**РЕФЕРАТ**

Отчет 29 с., 27 рис., X источн., 1 табл.

Ключевые слова: нагрузочное тестирование, системные ресурсы, производительность, тестирование сервисов, троттлинг

Объект исследования ВКР – нагрузочное тестирование сервисов.

Цель работы – на основе проведенного исследования методологии нагрузочного тестирования изучить производительность конкретного сервиса и, в случае необходимости, провести планирование ресурсов для повышенных нагрузок на данный сервис.

Выпускная квалификационная работа содержит 3 основных раздела. Первый посвящен методологии нагрузочного тестирования. В нем приведено исследование основных подходов и практик проведения и анализа результатов нагрузочного тестирования сервисов. Во втором разделе применяются изученные методологии и проводится первичное нагрузочное тестирование конкретного сервиса с анализом результатов. В третьем разделе приводятся рекомендации о планировании ресурсов тестируемого сервиса по результатам нагрузочного тестирования.

Результаты проведенных теоретических и практических исследований можно применять в промышленной разработке приложений.

**СОДЕРЖАНИЕ**

[1 ВВЕДЕНИЕ ПРО НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ 3](#_Toc198474557)

[2 МЕТОДОЛОГИЯ НАГРУЗОЧНОГО ТЕСТИРОВАНИЯ 5](#_Toc198474558)

[Нагрузочные тесты при разных бизнес-сценариях 5](#_Toc198474559)

[Нагрузочное тестирование на разных этапах жизненного цикла 6](#_Toc198474560)

[Первичное нагрузочное тестирование 6](#_Toc198474561)

[Стресс-тестирование 16](#_Toc198474562)

[ПРИМЕНЕНИЕ МЕТОДОЛОГИИ НАГРУЗОЧНОГО ТЕСТИРОВАНИЯ 19](#_Toc198474563)

[Постановка задачи 19](#_Toc198474564)

[Подготовка к первичному нагрузочному тестированию 19](#_Toc198474565)

[Проведение первичного нагрузочного тестирования 21](#_Toc198474566)

[ПЛАНИРОВАНИЕ РЕСУРСОВ ПОД НАГРУЗКИ 31](#_Toc198474567)

# **ВВЕДЕНИЕ ПРО НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ**

В промышленной разработке сервисов немаловажную часть занимает исследование их поведения на разных этапах их жизненного цикла. В особенности - непосредственно перед выпуском сервисов в production и уже после выпуска.

Отследить работу конкретной версии приложения и удостовериться, что оно работает так, как было задумано командой разработки, помогают различного рода тесты. Например, модульные тесты помогают отследить корректную работу отдельно взятых методов и компонентов, интеграционные тесты проверяют работу различных модулей в совокупности, допустим, взаимодействие приложения с базой данных. Сквозные тесты проверяют правильную работу всего приложения с точки зрения сценариев конечных пользователей.

Однако данные методы не дают нам ответа на вопрос о том, как сервисы повели бы себя в определенных нестандартных ситуациях, которые произойдут или могут произойти в будущем. Нужным инструментом в таком случае является нагрузочное тестирование, позволяющее выявить узкие места в работе приложения при различных профилях взаимодествия с сервисами, описать, какие пределы в производительности есть у текущей версии приложения для планирования дальнейших действий при разработке, и ответить еще на ряд технических и бизнес вопросов.

Команде разработки необходима информация о производительности сервисов при сценариях работы, отличающихся от текущих, по следующим причинам:

1. бизнес заинтересован в лояльности и хорошем опыте использования сервисов своей аудиторей. Это, в свою очередь, поможет бизнесу развиваться и вкладывать полученные ресурсы в дальнейшее развитие.
2. осуществляя регулярное нагрузочное тестирование, команда разработки имеет представление о возможностях производительности своих сервисов: какие из них хуже работают под нагрузкой, а какие лучше, сколько запросов в секунду выдерживают сервисы и какая у них скорость обработки запросов. Если команда разработки предполагает скорое расширение аудитории своих сервисов, знания о их производительности позволит заранее заложить нужные задачи для оптимизации узких мест и производительности сервисов. Это, в свою очередь, поможет сервисам пережить пики нагрузки без ущерба для пользовательского опыта.
3. механизм нагрузочного тестирования можно встроить в цикл выпуска новых версий сервиса, тем самым заранее будет возможно получать информацию о том, как очередная версия сервиса скажется на его производительности.
4. на ранних этапах разработки новых сервисов нагрузочное тестирование позволит разработчикам сделать выбор в пользу тех или иных технологий или алгоритмов, поможет избежать внесения крупных изменений в код в случае неудовлетворительной производительности на более поздних стадиях разработки проекта.

Разобравшись в проблемах, которые можно решить с помощью нагрузочного тестирования, рассмотрим, какие есть подходы к проведению нагрузочного тестирования и как ими пользоваться.

# **2 МЕТОДОЛОГИЯ НАГРУЗОЧНОГО ТЕСТИРОВАНИЯ**

## Нагрузочные тесты при разных бизнес-сценариях

В зависимости от определенных бизнес-сценариев, нагрузочные тесты могут проводиться по-разному. Кроме того, интерпретация результатов тестирования также напрямую зависит от этих сценариев. Разберем некоторые примеры:

1. В API-сервисе присутствует метод, производительность которого очень сильно зависит от переданных параметров. Например, имеем метод получения первых “n” документов пользователя. В зависимости от числа “n” метод будет отрабатывать за разное время. С помощью нагрузочного тестирования хотим проверить, как будет изменяться производительность метода вместе с ростом числа запрашиваемых документов. По результатам нагрузочного тестирования сможем подобрать оптимальную верхнюю границу для числа запрашиваемых документов.
2. В связи с приближающимися праздниками ожидается всплеск активности пользователей мессенджера, в два раза превышающий текущий уровень. Требуется через нагрузочное тестирование проверить, сможет ли сервис выдержать двойную нагрузку с примерно таким же соотношением запросов по методам и с такими же параметрами.

Как можно заметить, в зависимости от бизнес-требований, выдвинутых к нагрузочным тестам, имеем разные ответы на вопросы «что» и «как» тестировать. Без четко определенных целей и сценариев невозможно провести тестирование, которое принесло бы пользу разрабатываемому продукту.

## Нагрузочное тестирование на разных этапах жизненного цикла

Кроме различий в подходах к нагрузочному тестированию в зависимости от конкретных бизнес-сценариев, можно выделить такие различия и в зависимости от конкретного этапа жизненного цикла продукта. На разных таких этапах разработки продукта мы можем иметь разные методы проведения нагрузочных тестов.

### *Первичное нагрузочное тестирование*

Важным является первичное нагрузочное тестирование сервиса. Оно имеет свои особенности, поскольку тестируемый сервис либо еще не был запущен в production, либо для него еще не проводилось нагрузочное тестирование, а следовательно, нет предыдущих результатов, на которые можно было бы опираться и сравнивать изменения в производительности.

Первичное нагрузочное тестирование должно иметь целью подобрать правильные настройки окружения, такие как количество процессорных ядер, количество оперативной памяти, количество реплик приложения, на каких хостах лучше запустить сервис, для приложений на некоторых платформах – определить размер пула потоков.

Нагрузочное тестирование поможет определиться с правильными настройками самого приложения, например, определить размер пачек для пакетной обработки данных, поможет понять, как быстро сервис отвечает на запросы разных типов с разными параметрами запросов. Первичное нагрузочное тестирование повышает вероятность избежать аварий и критических ситуаций при старте сервиса, поскольку у команды разработки появляется хотя бы приблизительное представление о производительности сервиса, о его узких местах и о том, что можно ожидать от его работы в тех или иных ситуациях.

Для того, чтобы начать проводить нагрузочное тестирование и делать это качественно, сервис должен быть к этому подготовлен. Для этого сервису нужно соответствовать ряду критериев. Какие-то из этих критериев сервис может соблюдать сразу, а по каким-то будет требоваться доработка или даже масштабное изменение в архитектуре сервиса. Приведу несколько самых важных критериев:

1. сбор метрик сервиса. Для анализа результатов нагрузочных тестов и исследования производительности с сервиса должны собираться необходимые для этого метрики. Системные метрики являются выжным примером таких метрик. С сервиса необходимо собирать метрики по использованию процессора, оперативной памяти, сети, диска. Также важно собирать метрики по HTTP-взаимодействию сервиса, такие как количество запросов в секунду, квантили времени обработки запроса, коды ответа на запросы. Метрики будут отражать результаты теста, которые потом нужно будет интерпретировать.
2. площадка для тестирования. Для того, чтобы проводить нагрузочное тестирование сервиса, он должен быть где-то развернут. Для этого нужна площадка для тестирования. Нужно убедиться, что нагрузочное тестирование на данной площадке не помешало работе других команд разработки и не сказалось на пользователях сервиса. Одним из вариантов достижения таких условий будет развертывание отдельных реплик тестируемого сервиса специально под нагрузочное тестирование, а для сервисов-зависимостей – добавление дополнительных реплик для избыточности. Таким образом, тестируемый сервис не ставит сервисы-зависимости в ограничение для тестирования, а сами сервисы-зависимости не будут страдать от проведения нагрузочного тестирования.
3. горизонтальная масшабируемость сервиса. Это означает, что производительность кластера сервиса в целом может быть улучшена путем добавления новых реплик тестируемого сервиса. Таким образом, мы сможем протестировать сервис всего с одной запущенной репликой, а после тестирования экстраполировать результаты на большее количество реплик. Кроме того, тестирование одной реплики гораздо проще: не нужно затрачивать ресурсы на поддержание кластера реплик, нужно в разы меньше ресурсов для подачи нагрузки на сервис.

Теперь можно перейти к описанию конкретных тестов, которые помогут ответить на один из поставленных выше вопросов, интересующих команду разработки при проведении первичного нагрузочного тестирования.

Для начала рассмотрим тест на производительность сервиса. Он поможет выяснить, сколько запросов в секунду способен обработать сервис без значительного ухудшения показателей производительности. Во время проведения такого теста необходимо обращать внимание на количество запросов в секунду, время ответа сервиса и количество успешных или неуспешных HTTP кодов ответа.

Для проведения такого теста нам нужно подавать линейную нагрузку на сервис, постепенно увеличивая количество отправляемых запросов. Это нужно делать до тех пор, пока показатели производительности сервиса не ухудшатся. Такими индикаторами послужит увеличившееся время ответа на запросы и появление 5xx HTTP-кодов ответа.



Рисунок 1 – схематичный пример графика кодов ответа в секунду при проведении теста на производительность сервиса

В какой-то момент во время проведения теста сервис исчерпает один из своих ресурсов: процессор, память, сеть или диск, в следствие чего его производительность заметно ухудшится. Ухудшение производительности будет проявляться через увеличение времени ответа на запросы, истечение времени на обработку запросов, 5xx коды ответов HTTP, полную недоступность сервиса. Количество успешно обработанных запросов начнет снижаться, сервис перестает быть стабильным. Однако возможно и такое, что сервер достиг своего предела по количеству успешно обработанных запросов, но никакой из системных ресурсов не используется до конца. Это может говорить о возможном наличии ошибок проектирования сервиса или его конфигурации. Хорошей практикой будет являться проведение теста несколько раз, чтобы сгладить погрешность и добиться более правдивых результатов. Стоит отметить, что такая практика актуальна для любого вида нагрузочного тестирования, а не только для первичного теста на измерение производительности.

Важно правильно подобрать запросы, которыми будем нагружать сервис. Лучше проводить тестирование по разным типам запросов, чтобы убедиться, что для всех обрабатываемых сервисом запросов требуется примерно одинаковое количество ресурсов. Для каждого отдельного запроса лучше собрать смесь из запросов с разной тяжестью: легкие, средние и тяжелые. На тяжесть запроса влияют переданные в него параметры. Результаты тестирования на легких и тяжелых запросах не должны различаться на порядки.

Если результат тестирования показал, что разница в производительности между легкими и тяжелыми запросами отличается очень значительно, можно провести следующие шаги:

1. Можно разделить один сервис на несколько и распределить пользовательские сценарии между ними. Таким образом удастся минимизировать влияние одного типа запросов на другой, а профиль нагрузки в рамках каждого из таких сервисов может измениться с гораздо меньшей вероятностью. Однако при таком подходе придется потратить достаточно ощутимое количество времени, что не всегда бывает оправданным.
2. Можно попробовать сфокусироваться на худшем сценарии. В таком случае высока вероятность искусственно снизить пропускную способность сервиса путем более сильного влияния механизмов защиты от перегрузки на сервис. В некоторых ситуациях такой подход имеет место быть.
3. Если сервис уже используется в production среде, можно зафиксировать типичное распределение запросов по типу и переданным параметрам и тестировать сервис на таком наборе запросов. Этот способ не подойдет, если сервис еще не запускался в production.
4. При внедренном механизме троттлинга или rate limiting-а запросов, можно настроить квоты для каждого запроса в зависимости от его веса. Такой подход имеет ряд недостатков из-за сложности правильной реализации и долгой настройки.

Каждый из этих подходов имеет свои ситуации для применения, и выбор любого из них зависит от конкретных обстоятельств и конкретного сервиса.

Далее рассмотрим тест на настройку механизма защиты сервиса от перегрузки запросами – троттлинга. Троттлинг ограничивает число запросов, одновременно обрабатываемых сервисом, в целях защиты сервиса от перегрузки и деградации. При превышении такого порога запросы начинают отбрасываться с HTTP кодом ответа 429. Для того, чтобы извлечь максимальную пользу от использования троттлинга и не причинить ущерба работе сервиса при его внедрении, необходимо подобрать правильные настройки. В этом как раз поможет тест на настройку троттлинга.

Механизм троттлинга в некоторых аспектах может быть реализован по-разному, и какие-то детали настройки будут как раз зависеть от конкретной реализации, однако основная механика работы обычно одинакова, или отличия незначительны.

После проведения предыдущего теста на производительность сервиса у нас есть информация о том, какую максимальную нагрузку может выдержать сервис без деградации производительности. Исходя из этой информации у нас есть возможность посчитать, сколько запросов одновременно может быть обработано сервисом. Можем использовать закон Литтла из теории массового обслуживания, чтобы посчитать, сколько максимум запросов одновременно может обслуживать наш сервис без потери производительности. Закон Литтла гласит, что долгосрочное среднее количество *L* требований в стационарной системе равно долгосрочной средней интенсивности *λ* входного потока, умноженной на среднее время *W* пребывания заявки в системе. Данный закон применим к любым системам. Проецируя этот закон на наш частный случай, можем задать первое приближение емкости сервиса как *C* = *R* \* *T*, гдеС (capacity) – емкость сервиса, R (requests) – максимальное количество запросов в секунду, которое сервис может выдержать без потери производительности, T (time) – среднее время для обработки одного запроса. Таким образом получим значение C, которое отражает максимальное количество одновременно находящихся запросов на сервисе. После оценки первого приближения емкости троттлинга, нам нужно протестировать найденное значение на практике, выставив данное значение в настройках троттлинга и проведя нагрузочное тестирование с заданной настройкой. Возможно несколько результатов такого тестирования.

В первом случае найденная емкость оказалась слишком большой. Это означает, что настроенный троттлинг не дал никакого эффекта и не спас сервис от перегрузки. Получили такой же результат, как и при отсутствии троттлинга.



Рисунок 2 – результат нагрузочного тестирования на настройку троттлинга в случае слишком сильно завышенной настройки емкости сервиса

Во втором случае полученная нами емкость оказалась слегка завышенной. В такой ситуации на графиках результатов нагрузочного тестирования при достижении сервисом предела пропускной способности увидим возросшее время обработки запросов и немного нестабильное соотношение успешных и неуспешных ответов. В некоторых случаях, при чуть более сильном завышении емкости, можем получить 5xx HTTP коды ответов, но их частота и количество будут не такими значительными, как при первом случае.



Рисунок 3 - результат нагрузочного тестирования на настройку троттлинга в случае небольшого завышения емкости сервиса

В третьем случае емкость сервиса занижена, в результате чего механизм троттлинга начинает отбрасывать запросы гораздо раньше достижения сервисом максимальной пропускной способности и достижения границы по использованию какого-либо ресурса.



Рисунок 4 - результат нагрузочного тестирования на настройку троттлинга в случае выставления слишком низкой емкости сервиса

Теперь разберем случай, когда выставлена подходящая емкость. При правильной настройки емкости механизм троттлинга начнет отклюнять входящие запросы только тогда, когда количество запросов в секунду будет близко к максимальной пропускной способности сервиса. При этом утилизация одного из системных ресурсов также должна быть почти на уровне предельного значения (80-90% от предельного).



Рисунок 5 - результат нагрузочного тестирования на настройку троттлинга в случае правильной настройки емкости

Если результат проведения теста на настройку троттлинга не был близок к последнему рассмотренному случаю, необходимо скорректировать настройку емкости в зависимости от результата и провести тестирование повторно. Также важно упомянуть, что настройку троттлинга необходимо нормировать в соответствии с ограничивающим производительность ресурсом: например, если сервис ограничивает использование процессора, то нормировать емкость нужно в зависимости от количества ядер путем задания значения емкости на ядро процессора.

После проведения первичного нагрузочного тестирования сервиса команда разработки получает много полезной информации о работе своего продукта, которая поможет при его старте:

1. Максимальное количество запросов в секунду, которое может выдержать одна реплика тестируемого сервиса. Не стоит забывать, что этот показатель напрямую зависит от соотношения и параметров запросов, на которых тестировался сервис. Считаем, что соблюдено условие возможности горизонтального масштабирования сервиса, поэтому эти данные можно экстраполировать на большее количество реплик сервиса.
2. Данные о среднем времени ответов на запросы к сервису. Обычно ориентируются на 90-й и 95-й перцентили.
3. Правильно настроенный троттлинг, что необходимо для избежания значительного ухудшения производительности сервиса в случае, если будет превышена предельная пропускная способность.

Считаем, что соблюдено условие возможности горизонтального масштабирования сервиса, поэтому эти данные можно экстраполировать на большее количество реплик сервиса.

### *Стресс-тестирование*

Стресс-тестирование нужно для того, чтобы убедиться в возможности сервиса выдержать превышение запланированных нагрузок. К стресс-тестированию можно отнести следующие виды тестов:

1. Тест на линейный рост нагрузки
2. Тест на вспеск нагрузки с возвратом к норме
3. Тест на деградацию сервиса со временем

Рассмотрим их по порядку.

Тест на линейный рост нагрузки поможет отследить поведение сервиса в случае, когда количество приходящих на сервис запросов значительно возросло. Это аналогичный первичному нагрузочному тестированию тест для следующей стадии в жизненном цикле сервиса. Одно из отличий как раз заключается в том, что на время проведения данного теста нам известен профиль нагрузки с production площадки, который мы сможем использовать для тестирования сервиса. Для этого нужно собрать данные по соотношениям типов запросов. Хорошим результатом данного теста может считаться сценарий, при котором сервис продолжает успешно обрабатывать наибольшее возможное количество запросов без просадок в производительности. При приближении к лимитам одного из системных ресурсов начинает срабатывать настроенный на этапе первичного нагрузочного тестирования троттлинг, который отклоняет все избыточные запросы. При плохом результате троттлинг не защитит сервис от перегрузки, сервис попытается обработать каждый входящий запрос и просядет в проценте успешных ответов, длительности ответов на запросы. В таком случае необходимо произвести перекалибровку настроек троттлинга.

Тест на всплеск нагрузки с возвратом к норме позволяет проверить поведение сервиса при кратковременном кратном превышении количества входящих запросов и при спаде этого количества до прежнего значения. Для проведения теста необходимо прогреть сервис на небольшом количестве запросов и постепенно довести нагрузку примерно до 75-85% от той, которую способен выдержать сервис. После небольшой по продолжительности нагрузки на этом уровне доводим количество входящих запросов до примерно двухкратного от максимального и после нескольких минут возвращаем до прежнего уровня. При удовлетворительном результате тестирования получим следующее поведение сервиса: поскольку всплеск нагрузки был хоть и большим, но недолгим, часть запросов из всплеска должна быть поставлена в очередь и быть обработана. Аргументируется это тем, что лучше обработать входящие запросы с небольшой задержкой, если это не навредит сервису, чем сразу отклонить. Важно настроить правильный размер такой очереди. Обычно он колеблется от 500 до 1000 запросов. В противном случае, при большом размере очереди, рискуем чрезмерно нагрузить сервис. Те запросы, которым не хватило места в очереди, должны быть отброшены троттлингом и общая производительность сервиса не должна пострадать.

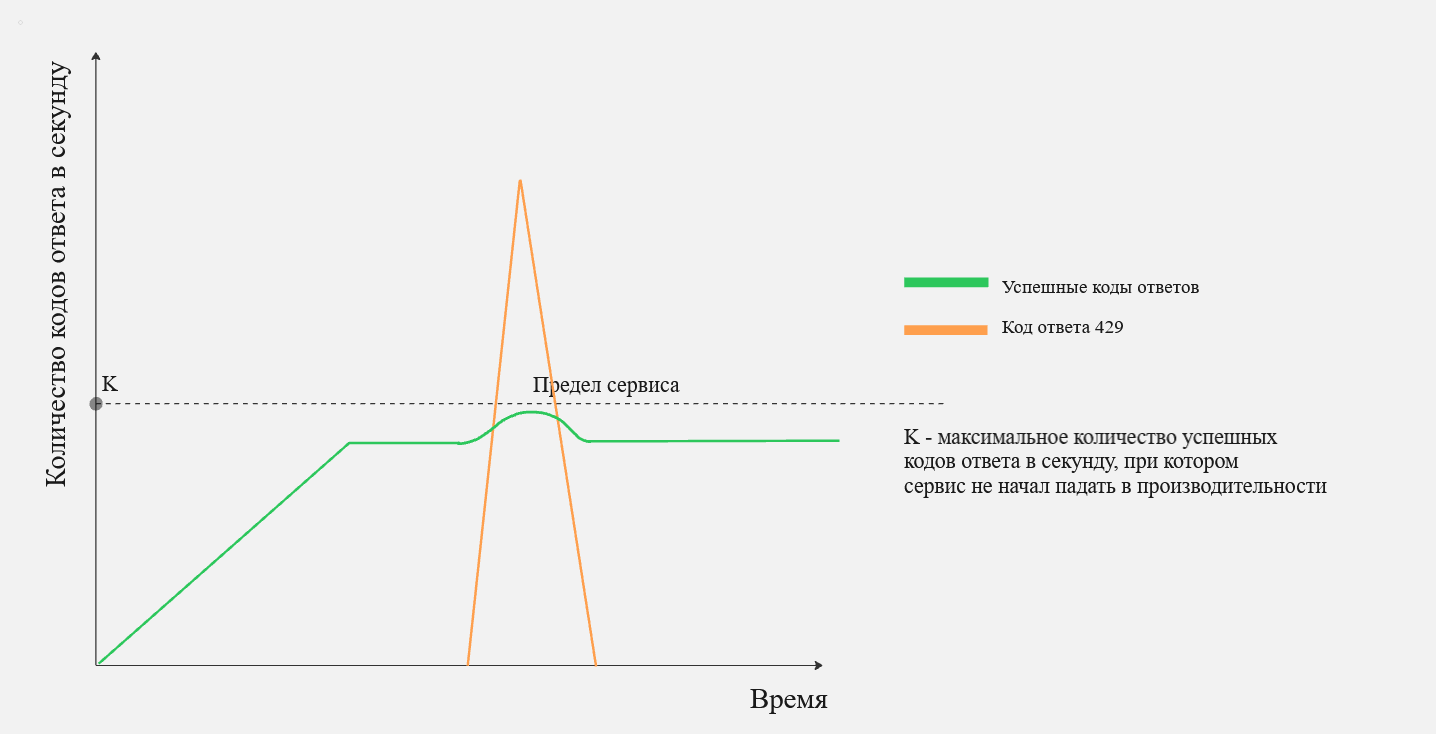


Рисунок 6 – желаемый результат работы сервиса в случае резкого всплеска нагрузки с возвратом к норме

В плохом случае троттлинг не отработает правильно и даст сервису деградировать, поскольку тот попытается обработать все входящие запросы. Еще один вариант плохого результата – сервис получил много запросов в очередь на пике нагрузки, время обработки запросов выросло, но при возврате количества запросов к прежнему уровню время обработки так и не вернулось к норме.

Тест на деградацию с течением времени нужен для выявления таких проблем, которые проявляют себя во время длительной работы сервиса. Например, сервис может штатно работать продолжительное время, но внезапно отказать из-за переполнения доступной ему памяти вследствие утечки. Такой тест должен зайдествовать в себе все методы сервиса и должен быть запущен на продолжительное время. При явной утечке сервис деградирует или вовсе откажет, не дождавшись завершения теста. Если тест все-же завершился, то следует проанализировать системные и сетевые метрики на предмет постоянного роста использования, например, оперативной памяти или каких-либо соединений.

# **ПРИМЕНЕНИЕ МЕТОДОЛОГИИ НАГРУЗОЧНОГО ТЕСТИРОВАНИЯ**

Проведя необходиые исследования, применим методологию нагрузочного тестирования на примере конкретных сервисов.

## Постановка задачи

У команды разработки имеется сервис, для которого еще не проводилось нагрузочное тестирование. Необходимо провести первичное нагрузочное тестирование с целью узнать лимит производительности сервиса и настроить механизм троттлинга. Также необходимо ответить на вопрос, сможет ли сервис при текущих настройках количества оперативной памяти и ядер процессора перенести двойную нагрузку, ожидающуюся в ближайшее время. Если ответ отрицательный, подобрать нужную конфигурацию ресурсов для сервиса.

## Подготовка к первичному нагрузочному тестированию

Для начала убедимся, что сервис отвечает минимальным требованиям для проведения нагрузочного тестирования. Поскольку сервис:

1. Имеет возможность горизонтального масштабирования
2. Поддерживает наблюдаемость: реализован сбор системных метрик и метрик HTTP-взаимодействия с сервисом
3. Запущен на специальной площадке для нагрузочного тестирования с неободимой конфигурацией, сервисы-зависимости также развернуты с избытком

то можем сказать, что сервис соблюдает необходимые минимальные требования.

Далее определимся с конфигурацией сервиса. Будем тестировать результат на одной реплике, которой дадим такое же количество ресурсов по ядрам процессора и оперативной памяти, какое имеется у одной реплики на production площадке. Таким образом сможем легко экстраполировать результаты тестирования одной реплики на несколько реплик с production площадки.

Осталось определиться с тем, какими запросами будем нагружать сервис. Поскольку сервис уже выпущен в production, нам точно известны текущие сценарии использования. Возьмем смесь запросов с боевой площадки, которую мы определим по HTTP-метрикам количества входящих запросов в секунду для каждого из методов сервиса. В результате составим таблицу соотношения конкретного метода и его веса по сравнению с самым наименее используемым методом, вес которого возьмем за единицу:

Таблица 1 – Соответствие конкретного метода и его веса

|  |  |
| --- | --- |
| № метода | Вес |
| 1 | 24,0 |
| 2 | 10,0 |
| 3 | 8,0 |
| 4 | 4,3 |
| 5 | 3,0 |
| 6 | 2,3 |
| 7 | 1,0 |

Данная таблица дает нам информацию о том, с каким соотношением должны отправляться запросы на конкретные методы во время проведения нагрузочного тестирования.

Теперь определимся, как будут составлены отправляемые запросы. Исходя из знаний о тестируемой системе, все методы оперируют двумя типами сущностей из бизнес-логики сервиса, и размер этих сущностей варьируется достаточно слабо, чтобы этой разницей можно было пренебречь. Поэтому считаем, что можно с одинаковым результатом запросить любую сущность из системы. Также из анализа телеметрии сервиса видим, что почти все запросы на получение объектов сущностей из базы данных запрашивают только 100 одних и тех же сущностей. Эти методы могут отдавать только один объект на один запрос. Для остальных методов из выборки переданные параметры никак не могут сказаться на результате их работы. Следовательно, делаем вывод, что для методов получения объектов из базы данных нужно предварительно выбрать 100 объектов, которые будут запрашиваться во время нагрузки. Для остальных методов можно использовать случайно сгенерированные параметры, для них достаточно только лишь соблюсти соотношение запросов.

Также стоит отметить, что все запросы методов в базу данных осуществляются по индексам, индексы на тестовом и production окружениях совпадают.

## Проведение первичного нагрузочного тестирования

Сначала проведем тест на производительность сервиса. Следуя методике, будем подавать линейную нагрузку на сервис, постепенно увеличивая количество отправляемых запросов. Возьмем, для начала, линейный рост нагрузки до 500 запросов в секунду в течение 5 минут с учетом соотношения запросов. Посмотрим на результаты.

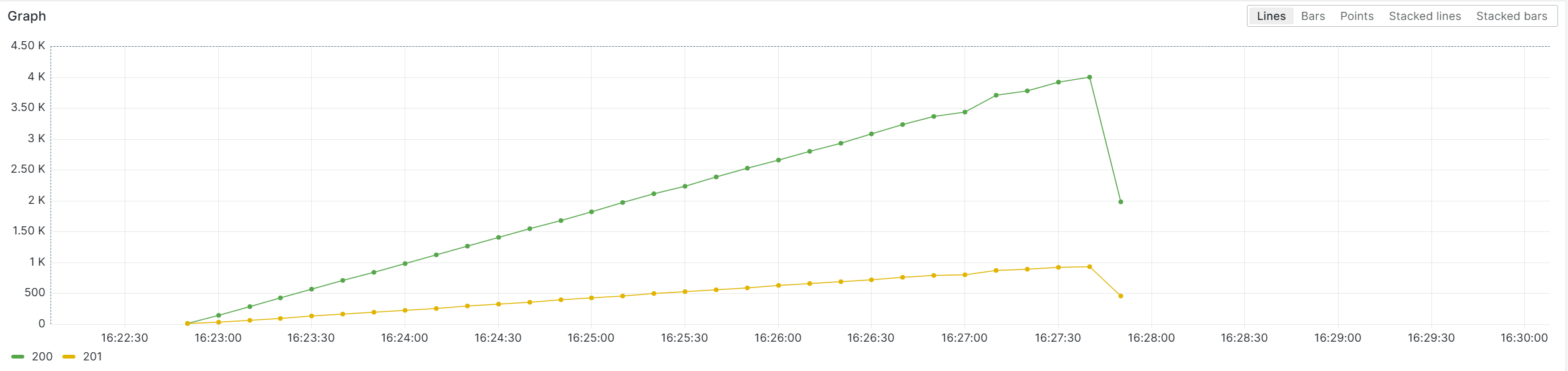


Рисунок 7 – график успешных HTTP ответов на запросы нагрузки в первом тесте на производительность сервиса

Видим линейно возрастающее число успешных ответов на запросы. График неуспешных ответов показывает, что запросов, завершившихся клиентской или серверной ошибкой, не было. Небольшой спад в конце связан с тем, что сервис продолжил обрабатывать оставшиеся запросы после завершения подачи нагрузки.

A graph with green lines

AI-generated content may be incorrect.

Рисунок 8 – график потребления ядер процессора в процентном соотношении от максимально допустимого в первом тесте на производительность сервиса

A white screen with black lines

AI-generated content may be incorrect.

Рисунок 9 – график потребления памяти в процентном соотношении от максимально допустимого в первом тесте на производительность сервиса

A graph with different colored lines

AI-generated content may be incorrect.

Рисунок 10 – график среднего времени ответа на запрос для каждого метода в миллисекундах в первом тесте на производительность сервиса

Как можем видеть из результатов, сервис без проблем выдержал линейно возрастающую нагрузку до 500 запросов в секунду. Ни один из системных ресурсов не превысил критическое значение. Количество серверных и клиентских HTTP ошибок равно нулю. Следовательно, нужно подать более высокую нагрузку.

Попробуем во втором тесте нагрузить сервис до 1250 запросов в секунду и посмотрим на результаты.

A graph of a graph

AI-generated content may be incorrect.

Рисунок 11 - график успешных HTTP ответов на запросы нагрузки во втором тесте на производительность сервиса

A graph with different colored lines

AI-generated content may be incorrect.

Рисунок 12 - график клиентских ошибок HTTP во втором тесте на производительность сервиса

A graph showing a green line

AI-generated content may be incorrect.

Рисунок 13 - график серверных ошибок HTTP во втором тесте на производительность сервиса

A white sheet with a red line

AI-generated content may be incorrect.

Рисунок 14 – график потребления ядер процессора в процентном соотношении от максимально допустимого во втором тесте на производительность сервиса

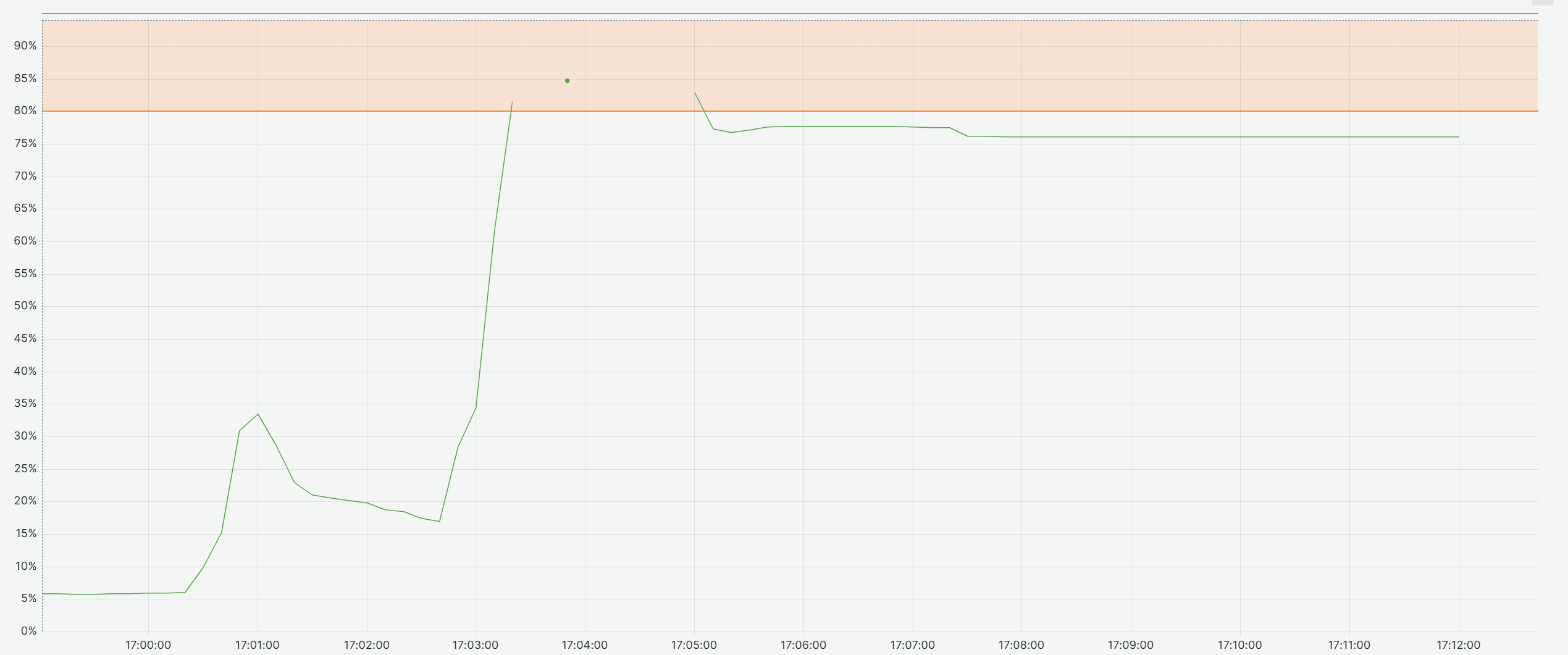


Рисунок 15 - график потребления памяти в процентном соотношении от максимально допустимого во втором тесте на производительность сервиса

A graph of a graph

AI-generated content may be incorrect.

Рисунок 16 - график среднего времени ответа на запрос для каждого метода в миллисекундах во втором тесте на производительность сервиса

Из результатов теста очевидно, что сервис не справился с нагрузкой и деградировал. При приближении уровня потребления процесса к 100% также очень быстро начало убывать свободное место в оперативной памяти. На графике среднего времени отета на запрос хорошо заметно плато в 30 секунд – это максимальное время ожидания ответа от сервиса нагрузочным агентом. Поскольку за все 30 секунд нагрузочный агент не смог дождаться ответа от сервиса, можно сделать вывод, что в этот промежуток времени сервис был максимально перегружен. Заметное ухудшение в производительности сервиса произошло на отметке примерно в 600 запросов в секунду, будем считать это значение максимальной пропускной способностью сервиса. Также во время тестирования выяснили, что главный ресурс, расходуемый сервисом – ядра процессора. В дальнейших тестах будем ориентироваться именно на этот ресурс.

Выяснив максимальную пропускную способность сервиса, рассчитаем необходимую емкость, которую выставим в настройки троттлинга. Умножаем максимальную пропускную способность сервиса в 600 запросов в секунду на среднее время ответа от сервиса – 50 миллисекунд, или 0,05 секунд. Получаем значение в 30. Попробуем включить механизм троттлинга, выставить это значение и посмотреть на результаты.

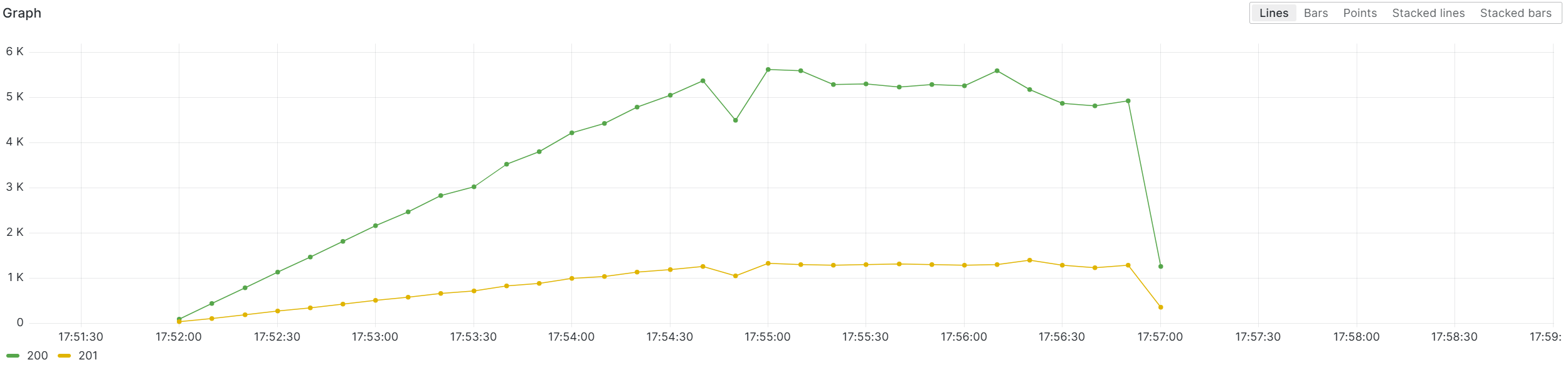


Рисунок 17 - график успешных HTTP ответов на запросы нагрузки в первом тесте на настройку троттлинга

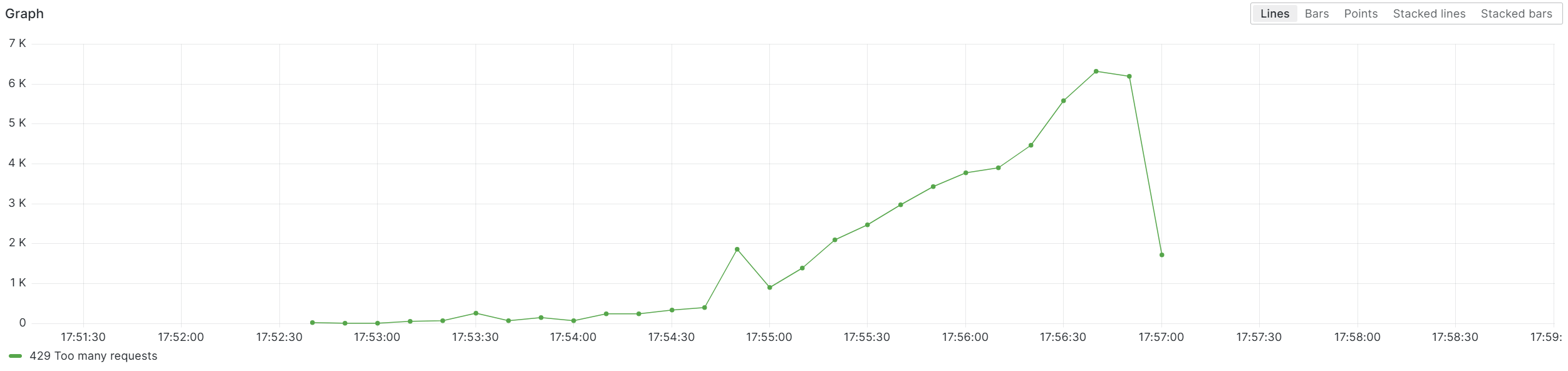


Рисунок 18 – график клиентских ошибок HTTP в первом тесте на настройку троттлинга

На графике клиентских HTTP ошибок заметны только ошибки с кодом 429 (Too Many Requests), выданные механизмом троттлинга



Рисунок 19 – график количества отклоненных механизмом троттлинга запросов в минуту в первом тесте на настройку троттлинга

A graph on a white surface

AI-generated content may be incorrect.

Рисунок 20 - график потребления ядер процессора в процентном соотношении от максимально допустимого в первом тесте на настройку троттлинга

Исходя из графиков видно, что сервис слишком рано начал отклонять пользовательские запросы – еще до близости к исчерпанию системных ресурсов. Первые отклоненные троттлингом запросы начали появляться в то же время, когда уровень потребления ядер процессора был на уровне всего-лишь 36%. Пробуем увеличить значение емкости в 2,5 раза и посмотрим на работу троттлинга в данном случае.

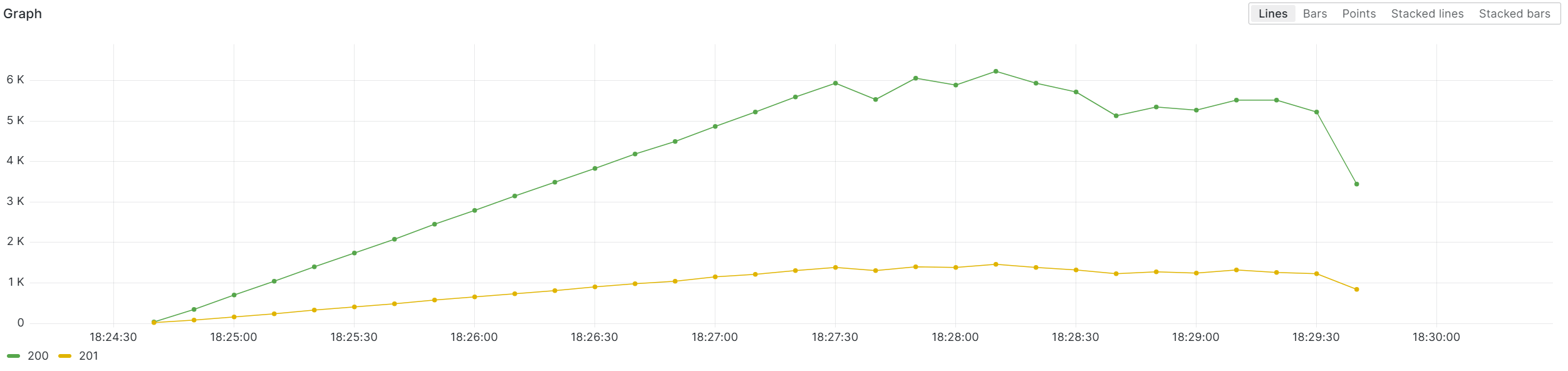


Рисунок 21 - график успешных HTTP ответов на запросы нагрузки во втором тесте на настройку троттлинга

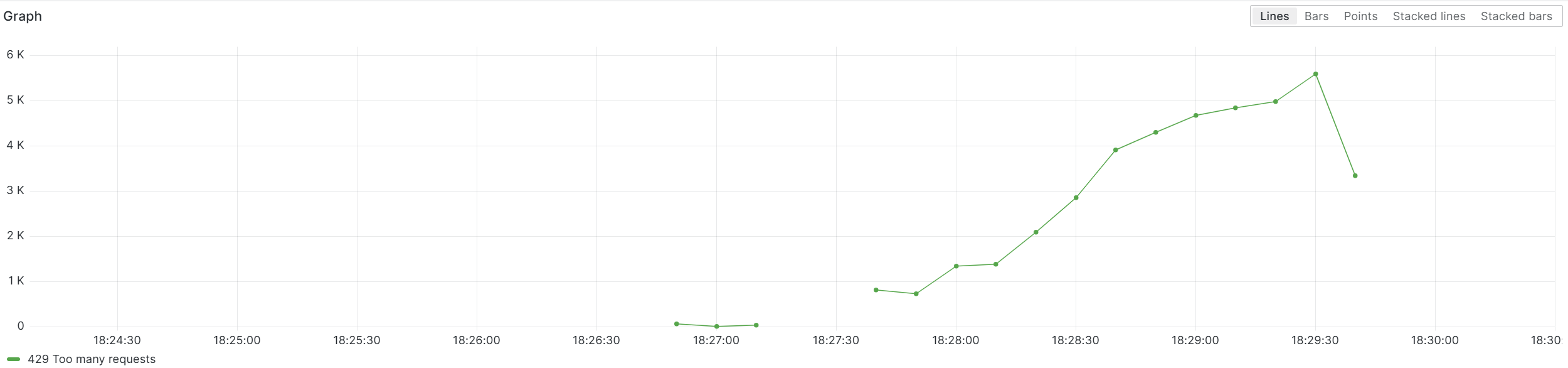


Рисунок 22 - график клиентских ошибок HTTP во втором тесте на настройку троттлинга

A graph on a white sheet

AI-generated content may be incorrect.

Рисунок 23 - график потребления ядер процессора в процентном соотношении от максимально допустимого во втором тесте на настройку троттлинга

По графикам видно, что механизм троттлинга начал отклонять первые запросы, когда уровень потребления ядер процессора сервисом был на уровне 85%. Количество отклоняемых запросов кратно увеличивалось по мере повышения нагрузки на сервис. Также нужно отметить, что при постоянно возрастающей за пределы пропускной способности сервиса нагрузке, количество успешных кодов ответа не падало и оставалось на близком к максимальной пропускной способности уровне. Из всего этого делаем вывод, что в результате тестирования удалось получить оптимальную настройку емкости троттлинга.

Осталось провести последний тест – всплеск нагрузки с возвратом к норме, в котором проверим, поможет ли настроенный механизм троттлинга справиться с резким подъемом нагрузки, в 2 раза превышающей максимальную пропускную способность и восстановить прежний уровень среднего времени ответа на запросы. Также посмотрим, сможет ли реплика выдержать 80% нагрузки от максимальной пропускной способности более продолжительное время, а поскольку такое значение, при экстраполяции на количество реплик на production окружении, больше чем в 2 раза превышает текущую нагрузку от реальных пользователей, то получим положительный ответ на вопрос о выдерживании сервисом двойной нагрузки от текущей.

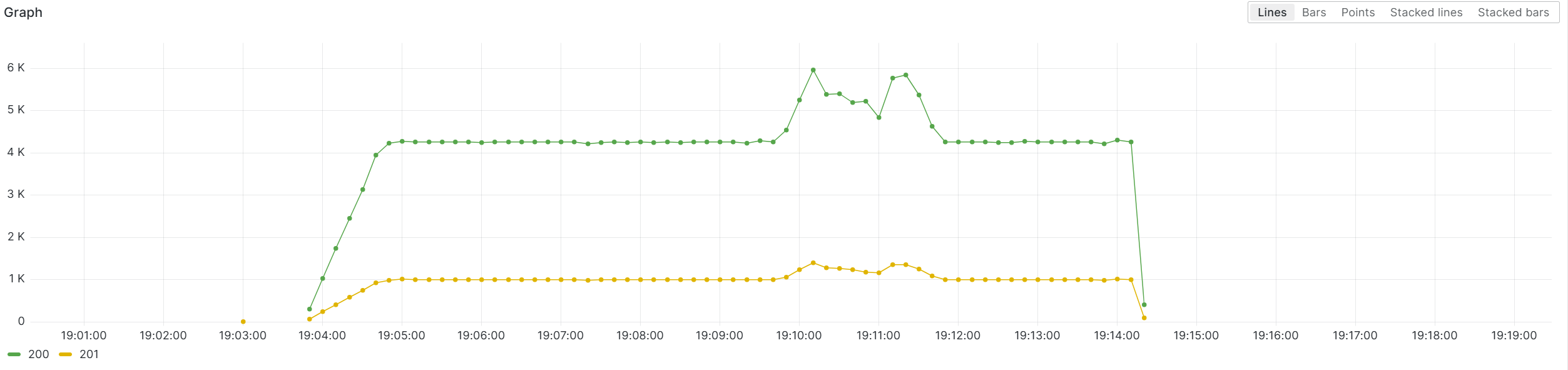


Рисунок 24 - график успешных HTTP ответов на запросы в тесте на всплеск нагрузки

A graph of a graph

AI-generated content may be incorrect.

Рисунок 25 - график клиентских ошибок HTTP в тесте на всплеск нагрузки

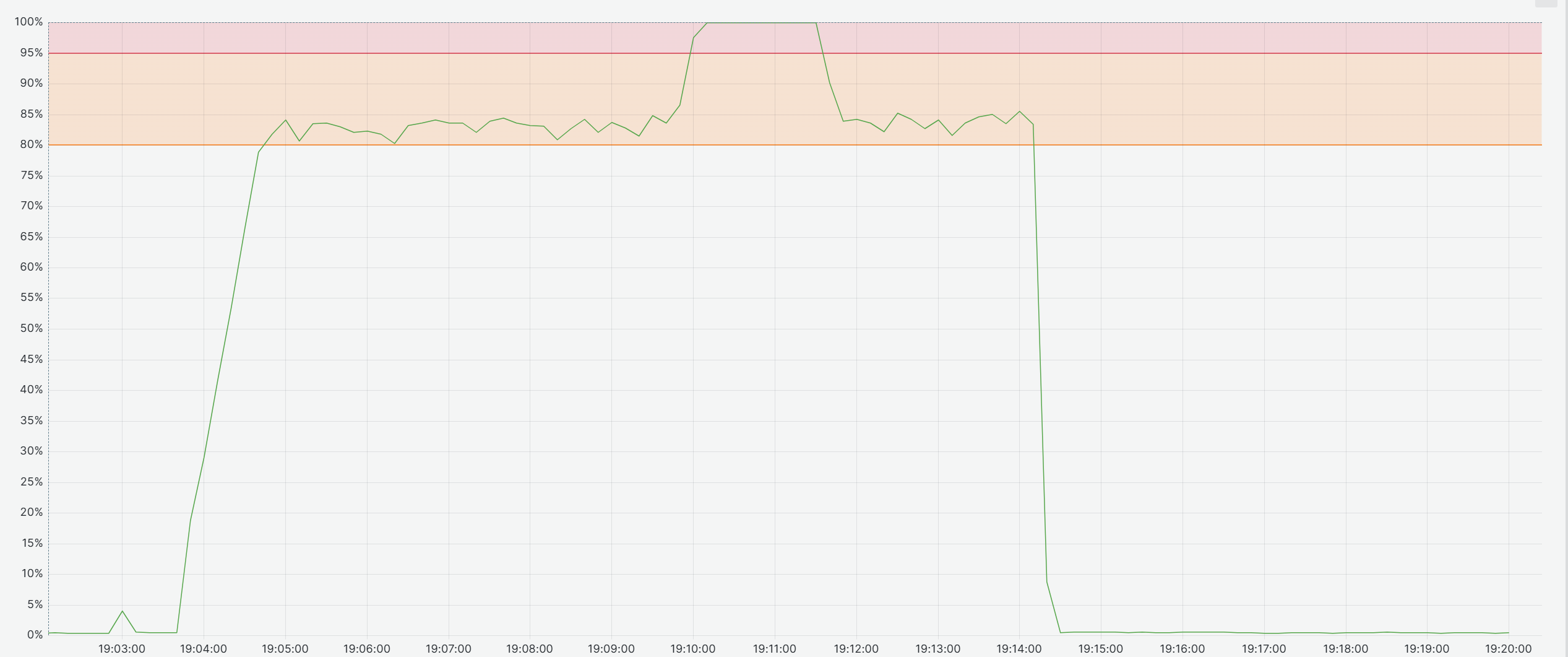


Рисунок 26 - график потребления ядер процессора в процентном соотношении от максимально допустимого в тесте на всплеск нагрузки

A graph showing a green line

AI-generated content may be incorrect.

Рисунок 27 – график времени ответа на запросы (95-й перцентиль)

Из результатов тестирования можно сделать вывод, что сервис выдерживает нагрузку в 80% от максимальной пропускной способности без деградации производительности, из чего, при экстраполяции результатов на production площадку, следует, что двойную пользовательскую нагрузку сервис сможет выдержать с хорошим запасом. Кроме того, результат тестирования показал, что после двухкратного от пропускной способности всплеска нагрузки сервис способен восстановить прежнее значение среднего времени ответа.

Первичное нагрузочное тестирование можно считать успешно пройденным.

# **ПЛАНИРОВАНИЕ РЕСУРСОВ ПОД НАГРУЗКИ**

После проведения нагрузочного тестирования следует определить, сколько ядер процессора, оперативной памяти и других системных ресурсов необходимо сервису для поддержания повышенной нагрузки. В нашем примере с первичным нагрузочным тестированием сервис показал, что для поддержания двойной нагрузки от текущей в production окружении всех системных ресурсов ему хватит. Но какие действия следовало бы предпринять, если бы по резульатам тестирования выяснилось, что сервису не хватает ресурсов для поддержания необходимого уровня нагрузки? Первым делом, определить вид такого ресурса: это может быть количество ядер процессора, количество оперативной памяти, размер пула подключений к базе данных, количество worker-потоков из пула потоков. Дальнейшие действия будут зависеть от конкретного ресурса, который истощается при нагрузке. Возьмем случай, когда никаких архитектурных ошибок в коде сервиса не имеется, ресурсы расходуются оптимально, но самого выделенного значения ресурса недостаточно - необходимо увеличить его количество. Рассмотрим случаи, в которых за необходимый ресурс команде разработки придется доплачивать: это физические ресурсы, такие как количество места на диске, количество оперативной памяти или ядер процессора. Предположим, что команда разработки располагает необходимыми финансами и может себе позволить увеличить показатели недостающего системного ресурса. В таком случае следует рассчитать необходимое желаемое значение для него. Например, при недостатке ядер процессора следует подбирать такое количество ядер, чтобы при целевой повышенной нагрузке до 100% потребления ядер процессора оставался хороший запас в 40-50%. После подбора желаемого показателя ресурса следует повторно провести нагрузочное тестирование с целью подтверждения пользы от внесенных изменений. Точное планирование показателей ресурсов, в том числе и методика подбора этих показателей, будет варьироваться от сервиса к сервису.