# 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.

# 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 сервиса

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 тестирование 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.