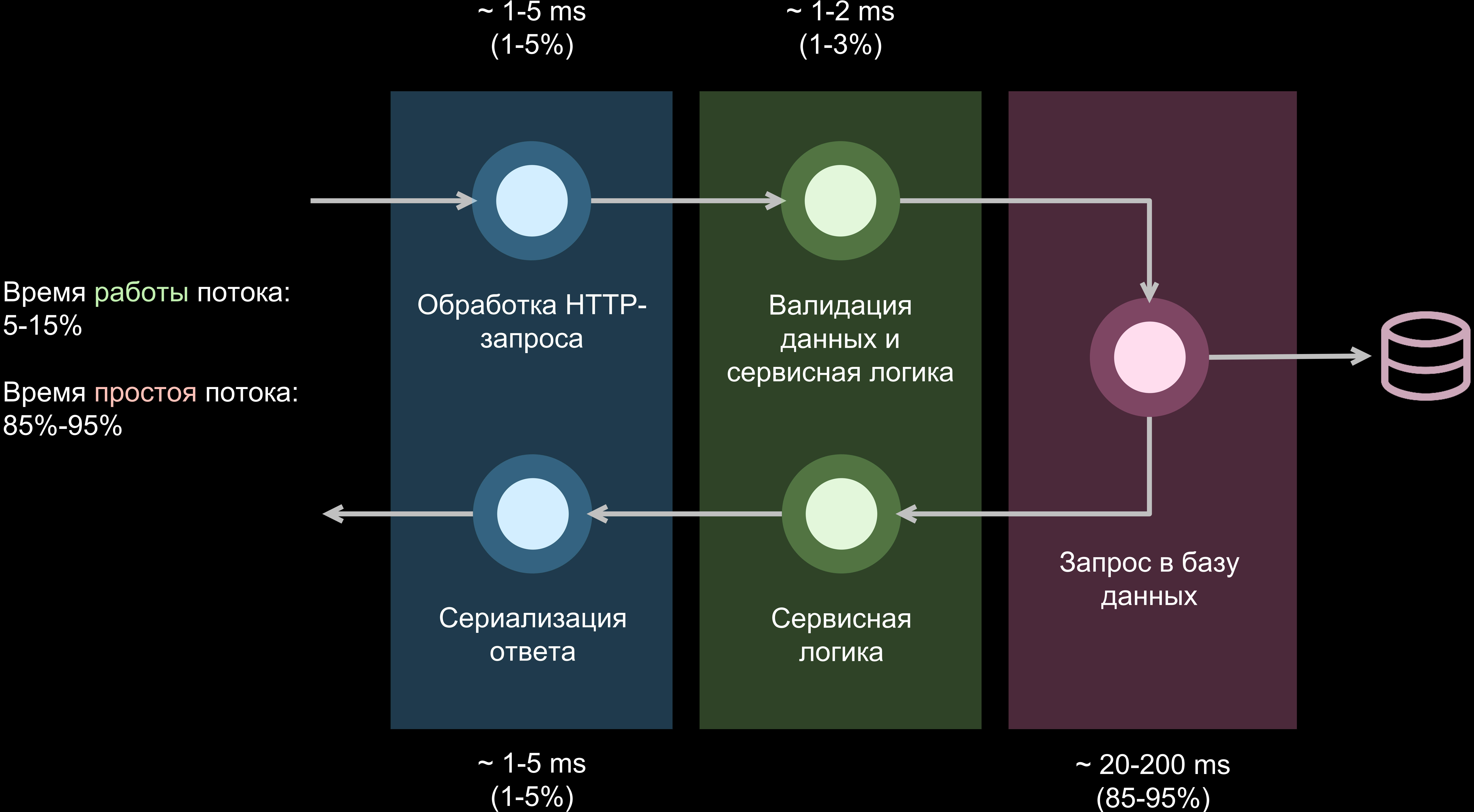
Время выполнения запроса блокирующимся потоком



Время выполнения запроса блокирующимся потоком

Время работы потока:
5-15%

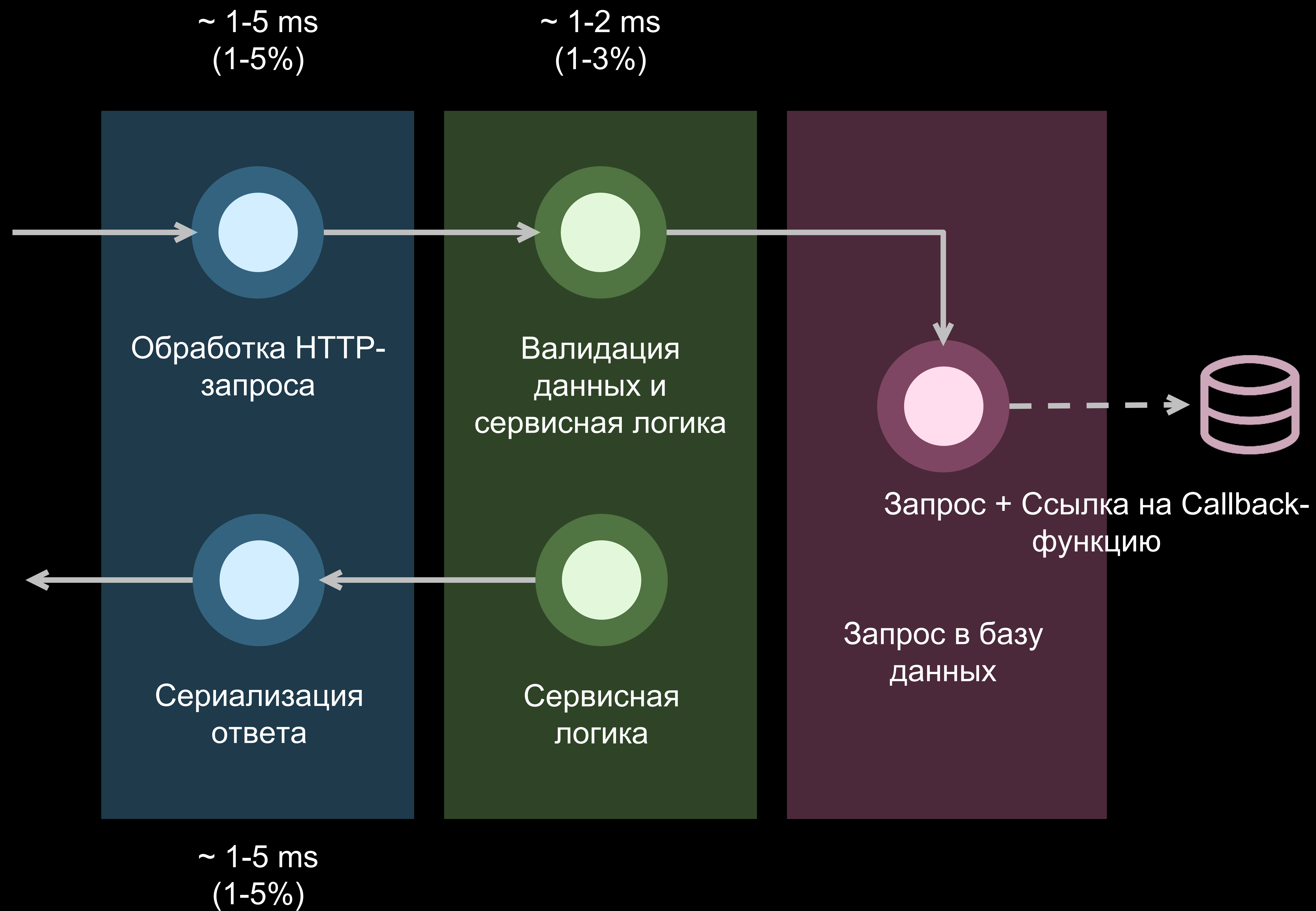
Время простоя потока:
85%-95%



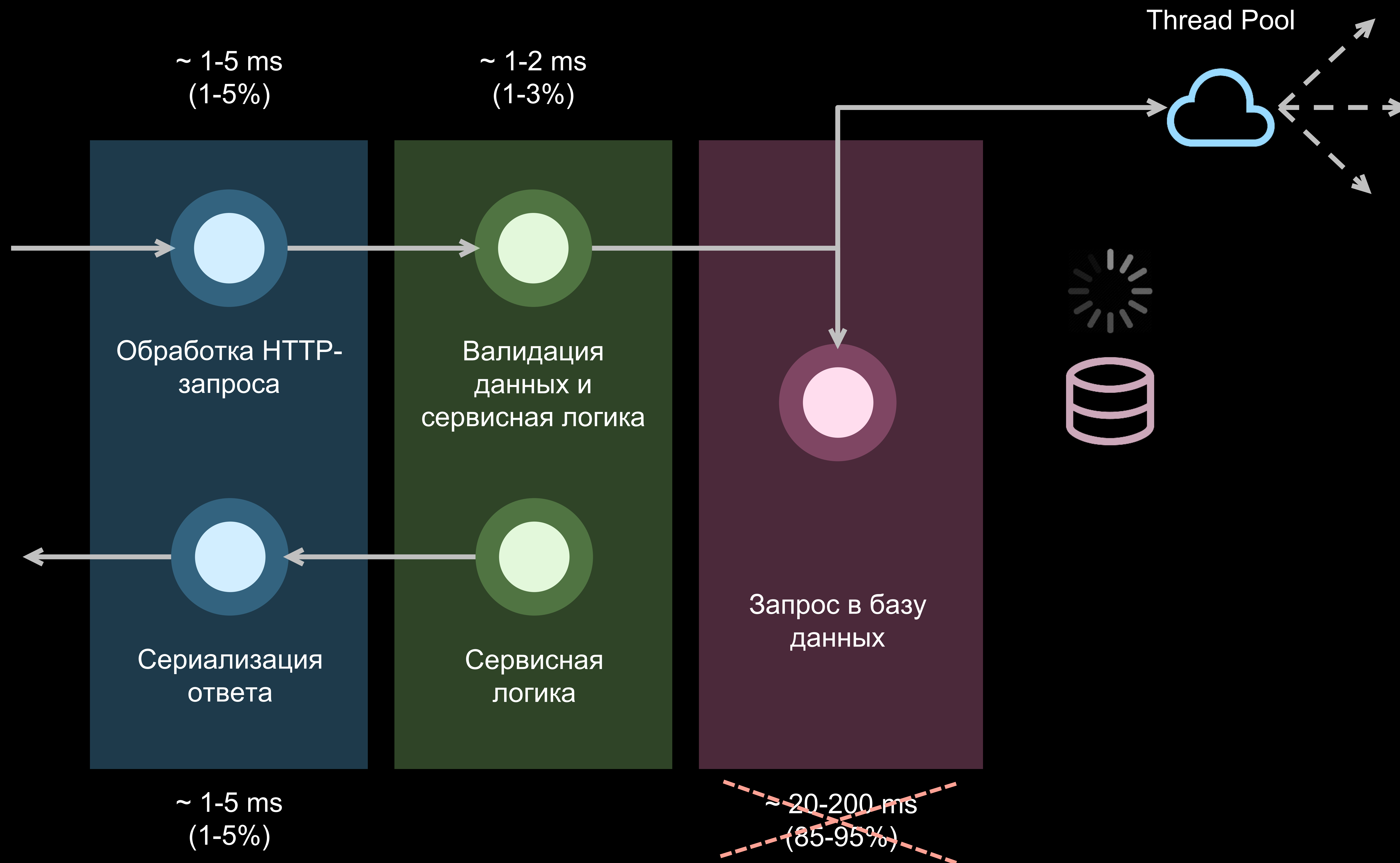
Кажется, что потоки едят свою оперативку незаслуженно.

Решение: высвободить поток на время ожидания операций ввода / вывода, чтобы он мог выполнять другие задачи и заслуженно есть свою оперативку.

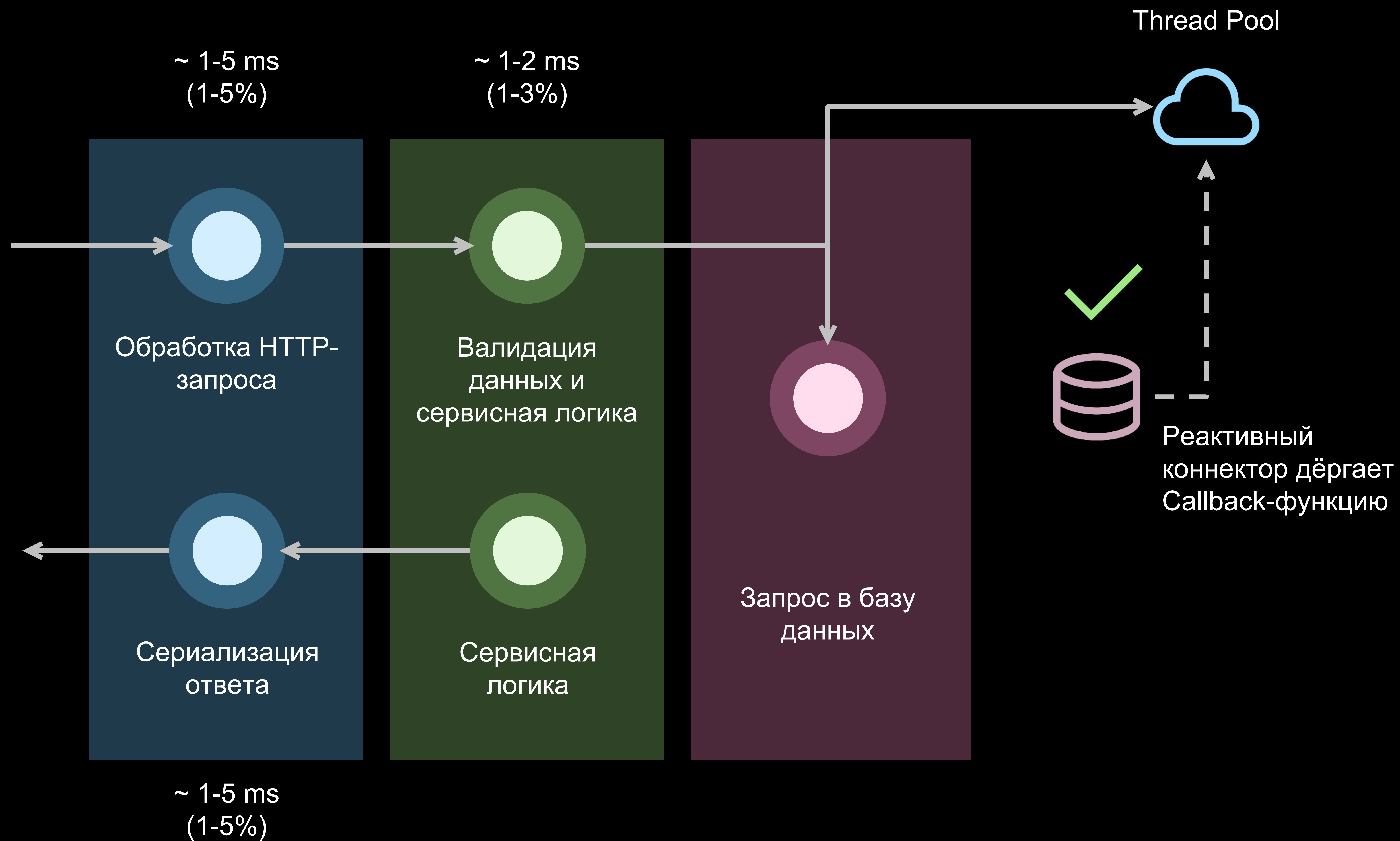
Время выполнения запроса реактивным потоком



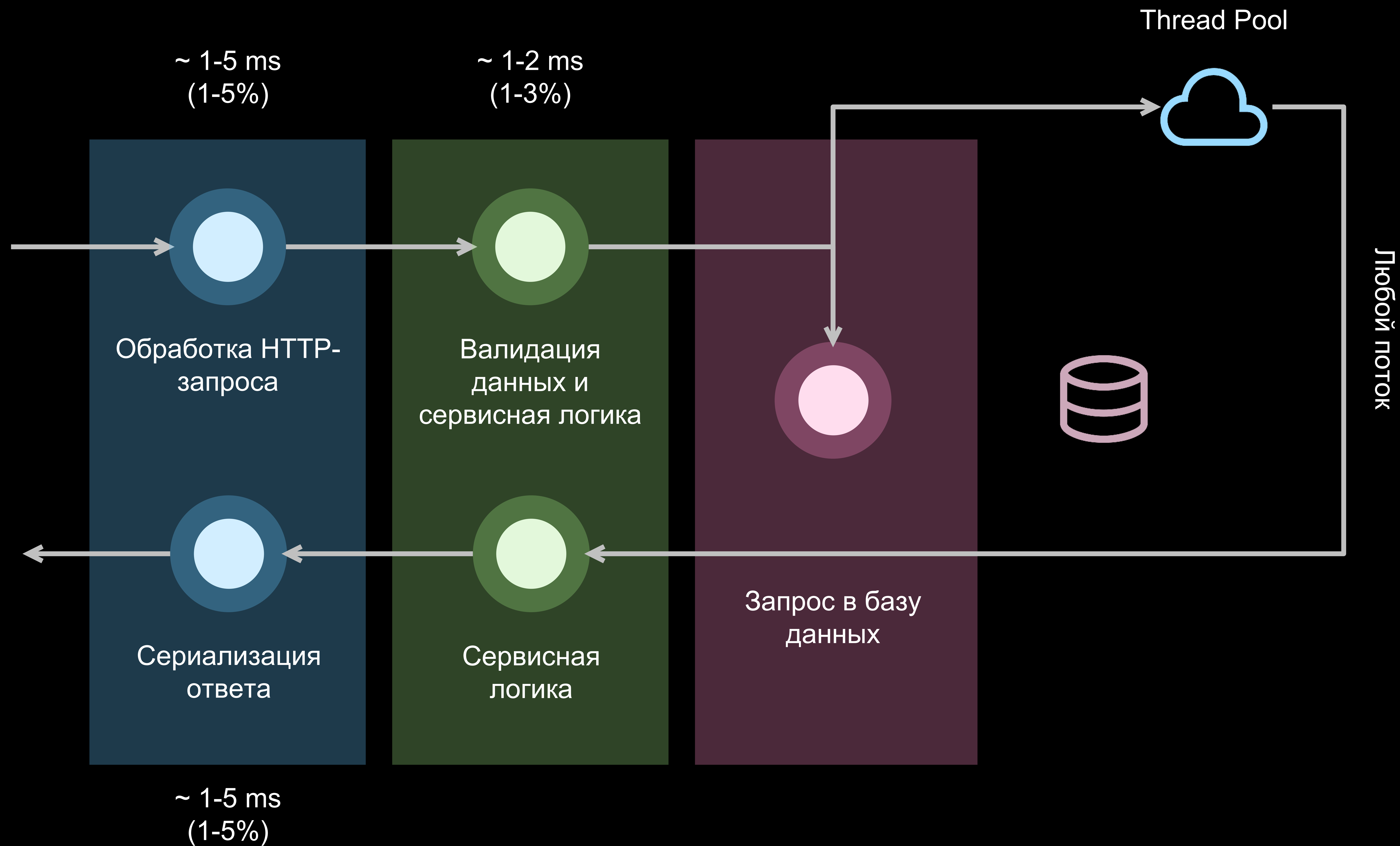
Время выполнения запроса реактивным потоком



Время выполнения запроса реактивным потоком



Время выполнения запроса реактивным потоком



Реактивные потоки едят свою оперативку заслуженно.

Плановая нагрузка на разные типы сервисов

Время выполнения запроса	Блокирующий стек: 150 ms			Реактивный стек: 10 ms	
	Кол-во потоков	RAM		Кол-во потоков	RAM
Один известный поисковик	4.5 - 9 K	9 – 18 GB	~	300 - 600	0.6 – 1.2 GB
Один известный видеохостинг	150 – 300 K	300 – 600 GB	~	10 – 20 K	20 - 40 GB
Одна известная социальная сеть	300 – 750 K	600 GB – 1.5 TB	~	20 – 50 K	40 – 100 GB
Один известный интернет-магазин	750 K – 1.5 M	1.5 – 3 TB	~	50 – 100 K	100 – 200 GB
Один известный мессенджер	1.5 - 3 M	3 – 6 TB	~	100 – 200 K	200 – 400 GB
SaaS-стартап	15 – 1 500	30 MB – 3 GB	~	15 – 1 500	2 – 200 MB
Банковское приложение	150 – 7 500	300 MB – 15 GB	~	150 – 7 500	200 MB - 1 GB
Криптобиржа	15 – 150 K	30 – 300 GB	~	15 – 150 K	2 – 20 GB

Я подниму 10 000 потоков и справлюсь с нагрузкой.

Мои 600 реактивных потоков будут выполнять работу твоих 10 000.

И их даже, возможно, потянет операционная система.

Но не факт.

Мои 600 реактивных потоков будут выполнять работу твоих 10 000.

Реактивные библиотеки позволяют работать с I/O сервисами без блокировки потоков.

Но мы, разработчики, крайне неохотно работаем с реактивными библиотеками.

И нас можно понять. Реактивная разработка может стать сущим адом.

Реактивные потоки. Проблемы.

Сложность кода и отладки.

Callback Hell / Pyramid of Doom

В реактивном программировании код строится в виде цепочек операторов (map, flatMap, filter, zip). Если цепочка становится слишком длинной, код превращается в «лапшу».

Проблемы:

Глубокий уровень вложенности.
Сложности с отслеживанием ошибок.
Теряется линейность кода.

Размытие стека вызовов.

Стек вызовов управляется сложными цепочками операторов вроде flatMap, retryWhen, onErrorResume.

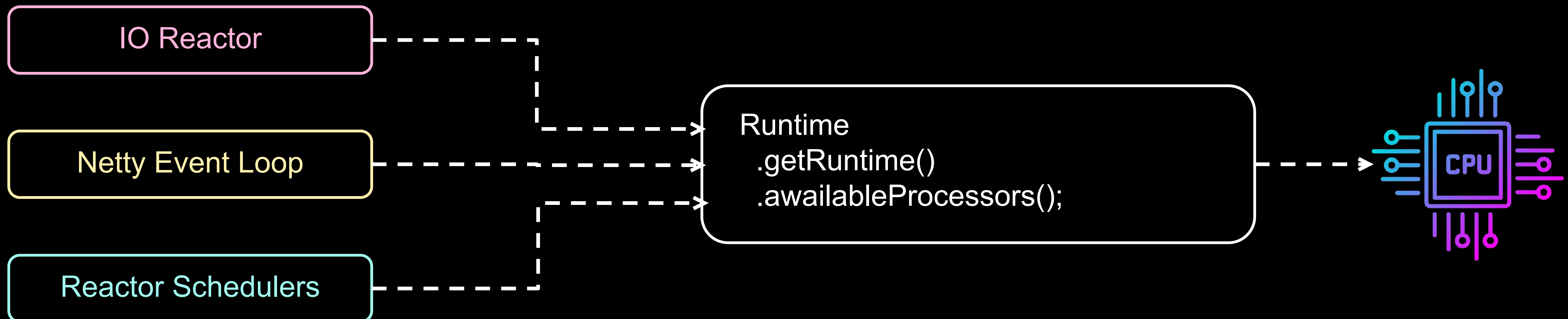
Реактивные потоки. Проблемы.

Сложность кода и отладки.

Проблемы с блокирующим кодом.

Реактивность работает на ограниченном количестве потоков

В Spring WebFlux по умолчанию используется количество потоков, равное количеству ядер. При использовании частой практики «несколько 1-2 ядерных pods» блокирование потока полностью остановит приложение.



Реактивные потоки. Проблемы.

Сложность кода и отладки.

Проблемы с блокирующим кодом.

Реактивность работает на ограниченном количестве потоков

В Spring WebFlux по умолчанию используется количество потоков, равное количеству ядер. При использовании частой практики «несколько 1-2 ядерных pods» блокирование потока полностью остановит приложение.

Риск deadlock-ов (как следствие)

Ситуация возникает, когда мы блокируем единственный доступный поток в реактивном пуле, и это предотвращает выполнение других задач, включая ту, которая должна была разблокировать первую.

Реактивные потоки. Проблемы.

Сложность кода и отладки.

Проблемы с блокирующим кодом.

Неоптимальная утилизация памяти.

Создание множества объектов

При вызове операторов map, flatMap и прочих создаются объекты-подписки. Так, цепочка Flux.map().filter().flatMap() создаст 3 объекта-подписки.

Нет встроенной отмены дочерних операций при ошибке.

Subscriber управляет потоком данных через Subscription. Ошибка в одном из операторов приводит к вызову onError у подписчика, но не отменяет другие асинхронные операции, которые уже были запущены.

Обычные блокирующие потоки

Написаны в привычном императивном стиле

Блокируются при I/O операциях

Масштабируемость ограничена числом потоков

Сложность в параллельной разработке

Реактивные потоки

Не блокируются и тем самым экономят ресурсы

Написаны в сложно понимаемом стиле цепочек вызовов

Проблемы с блокирующим кодом

Неоптимально утилизируют память

Эти потоки – физические!

Сотни потоков даже при реактивном подходе на двух ядрах поды убьют всю производительность!

Обычные блокирующие потоки

Написаны в привычном императивном стиле

Блокируются при I/O операциях

Масштабируемость ограничена числом потоков

Сложность в параллельной разработке

Реактивные потоки

Не блокируются и тем самым экономят ресурсы

Написаны в сложно понимаемом стиле цепочек вызовов

Проблемы с блокирующим кодом

Неоптимально утилизируют память

Вывод: потоки должны быть виртуальными.

Корутины Kotlin и виртуальные потоки Java концептуально схожи, поскольку легковесны, отделены от физических потоков и очень эффективны при выполнении операций ввода-вывода.

Coroutine: что это?

Coroutine — это объект, управляемый рантаймом,
а не Планировщиком ОС.

Coroutine — это объект.

В отличие от потока, который управляется
Планировщиком ОС, корутина управляется
рантаймом, как и любой объект в JVM.

Coroutine: что это?

Coroutine — это объект, управляемый рантаймом, а не Планировщиком ОС.

Coroutine выполняются последовательно.

Корутины не требуют подписок и реактивных цепочек. Код написанный на корутинах — это обычный императивный код, который мы пишем в не-реактивных приложениях.

Это даёт нам понятную последовательную логику в коде и избавляет от Callback Hell. Подписок и реактивных цепочек просто нет.

Coroutine: что это?

Coroutine — это объект, управляемый рантаймом, а не Планировщиком ОС.

Coroutine выполняются последовательно.

Coroutines используют кооперативную многозадачность.

Корутины подчинены кооперативной многозадачности. Это значит, что корутина выполняется ровно до того момента, когда разрабочик или JVM решили её приостановить. Это полностью контролируемый со стороны разработчика / JVM процесс.

В отличие от физических потоков, которые подчинены вытесняющей многозадачности и прерываются произвольно.

Coroutine: что это?

Coroutine — это объект, управляемый рантаймом, а не Планировщиком ОС.

Coroutine выполняются последовательно.

Coroutines используют кооперативную многозадачность.

Для выполнения своих задач coroutines используют потоки.

Для выполнения задач корутины используют потоки.

Когда корутина получает управление над задачей, ей выделяется поток. Она использует поток для решения своих задач, пока не достигает точки приостановки, после чего на поток «пересаживается» следующая корутина из очереди.

Важно:

Потоки не управляют данными. Потоки дают корутинам попользоваться своим стеком. Корутины используют стек, чтобы сохранить туда свои данные, пока они «бегут». Как только корутина приостанавливается, она сохраняет данные в Continuation и возвращает поток в Thread Pool. Стек при этом очищается.

Coroutine: что это?

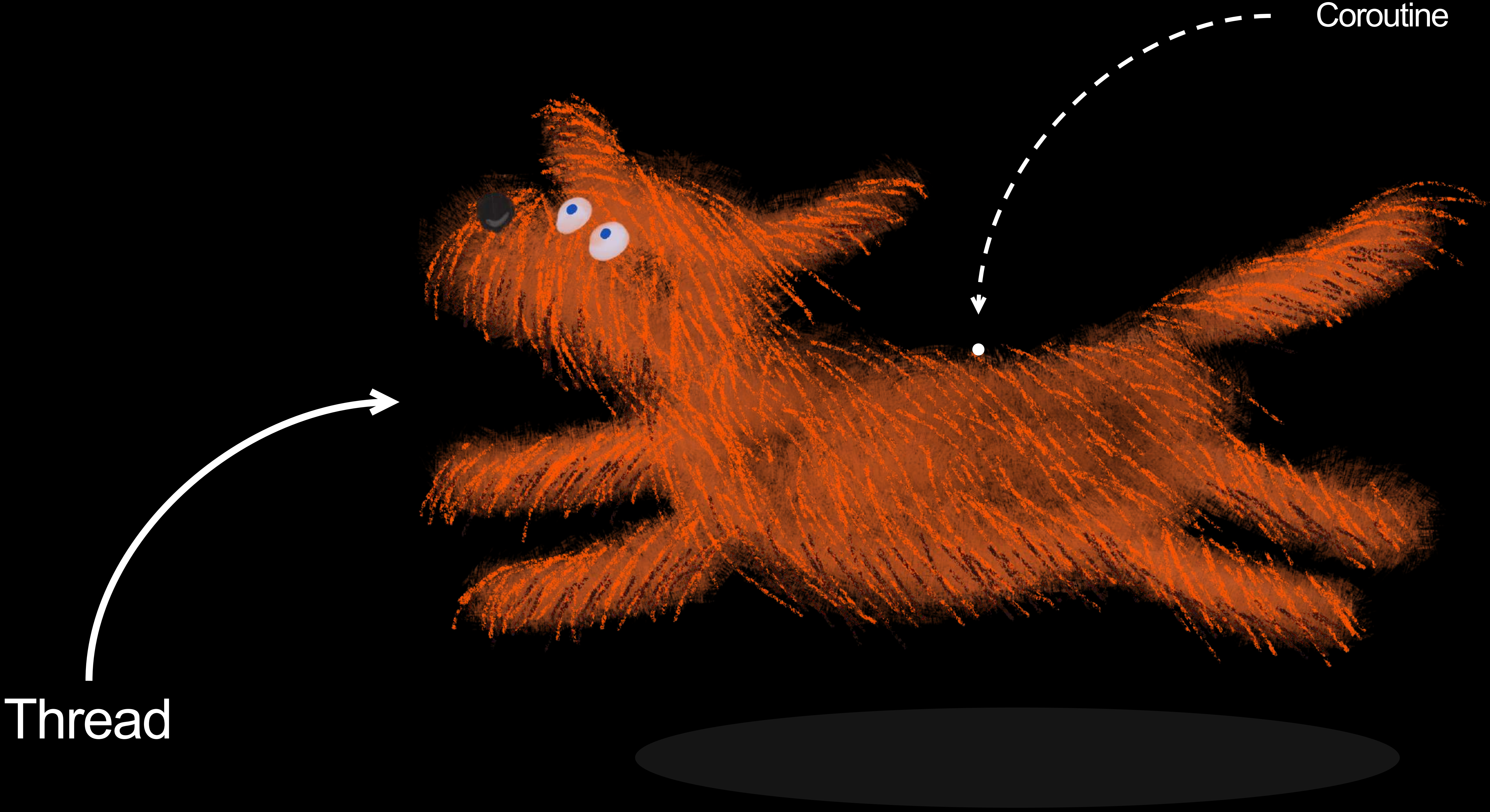
Coroutine — это объект, управляемый рантаймом, а не Планировщиком ОС.

Coroutine выполняются последовательно.

Coroutines используют кооперативную многозадачность.

Для выполнения своих задач coroutines используют потоки.

Учитывая очень маленькие размеры корутин относительно потоков, будет уместно сравнение блохи (корутины) и собаки (потока).


Coroutine
Thread

Coroutine: что это?

Coroutine — это объект, управляемый рантаймом, а не Планировщиком ОС.

Coroutine выполняются последовательно.

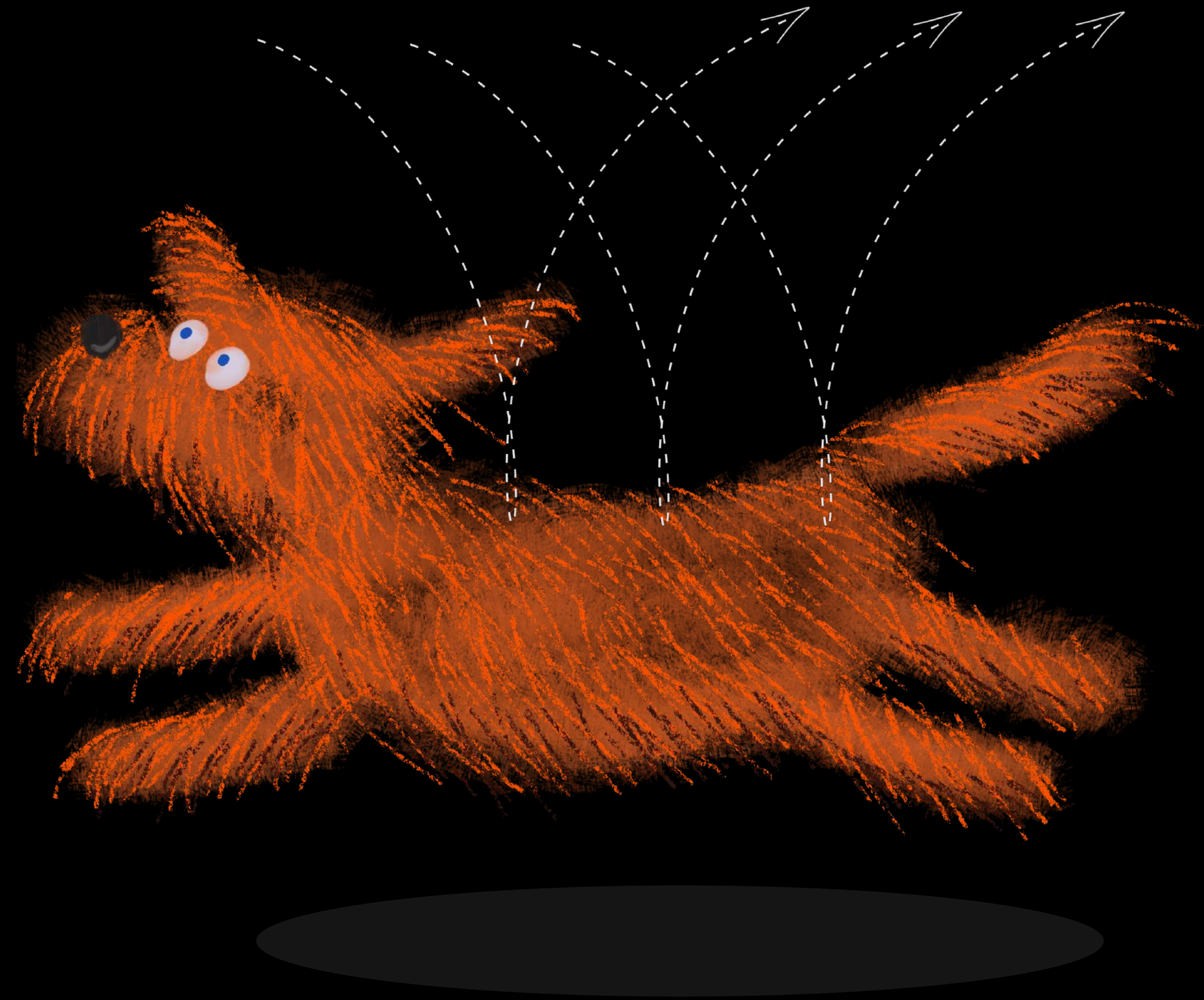
Coroutines используют кооперативную многозадачность.

Для выполнения своих задач coroutines используют потоки.

Coroutines просто берут следующий поток и «едут» на нём до следующей точки переключения. Поток не знает, какую бизнес-операцию он выполняет. Он просто бежит и всё.

При этом, существуют некоторые правила:

- На собаке может находиться только одна блоха.
- Собака не знает, какое задание она выполняет. Она просто бежит туда, куда ей укажет блоха.
- Блохи конкурируют за собак.



Корутины конкурируют за физические потоки

Если мы попытаемся вызвать миллион физических потоков, то этого, скорее всего, не позволит сделать операционная система.

Если мы попытаемся вызвать даже тысячу потоков, ядра процессора захлебнутся на переключении контекста между ними (вытесняющая многозадачность).

Печальный опыт Романа Елизарова

Но мы можем вызвать миллион корутин на 8 потоках (которые будут выполняться на 8 ядрах) без ущерба для производительности.

Да, мы будем держать миллион параллельных задач и нам за это ничего не будет.

Позитивный опыт Романа Елизарова.
Ссылка та же.

Корутины или виртуальные потоки: производительность

Параллельное выполнение	Regular Threads	Virtual Threads	Coroutines
100K	54,649 ms	9,477 ms	5,302 ms
200K	98,614 ms	12,260 ms	5,415 ms

Ali Behzadian’s Benchmarks

Виртуальные потоки концептуально похожи на корутины и придуманы для решения похожих задач, но есть одно отличие.

Это отличие называется «магией виртуальных потоков».

Виртуальные потоки: что это?

При выполнении блокирующей операции физический поток высвобождается. Когда блокирующая операция завершается, виртуальный поток возобновляется на любом доступном физическом потоке.

При вызове блокирующей операции:

1. JVM понимает, что метод вызывается в виртуальном потоке.
2. Вместо реального блокирования, JVM ставит асинхронную подписку на событие готовности данных.
3. Виртуальный поток приостанавливается, его состояние сохраняется в стеке.
4. Физический поток переключается на другую задачу.

При получении данных:

1. JVM получает уведомление через Selector.
2. Виртуальный поток пробуждается и продолжает выполнение.

И это работает даже на блокирующем стеке.

При вызове блокирующей операции ввода / вывода JVM высвобождает физический поток и делает его доступным для других виртуальных потоков.

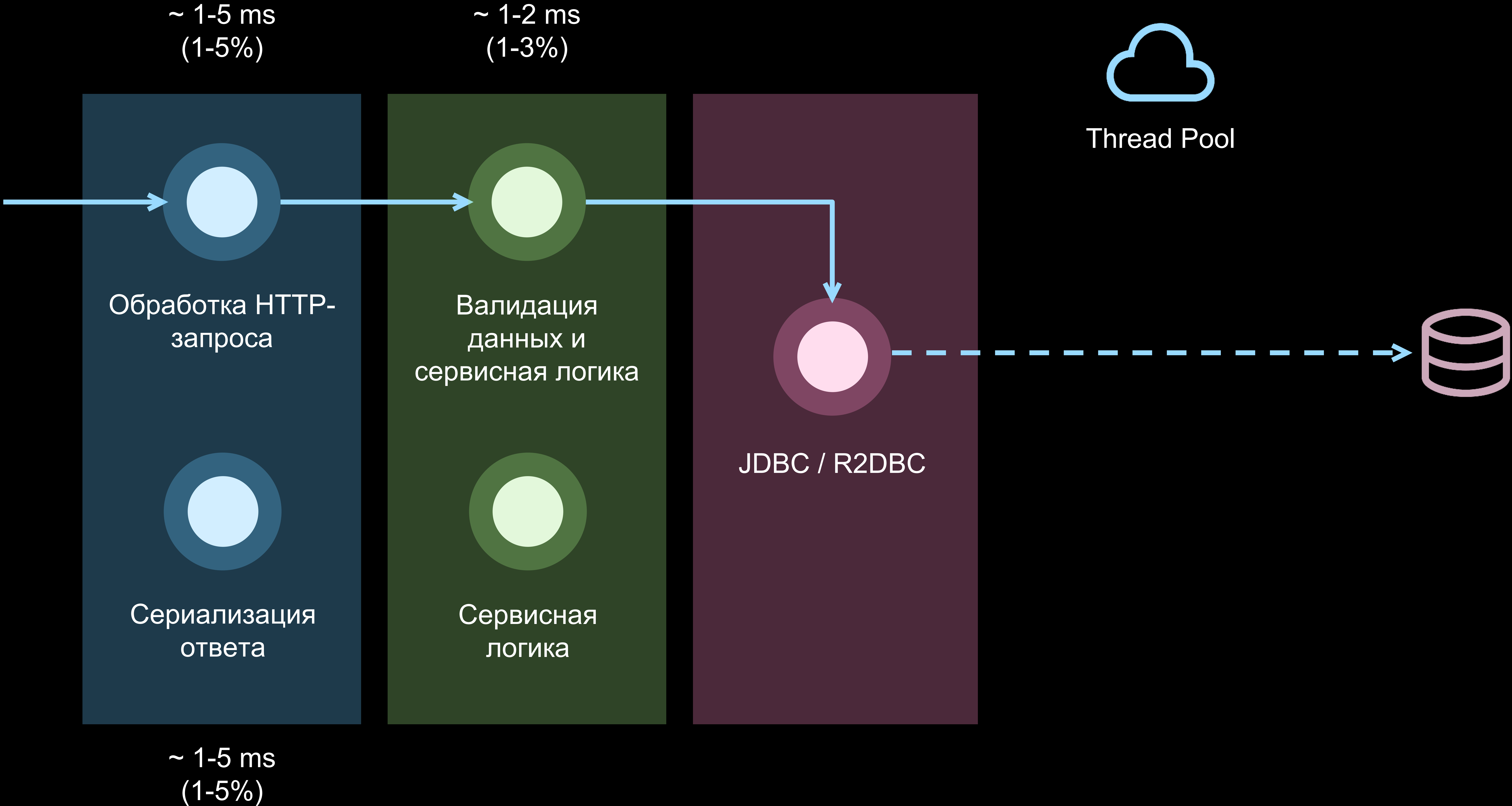
Пояснение от Oracle

When code running in a virtual thread calls a blocking I/O operation, the Java runtime suspends the virtual thread until it can be resumed.

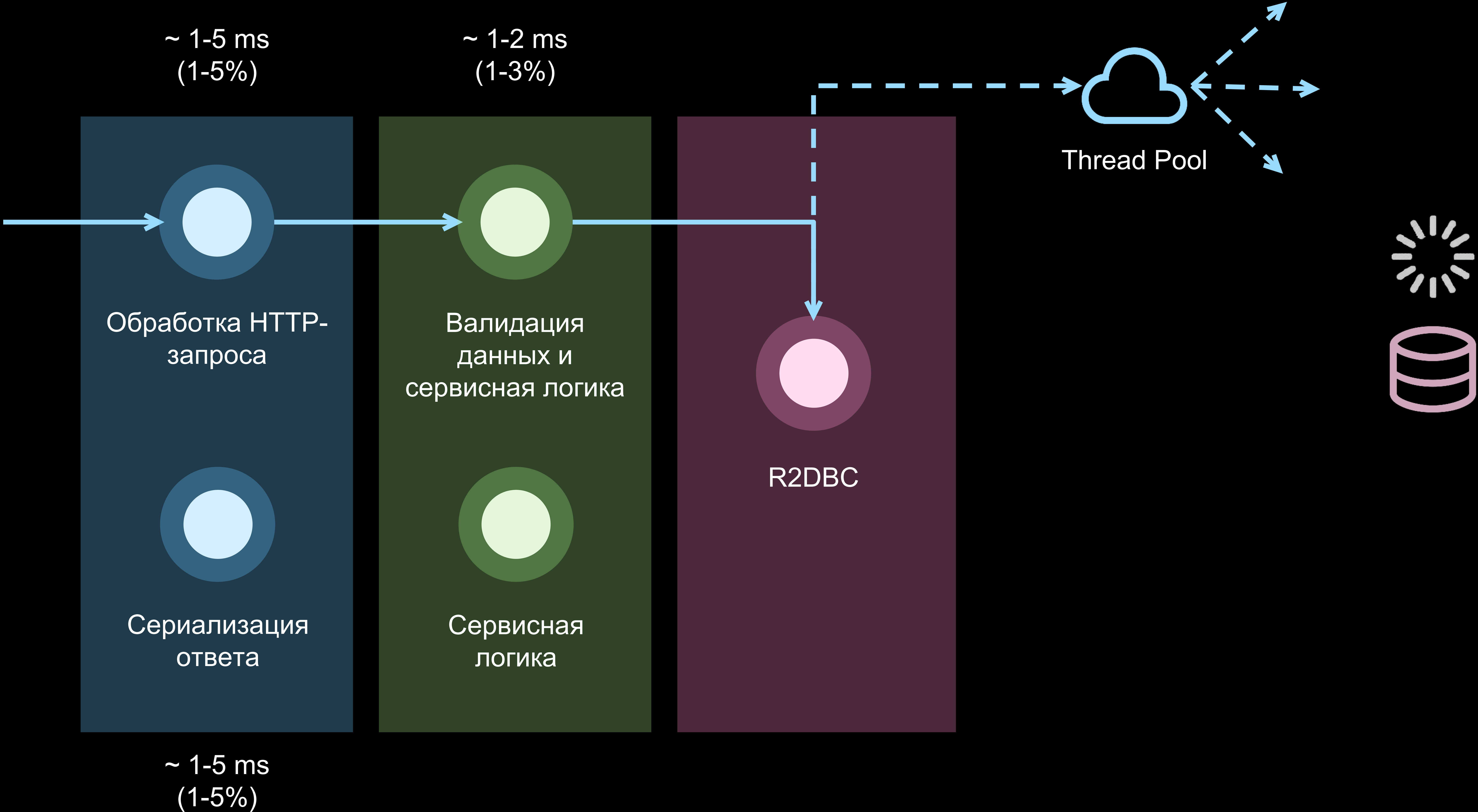
The OS thread associated with the suspended virtual thread is now free to perform operations for other virtual threads.

(но, если вы выполняете операцию блокировки в блоке synchronized, тут уж JVM бессильна и поток блокируется по-настоящему)

Виртуальные потоки:
как работают с блокирующими операциями ввода-вывода?



Виртуальные потоки:
как работают с блокирующими операциями ввода-вывода?

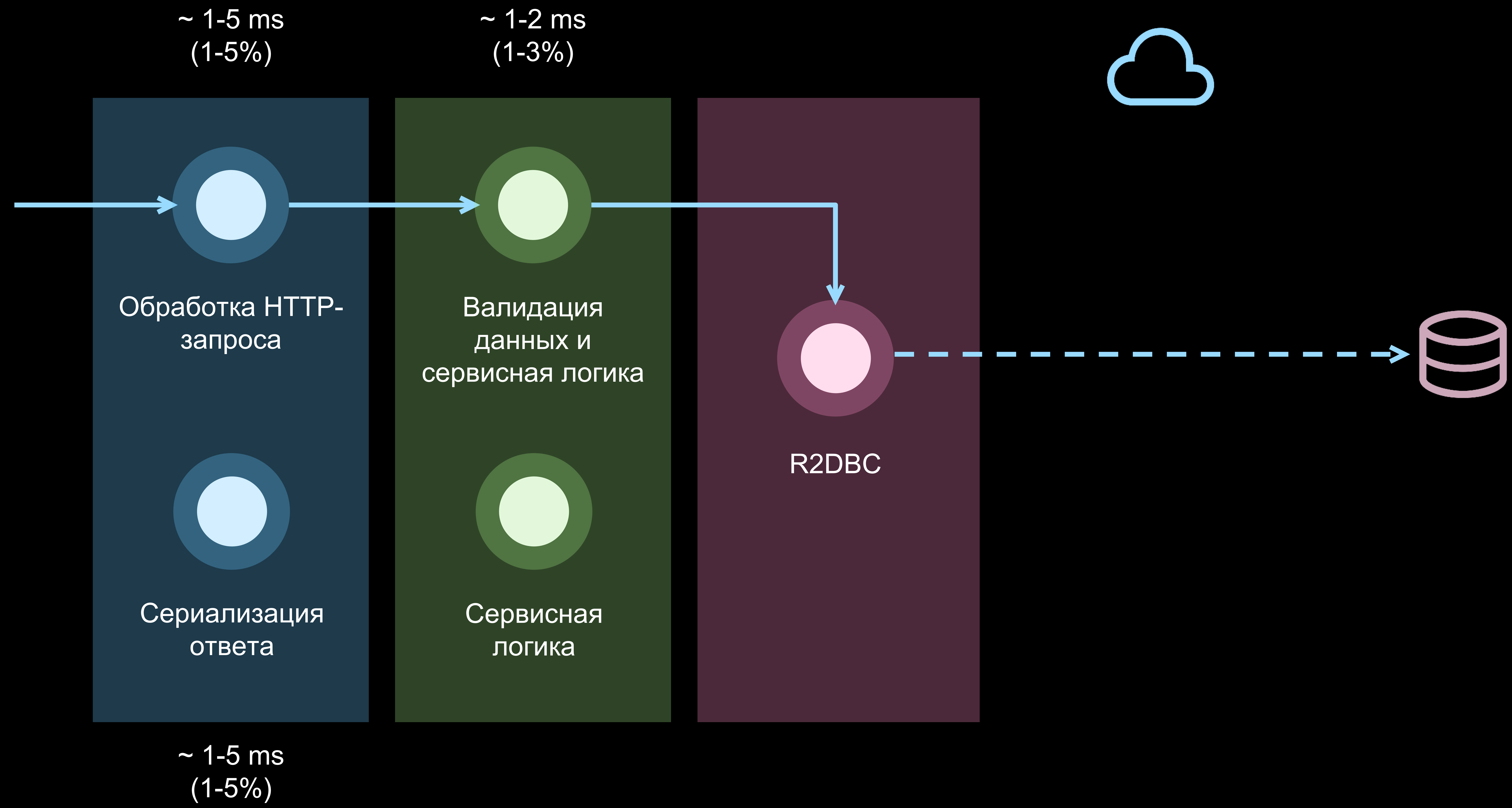


А вот корутины так не могут.

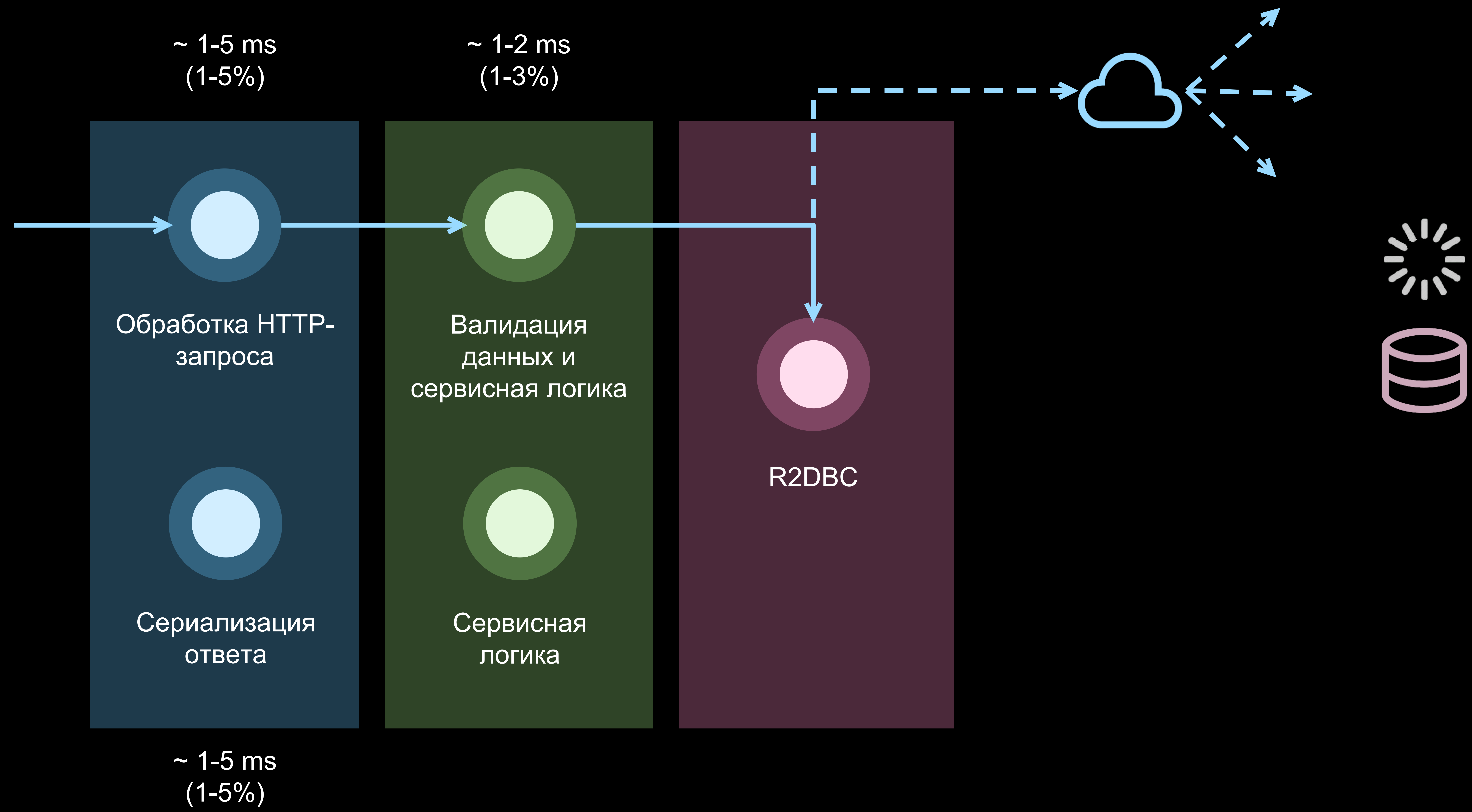
А всё потому, что они работают на уровне функций, а не потоков. В отличие от виртуальных потоков, которые интегрированы в JVM.

Но выход есть.

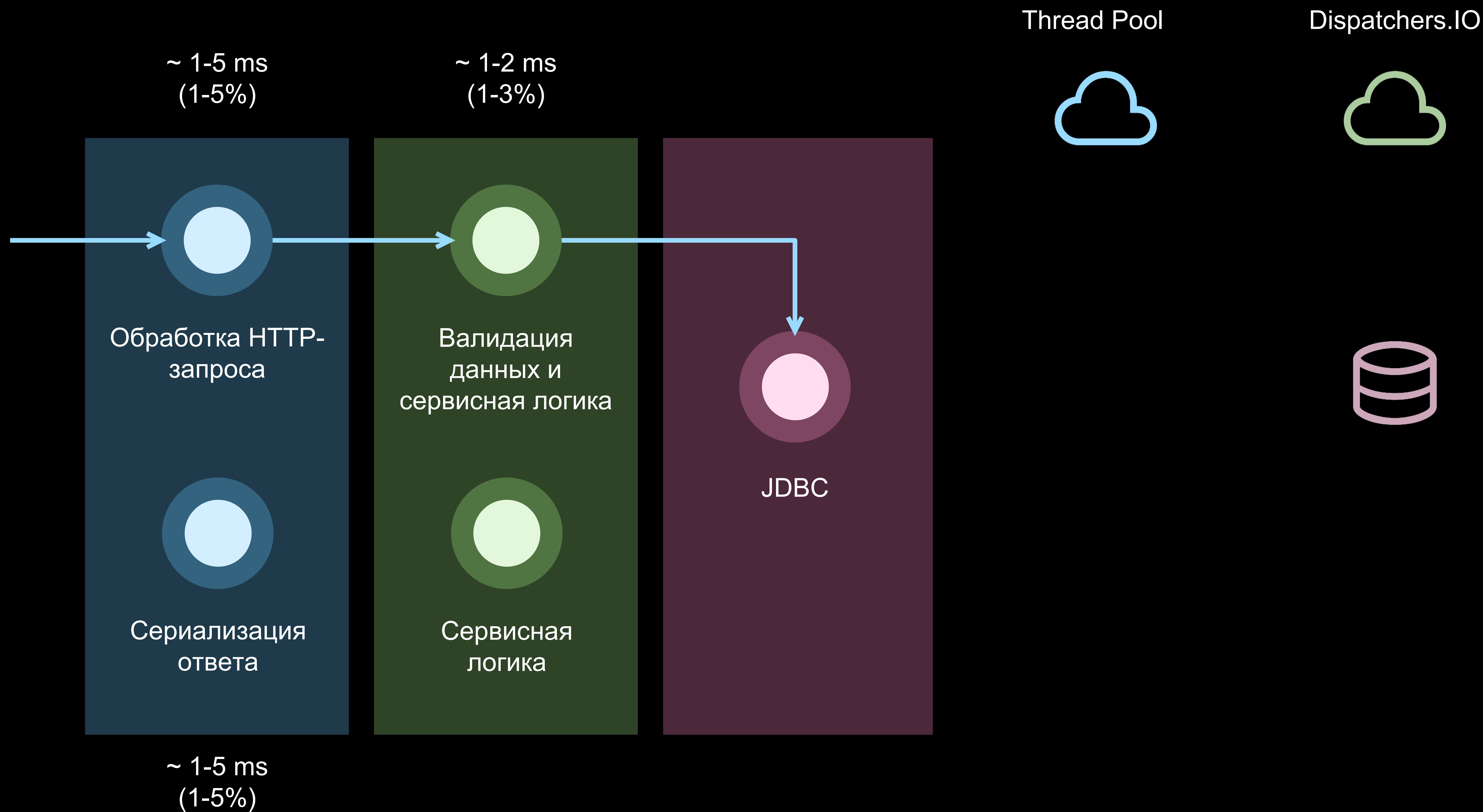
Корутины: операции ввода / вывода при неблокирующем стеке.



Корутины: операции ввода / вывода при неблокирующем стеке.



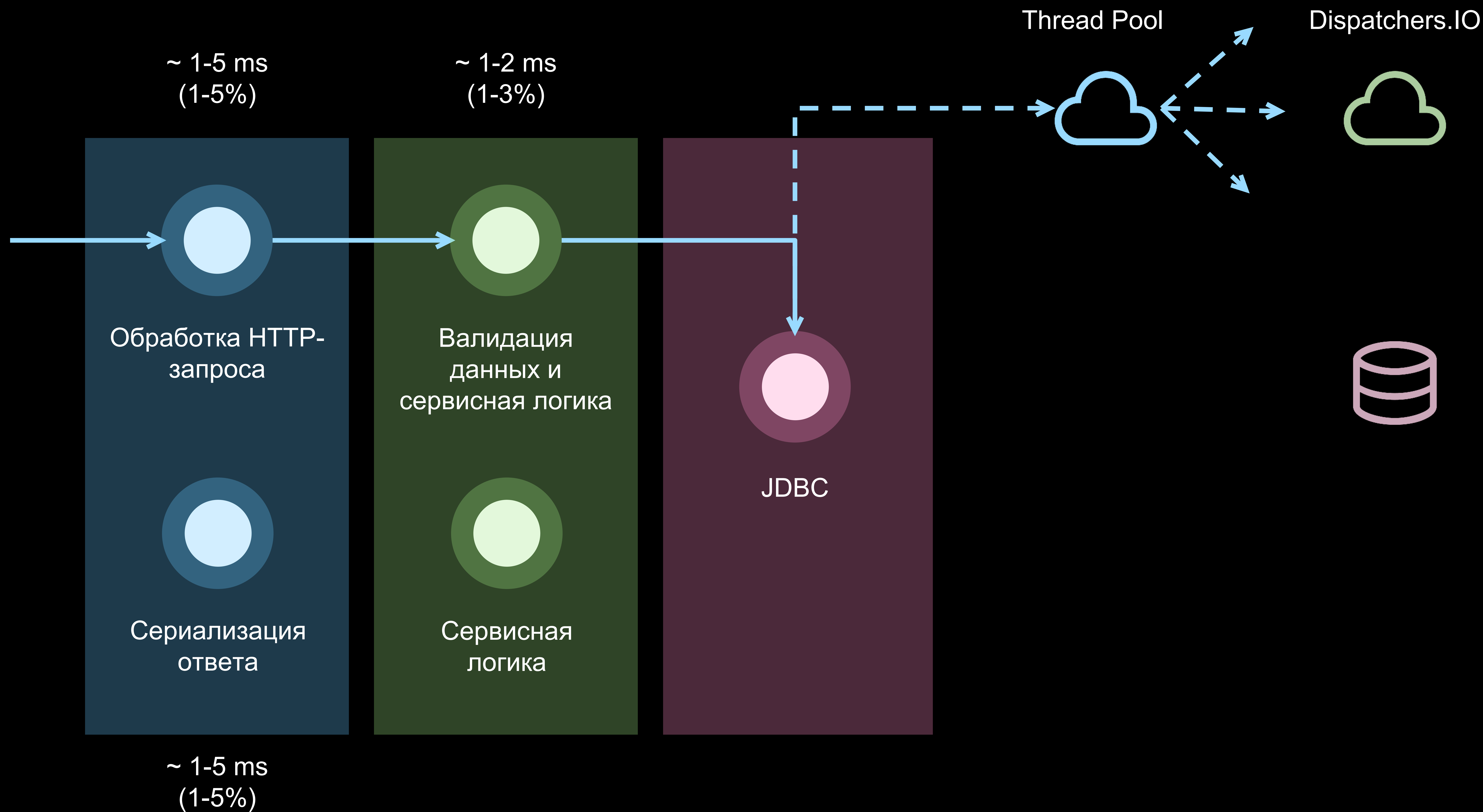
Корутины: как они помогут нам с блокирующими операциями ввода-вывода?



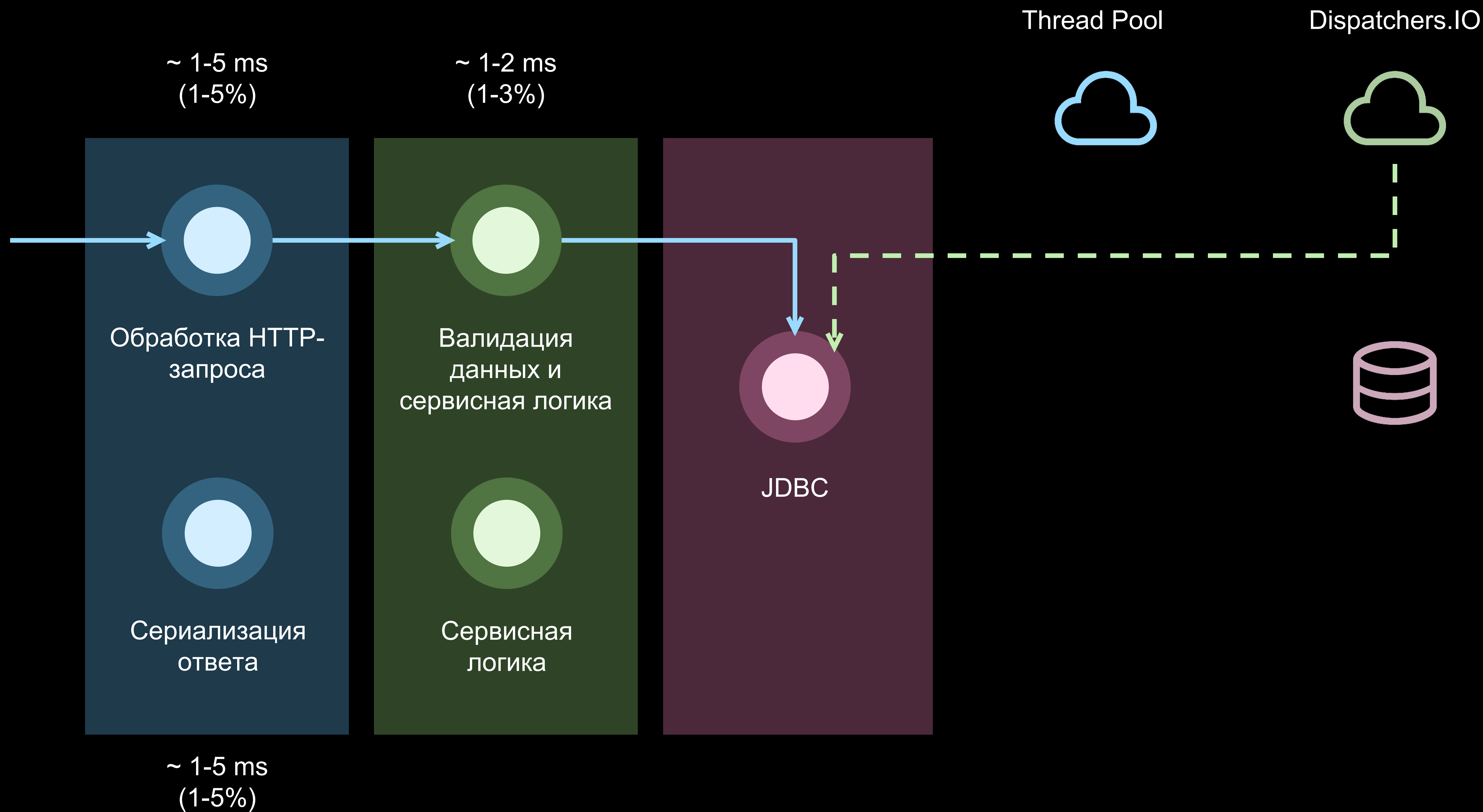
Диспетчеры корутин

Dispatchers.MAIN	Dispatchers.IO	Dispatchers.Default	Dispatchers.Unconfined
Работа с UI (Android)	Блокирующие I/O операции	CPU-интенсивные задачи	Без привязки к конкретному пулу
Основной поток	До 64 потоков	По количеству ядер	Любой поток

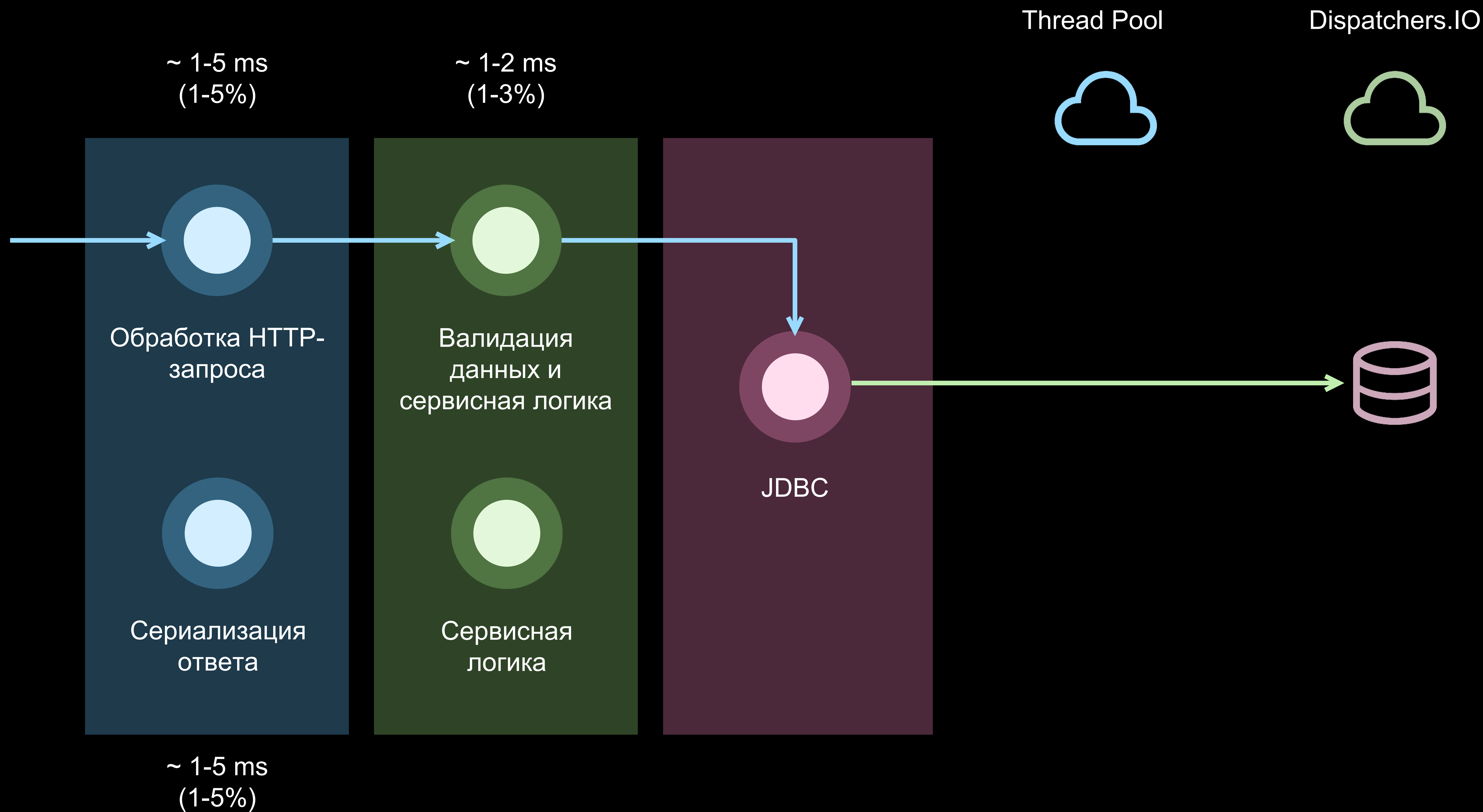
Корутины: как они помогут нам с блокирующими операциями ввода-вывода?



Корутины: как они помогут нам с блокирующими операциями ввода-вывода?



Корутины: как они помогут нам с блокирующими операциями ввода-вывода?



Благодаря отдельному диспетчеру с собственным пулом, корутины минимизируют ущерб от блокировки потока.

Подведём итог.

Итого. Ответы на вопросы.

Помогает ли неблокирующий код существенно сэкономить системные ресурсы?

Да, однозначно. Время ожидания выполнения операций ввода / вывода занимает почти всё время выполнения запроса. Высвобождение потока позволяет направить его ресурсы на обработку других запросов вместо ожидания.

Затраты ресурсов на содержание потока при этом падают на порядки.

Итого. Ответы на вопросы.

Помогает ли неблокирующий код существенно сэкономить системные ресурсы?

Нет, даже при реактивном подходе параллельное выполнение сотен физических потоков на паре ядер процессора захлебнётся на переключении контекста.

Значит ли это, что создать несколько тысяч или даже сотен физических потоков является хорошим решением?

Количество физических потоков должно приближаться к количеству ядер поды.

Для всего остального есть потоки виртуальные. И корутины.

Итого. Ответы на вопросы.

Помогает ли неблокирующий код существенно сэкономить системные ресурсы?

Значит ли это, что создать несколько тысяч или даже сотен физических потоков является хорошим решением?

Решают ли виртуальные потоки и корутины проблему блокировки потоков?

Да, решают.

Виртуальные потоки:

При выполнении блокирующей операции виртуальный поток приостанавливается, carrier thread высвобождается, а при окончании операции I/O виртуальный поток возобновляется на любом доступном carrier thread.

Корутины:

При использовании suspend-функции, при вызове неблокирующего стека корутина приостанавливается, а поток высвобождается. При окончании выполнения операции корутина возобновляется на любом доступном потоке.

Итого. Ответы на вопросы.

Помогает ли неблокирующий код существенно сэкономить системные ресурсы?

Значит ли это, что создать несколько тысяч или даже сотен физических потоков является хорошим решением?

Решают ли виртуальные потоки и корутины проблему блокировки потоков?

При этом, если корутина выполняется на блокирующем стеке, она не может сделать выполнение операции неблокирующим.

Но благодаря отдельному Thread Pool, доступному через Dispatchers.IO, корутины могут минимизировать потери производительности.

Впрочем, никто не мешает вам поменять стек приложения на реактивный, если у вас Котлин.

Я так делал, и мне за это ничего не было.

Спасибо!

Профиль на Хабре

Канал в Telegram