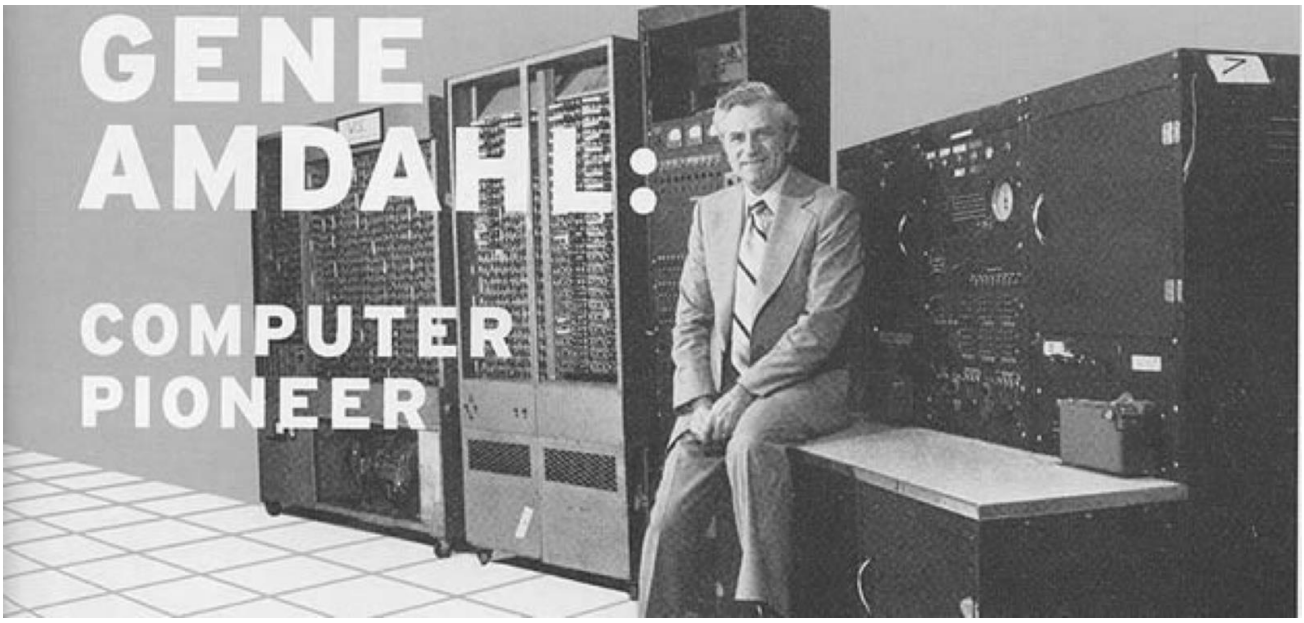
ozon{tech

Нагрузочное тестирование

План занятия

Закон Амдала



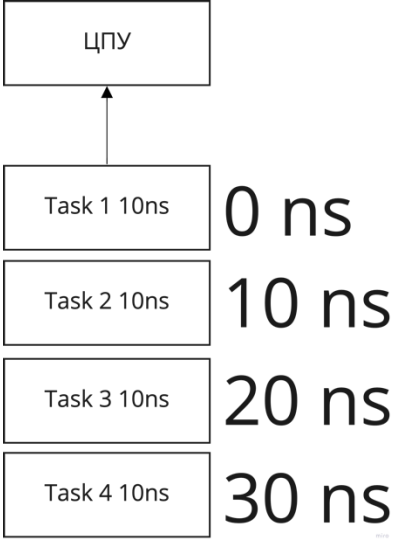
Закон Амдала звучит так:
В случае когда задача разделяется на несколько частей, суммарное время ее выполнения на параллельной системе не может быть меньше времени выполнения самого медленного ее фрагмента.
Из этого закона есть два следствия:

1. Не для всякой задачи имеет смысл наращивание числа процессоров в вычислительной системе.
2. С определенного момента добавление новых узлов в систему будет увеличивать время расчета задачи.

Это значит, что с добавлением большего числа процессов растет конкуренция за общие ресурсы.
Ими могут быть:

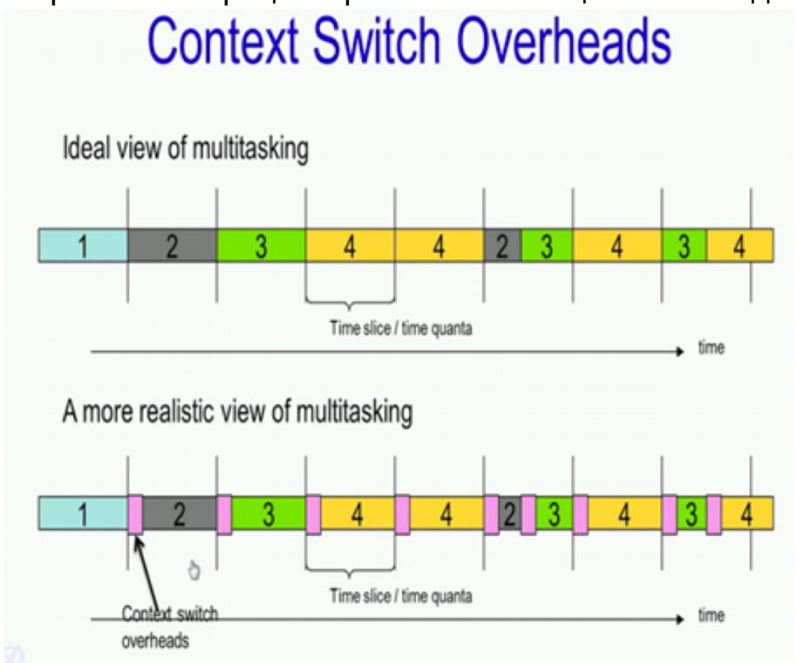
1. Вычислительные ядра CPU
    a. Context Switch — растет количество переключений контекста вычислений на процессоре, что приводит к неэффективному расходованию процессорного времени.
2. Шины данных.
3. Кеш.
4. Соединения с БД.
5. И т. д.

Представим себе, что процессор выполняет все задания последовательно.



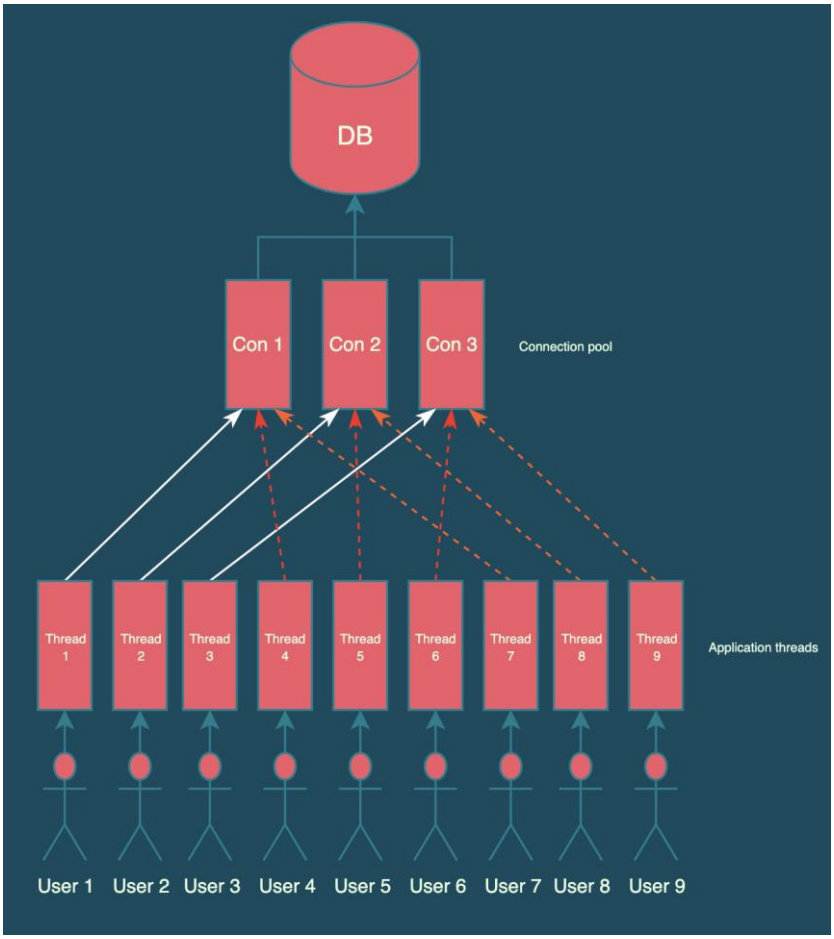
Тогда каждая следующая задача в очереди на выполнение будет исполняться дольше предыдущей, т. к. время выполнения предыдущих заданий добавляется к времени выполнения следующей в очереди задачи. Этот принцип работает везде, где есть очередь.

На реальном процессоре с вытесняющей многозадачностью происходит следующее:



Задачи на исполнение в очереди бьются на кусочки, и каждое ядро процессора обрабатывает все задачи одновременно. Однако на смену контекста задачи тратится дополнительное время: ведь нужно выгрузить в память результаты работы предыдущей таски, загрузить в память и кеши новую задачу (или результат предыдущего исполнения). Это называется Context Switch.
Чем чаще происходит смена контекста, тем больше накладных расходов тратится на него. И ЦПУ может вообще только этим и заниматься. Подобный показатель обычно можно найти в метриках. В хорошо спроектированных системах Contex Switch под нагрузкой не превышает 89–90 тыс. операций в секунду. В плохо спроектированных — может доходить и до 500–600 тысяч.
Для JVM Based языков, например, есть следующая рекомендация: размер тред пулов принято делать равным количеству ядер на сервер, умноженному на 2. При таком подходе достигается оптимальная производительность и утилизация потоков.
К Golang такой подход не применяется, т. к. там используется совершенно другой подход к многопоточности.

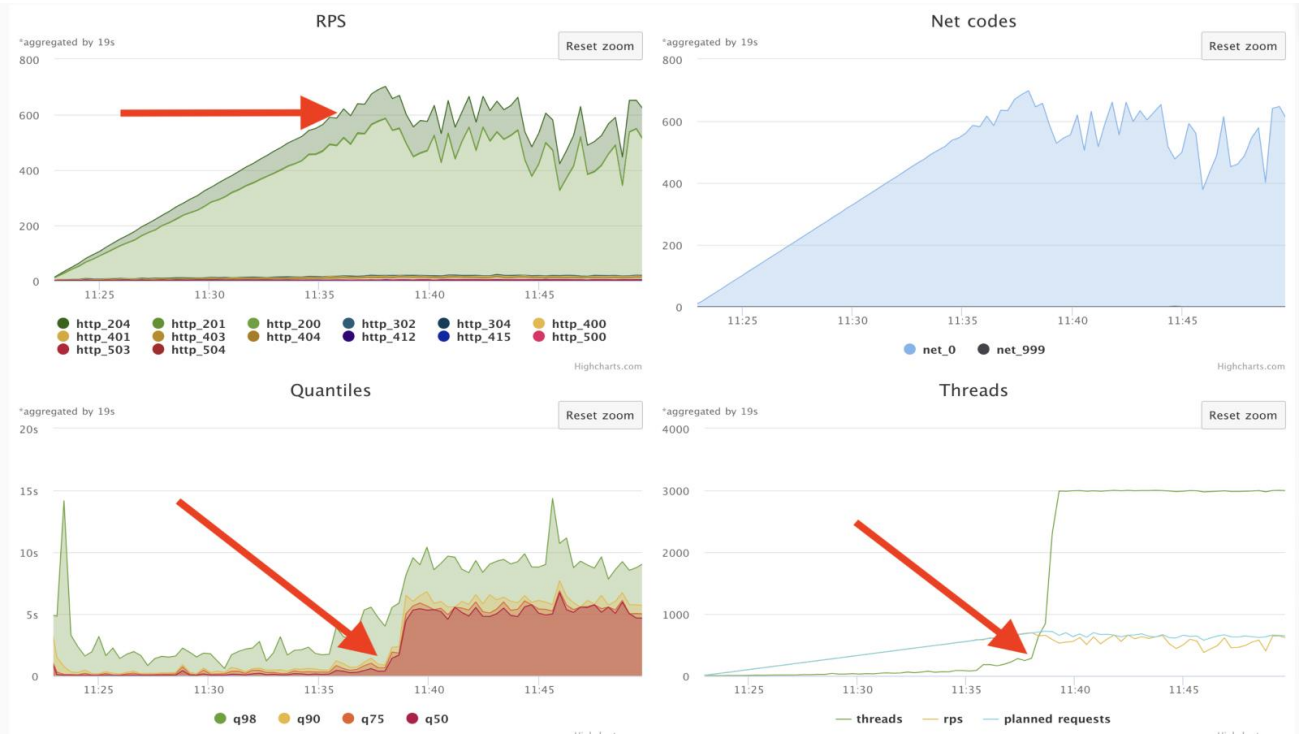
Пример того же принципа в другом контексте:



Предположим, у нас есть приложение и БД с пулом соединений. В нем только три коннекта. В приложении девять рабочих потоков, которые обслуживают пользователя. Допустим, три пользователя одновременно выполнили какой-то запрос. В этом случае три потока приложения займут все три соединения одновременно. Если случится, что в приложение придут уже шесть пользователей, следующие три будут ждать, пока не освободятся доступные коннекты к БД. Это пример так называемого бутылочного горлышка на соединениях к БД. В классических архитектурах клиент-серверных приложений именно это — наиболее частая причина деградации производительности.

Точка насыщения системы

Точка насыщения системы — уровень нагрузки, при котором дальнейшее наращивание числа пользователей ведет к увеличению времени отклика.



Посмотрите на левый верхний график: он отображает подаваемую нагрузку в запросах в секунду (RPS). Видно, что в какой-то момент рост RPS прекращается и начинаются пилообразные флуктуации.

На левом нижнем графике мы видим, что в момент, когда начинаются флуктуации, происходит рост времени отклика. Это как раз характерно для обнаружения точки насыщения системы.

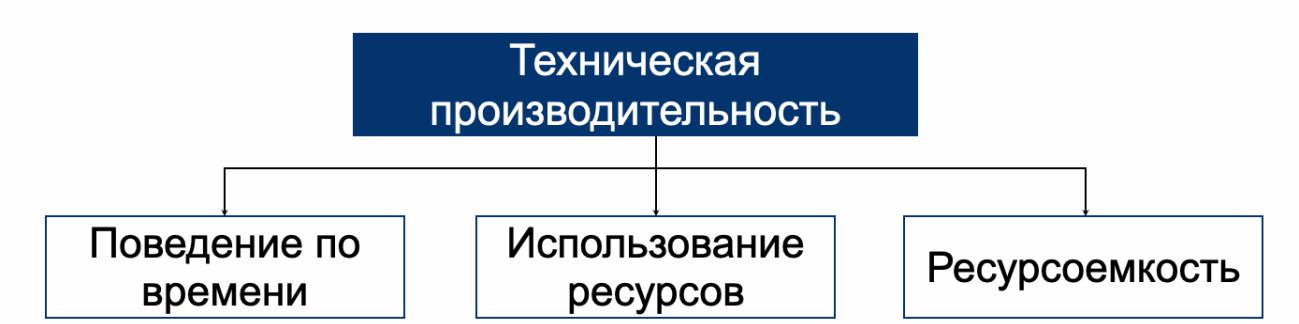
На правом нижнем графике понятно, что в момент начала роста времени отклика, подскочили потоки на генераторе нагрузки до максимального сконфигурированного уровня (3000). Дальнейший рост подаваемой нагрузки остановился, так как все свободные потоки стали заниматься ожиданием ответов от сервера.

Точка насыщения указывает на исчерпание какого-либо ресурса.

Что такое производительность

Согласно ИСО 25010, производительность определяется как нефункциональная характеристика качества с тремя составляющими:

- Поведение по времени.
- Использование ресурсов.
- Ресурсоемкость.



Поведение по времени: эта сторона посвящена исследованию способности компонента или системы отвечать на действия пользователей или других систем в течение заданного времени и при заданных условиях. Когда мы говорим «нагрузочное тестирование», чаще всего подразумеваем оценку поведения системы по времени.

Использование ресурсов: эта характеристика посвящена исследованию особенностей потребления ресурсов нашей системой. Таких как ЦПУ, объем оперативной памяти, место на диске, пропускная способность сети. Акцент на это исследование появляется, если принимается решение о его необходимости и существуют определенные риски, связанные с исчерпанием какого-либо ресурса.

Пример: если ваша система использует Redis и работает с большими объемами данных, имеет смысл смотреть на потребление ресурсов Redis во время тестов. Может возникнуть ситуация, при которой резко вырастет количество запросов к Redis или объем потребляемой памяти. Это может привести как к деградации производительности, так и к простою в случае падения.

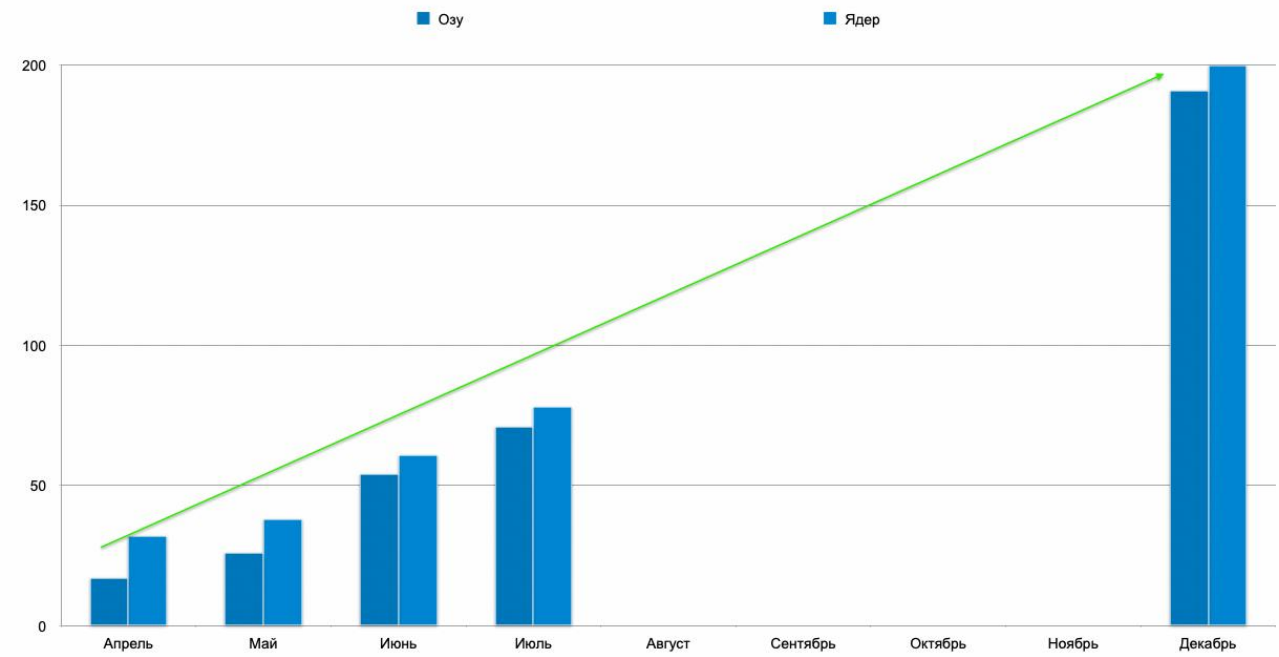
Ресурсоемкость — способность системы справляться с нагрузками при возрастающих объемах данных.

Пример: производительность одной и той же системы сразу после запуска и через год работы может отличаться драматически. С увеличение срока жизни проекта растут объемы данных, размеры индексов и т. д. Следует проводить Capacity planning, чтобы не искать срочно в бюджете средства на новый сервер БД, а то старый вдруг перестал справляться.

Поскольку тестирование производительности должно учитывать эти три различные характеристики качества, оно нередко оказывается экспериментальной деятельностью, что необходимо для проведения замеров и анализа специфических параметров системы. Подобный процесс может носить итеративный характер и использоваться для помощи в системном анализе, разработке и реализации, позволяя принимать решения в масштабе архитектуры и формировать ожидания заинтересованных лиц.

Capacity planning

Понятие Capacity planning тесно связано с ресурсоемкостью системы.



В Ozon считают модель роста нагрузки исходя из количества заказов и RPS на сайт. На основе планов маркетинга и бизнеса выставляется планка по требованиям по RPS на отдельные сервисы. Под эти требования строится бюджет, закупается железо и проводятся нагрузочные тесты. Подробнее можете почитать в статье нашего CEO на Хабре: [Пережить распродажу на Ozon: хайлоад, сковородки и 38 инфарктов]

Риски

Решать, проводить тестирование или нет и какое именно, рекомендуется на основе анализа рисков вашей системы. Мы рекомендуем проводить небольшие сессии с командой, например, раз в квартал, на которых надо обсуждать текущие и будущие риски вашей системы. Пример таблички с анализом:

Риск	Вероятность	Влияние на продукт	Как снизить
Деградация времени отклика из-за наплыва клиентов во время распродажи	Высокая	Негативный пользовательских опыт; Репутационные потери;	Провести НТ с релевантной нагрузкой.
Утечка памяти	Средняя	Потеря стабильности системы; Репутационные потери;	Провести Тестирование стабильности

Совокупность нефункциональных требований и тестов для их валидации и решений по снижению рисков и будет вашим тест-планом тестирования производительности.

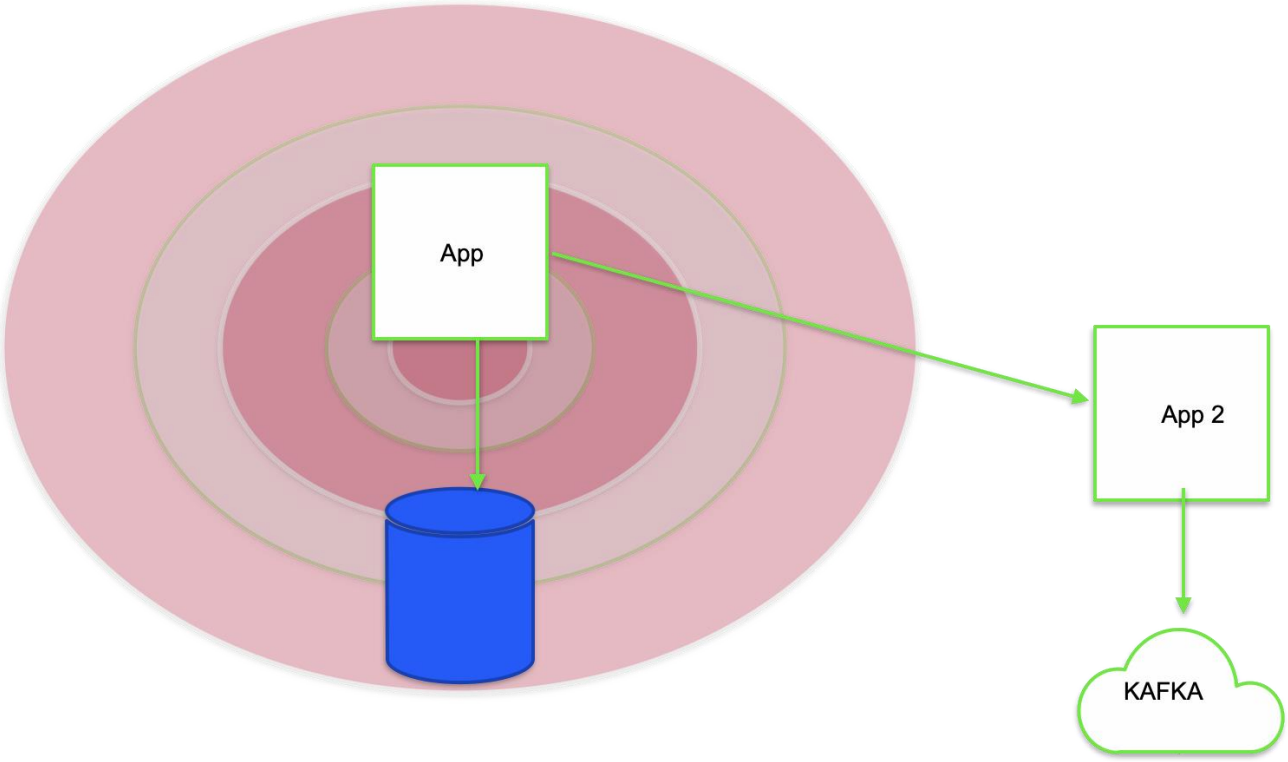
Основные принципы тестирования производительности

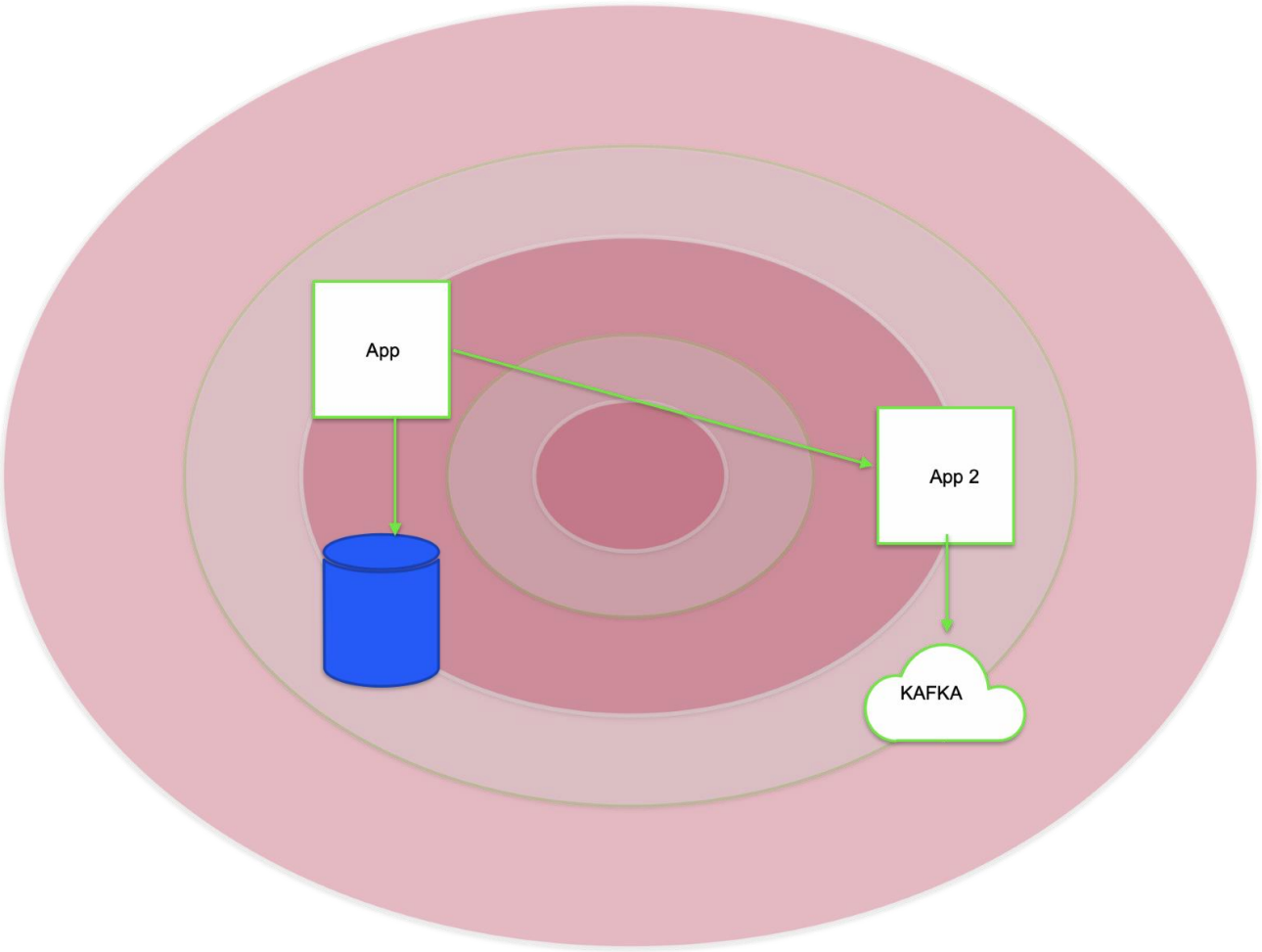
1. Тесты должны быть воспроизводимыми. Нагрузочное тестирование — очень дорогой процесс. Чтобы оно было эффективным, необходимо, чтобы тесты, работающие в одинаковых условиях, давали идентичный результат. Нет ситуации хуже, чем раз за разом перезапускать один и тот же тест и получать каждый раз совершенно разный результат. Причина часто в том, что какой-то элемент системы отличается от продакшна. Либо же ваш сервис зависит от какой-то внешней интеграции, производительность которой вы не можете контролировать. В этом случае, возможно, стоит использовать специализированные заглушки для внешних интеграций, чтобы лучше контролировать среду.
2. Тесты должны быть результативными. Тестирование производительности (дорогое удовольствие), поэтому оно должно быть направлено на получение результата. Нет ничего хуже, чем заниматься тестирование ради тестирования. Так что каждый ваш тест должен иметь цель и смысл.
3. Тесты должны соответствовать ожиданиям заинтересованных лиц. Очень часто на результаты тестов производительности смотрит менеджмент, ведь ими в том числе закрываются риски бизнеса. Поэтому крайне важно не просто проводить тестирование производительности, но и ориентироваться на то, что хотят выяснить заинтересованные лица. Пример: команда смежного проекта пришла к с новостью, что хочет интегрироваться с вашим сервисом, и собирается нагружать его с частотой N. В этом случае проведите дополнительное исследование, чтобы понять, выдержит ли ваш сервис SLA по нагрузке в новых условиях.
4. Тесты должны быть разумными по бюджету. Нагрузочное тестирование — увлекательный процесс, можно потерять связь с реальностью. Поэтому всегда помните о бюджете. Иногда имеет смысл собирать точную копию продакшна, а это могут быть сотни серверов. Так следует поступать, если риски от непроведения теста перевешивают финансовые потери от возможных дефектов производительности.
5. Тесты должны проводиться на релевантной копии системы, по возможности приближенной к проду. Если ваша система на более слабом стенде держит уровень нагрузки продакшна, тогда нагрузочное тестирование возможно. Если нет, результаты от более низкой нагрузки не смогут выявить большинство проблем потери производительности, кроме самых явных. Такое тестирование экономически невыгодно.

Объект тестирования

Как и в обычном тестировании, объект тестирования — отдельный компонент, сервис, набор сервисов, система или система систем, которые принимаются как единый объект тестирования. Что является объектом тестирования, определяется на этапе подготовки к тестированию.

Пример:

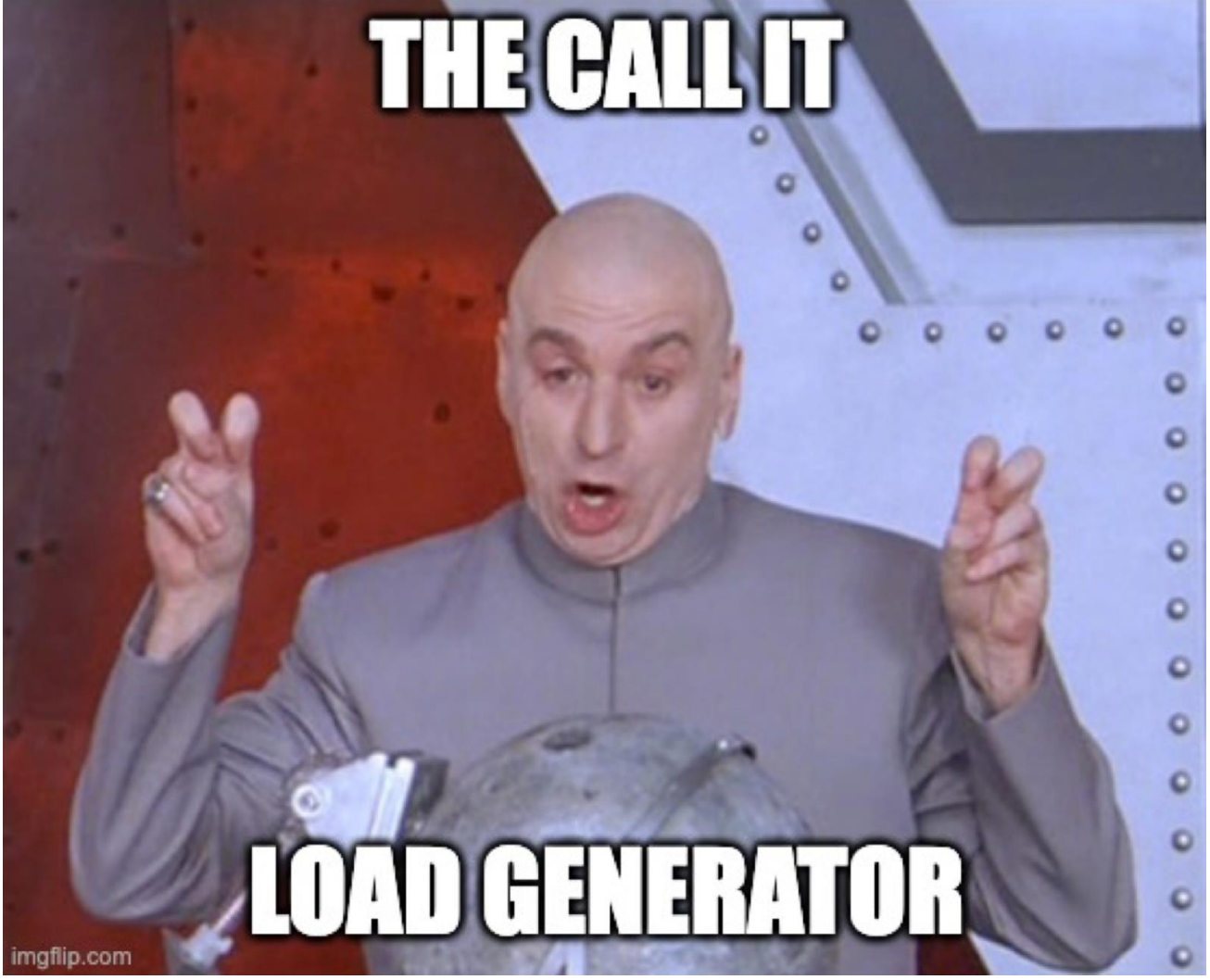



App
App 2
KAFKA

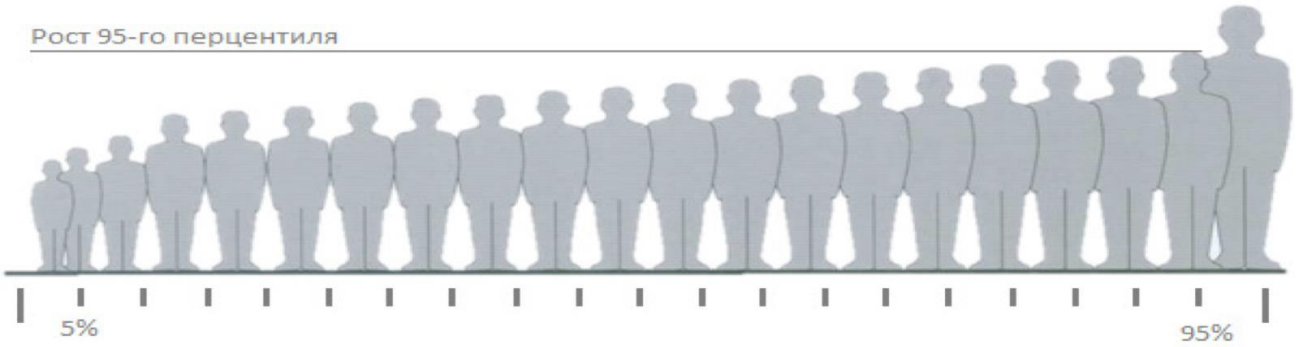
Генератор нагрузки

Генератор, используя интегрированную среду разработки, редактор скриптов или набор инструментов, способен создавать и запускать множественные экземпляры клиентов, имитирующих поведение пользователя согласно заданным рабочим профилям. Создание нескольких экземпляров подобных клиентов за короткие промежутки времени вызовет нагрузку на тестируемую систему. Генератор как создает нагрузку, так и собирает метрики для последующего создания отчетов.

Задача тестов производительности — имитировать реальный ход событий настолько, насколько это практически возможно. Часто это означает, что потребуются пользовательские запросы, поступающие из разных мест, а не только из одного. Окружения, построенные с несколькими точками создания нагрузки, следует распределить так, чтобы источники нагрузки не находились в одной сети. Это обеспечивает реалистичность тестирования, хотя иногда искажает результаты, если промежуточные сетевые сегменты создают задержки.


THE CALL IT
LOAD GENERATOR
imgflip.com

Перцентиль

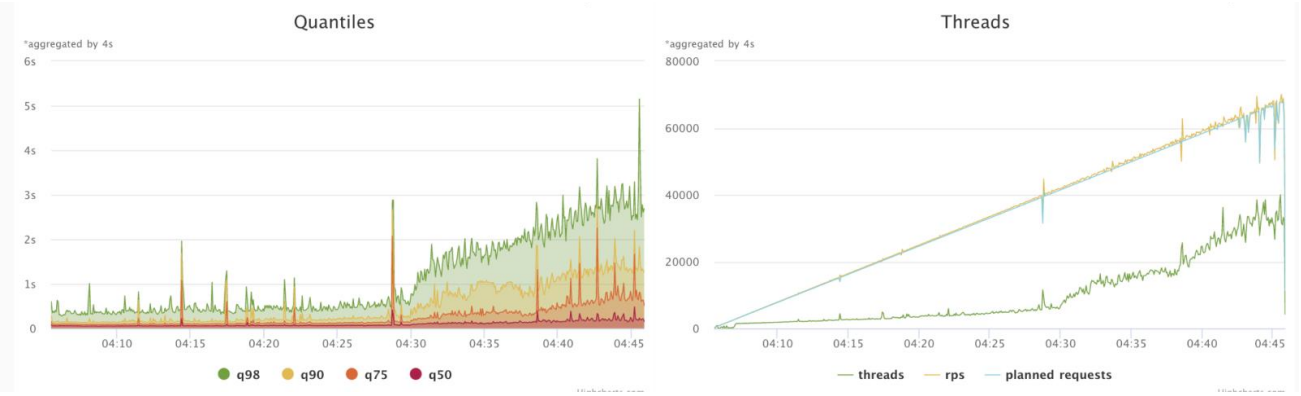


Это статистический термин, используемый в нагрузочном тестировании. При выполнении тысяч запросов в систему всегда могут быть случайные отклонения, которые не влияют на общие результаты. (Если, конечно, контекст тестирования это позволяет) В случае нагрузочного тестирования их принято трактовать так: время отклика системы по запросу на N-м перцентиле меньше некоторого числа. Например, когда говорят, что при нагрузке в 100 RPS время отклика по 95 перцентилю составляет 200 мс, это значит, что 95 запросов из 100 каждую секунду отрабатывают быстрее 200 мс. И есть еще пять запросов, которые отрабатывают дольше 200 мс.

Поэтому при обработке результатов обычно смотрят на такие показатели, как макс. (т. е. максимальное время отклика) и 50 перцентиль (или медиана, т. е. половина всех запросов за промежуток времени).

Степень критичности и нефункциональные требования системы влияют на то, по каким критериям следуют обрабатывать результаты. Для Mission critical систем может использоваться 99 или 98 перцентиль. Чаще всего используется 95 перцентиль.

Открытая модель нагрузки



- Количество пользователей не ограничено.
- Характерна для интернета.
- Самый «жесткий» вариант нагрузки.
- Замедление сервиса будет приводить к росту нагрузки на сервис, т. к. пользователи никуда не уйдут и начнут нажимать Refresh.

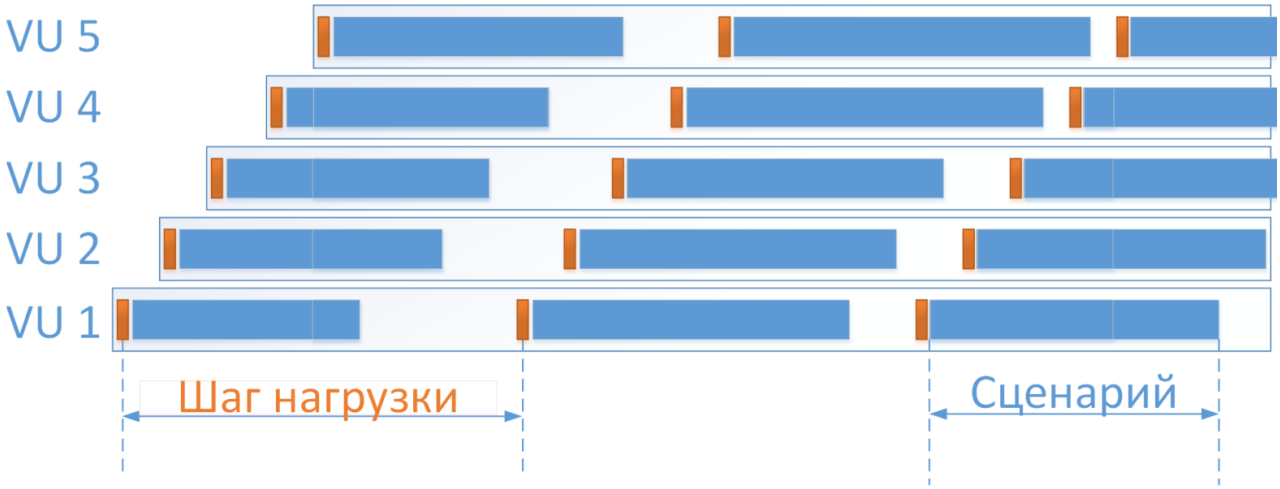
Закрытая модель нагрузки

- Количество пользователей ограничено.
- Характерна для внутренних сервисных продуктов — больших и не очень.
- Увеличение времени отклика ведет к уменьшению нагрузки на сервис, так как новые пользователи не придут.

Сценарный подход

При составлении модели нагрузки часто используется обозначенный в заголовке подход. Его суть в том, что для построения модели нагрузки воспроизводятся типовые пользовательские сценарии. Этот подход непросто поддерживать, поэтому применять его следует только тогда, когда без него обойтись нельзя. Пример: в вашей системе реализована сложная бизнес-логика. По этой логике для отправки следующего запроса из списка нужны данные из ответа предыдущего запроса..

- Сложно высчитать итоговый RPS.
- Оперирует понятием *сценарий в секунду*.
- Обычно слабо приближен к продакшну.
- Стоимость приближения имеет ценно-временное выражение: чем ближе сценарий к проду, тем он дороже.



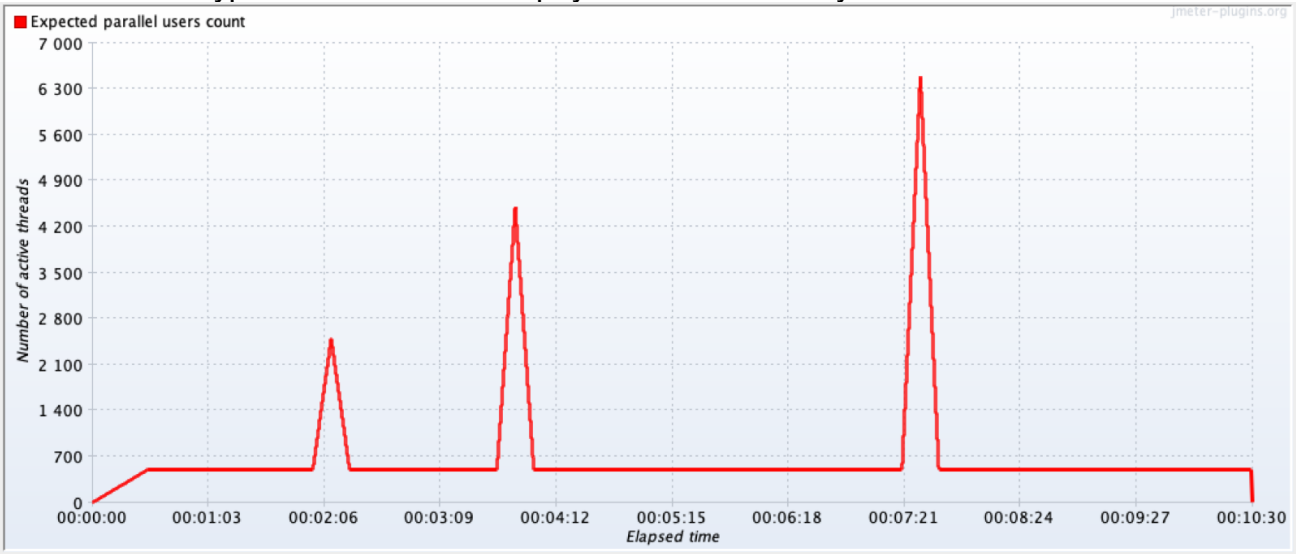
Данный подход использует такие понятия, как шаг нагрузки и виртуальный пользователь. В техническом плане виртуальный пользователь — поток, треда или горутина. Для вычисления итогового RPS обычно используется модель в Excel. Примеры можно найти в интернете.

RPS-подход

Этот подход исторически более простой. Технически почти все модели нагрузки можно свести к RPS-подходу. Возможны смешанные варианты, когда используется RPS-подход и сценарии. Например, нагрузка считается в количестве стартовавших сценариев в секунду. Такое характерно, например, для телеком-домена: количество новых звонков в секунду, когда каждый звонок имеет фиксированный набор событий.
Данный подход оперирует понятием “количество запросов в единицу времени”.
Запросы лучше всего подготавливать заранее, чтобы не жечь ЦПУ в процессе теста и получать статистически более достоверные результаты.
Чаще всего данные для запросов можно брать прямо из логов. На крайний случай можно попросить ваших администраторов включить расширенное логирование на NGINXах для сбора данной статистики.
.

Виды тестирования производительности

1. **Нагрузочное тестирование** исследует способность системы обрабатывать растущие объемы ожидаемой реалистичной нагрузки как следствие запросов на совершение транзакций контролируемым числом одновременно работающих пользователей или процессов. (ISTQB)
2. **Стрессовое тестирование** исследует способность системы или компонента обрабатывать пиковые объемы нагрузки на пределе или за пределами ожидаемой или предусмотренной спецификацией пропускной способности. Стрессовое тестирование также используется для оценки работоспособности системы в условиях уменьшенной доступности ресурсов, таких как вычислительные мощности, полоса пропускания сети, память.
3. **Тестирование масштабируемости** исследует способность системы удовлетворять будущим требованиям эффективности, которые могут превосходить настоящие требования. Цель таких тестов — способность системы расти в объеме (например, с увеличением количества пользователей, объемов хранимых данных, не нарушая текущие требования производительности. (ISTQB)
4. **Тестирование производительности при всплесках нагрузки** исследует способность системы правильно реагировать на внезапные всплески нагрузки до пикового уровня и вернуться в устойчивое состояние.



5. **Тестирование стабильности** исследует устойчивость работы системы на протяжении временного периода, достаточного для ее эффективного использования. Это тестирование позволяет убедиться, что в системе нет проблем, связанных с доступным объемом ресурсов (например, утечек памяти, соединений к БД, пулов потоков выполнения), которые могли бы со временем привести к ухудшению производительности и/или возникновению аварий.
6. **Тестирование ресурсоемкости** определяет максимальное количество пользователей и/или транзакций, которое система может поддерживать, обеспечивая при этом выполнение установленных требований производительности. Цели также могут быть сформулированы в отношении объемов данных, полученных в результате операций.

Регрессионное нагрузочное тестирование (НТ)

Если решаете проводить нагрузочные тесты каждый релиз, есть смысл проводить регрессионное нагрузочное тестирование. К такому выводу обычно приходят, если сервис имеет серьезные риски, связанные с потерей производительности, и его поломка влечет за собой финансовые и репутационные потери для компании.
Результаты таких тестов надо обрабатывать с точки зрения:

- Сравнивать качество подаваемой нагрузки.
- Сравнивать по метрикам производительности запросов.
- Сравнивать по метрикам производительности сервисов.

По результату таких сравнений - принимается решение о выпуске релиза. Либо же релиз отправляется в доработку.
Главная проблема — отсутствие нормального инструментария. Grafana + Python – наш любимый швейцарский нож.

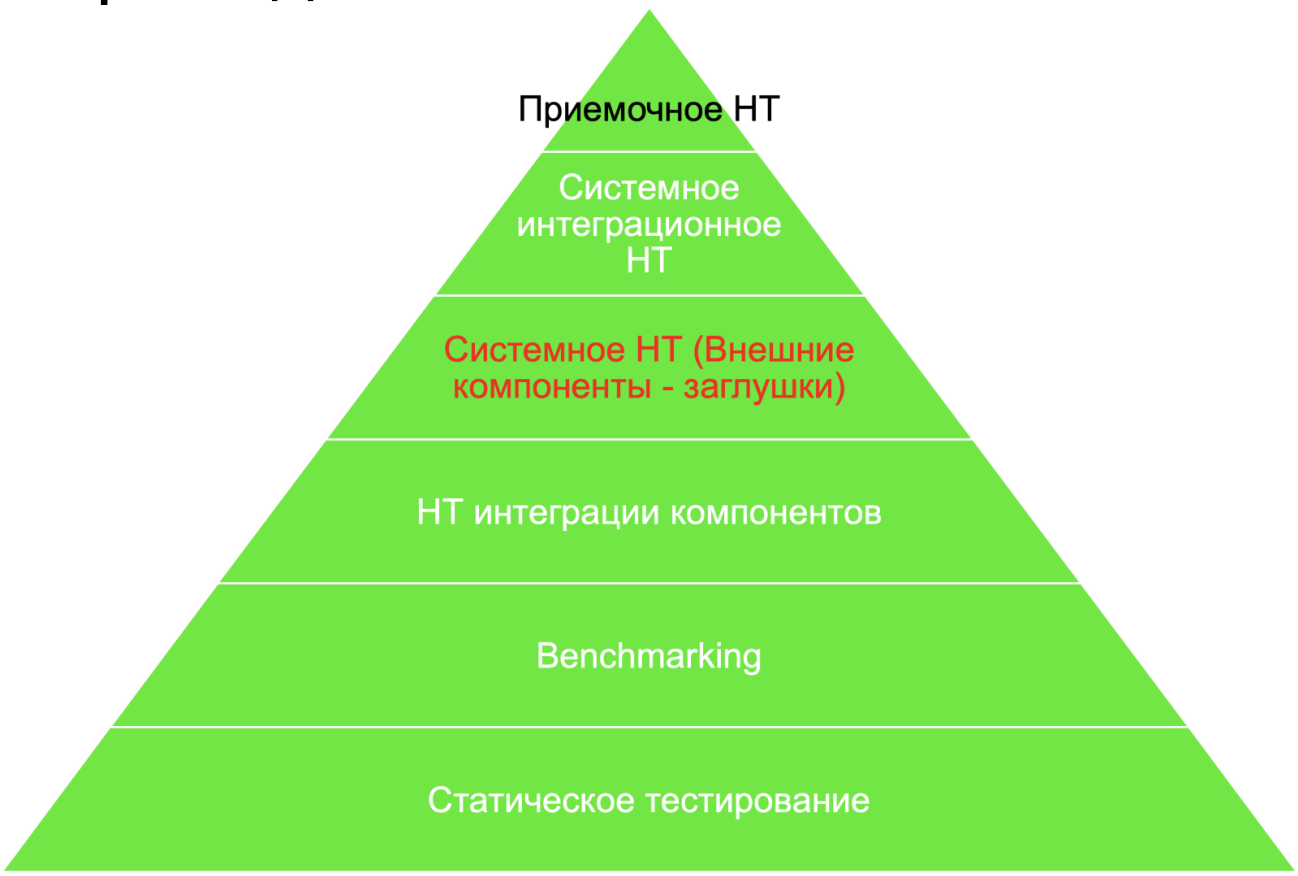
Статические активности НТ

- Рецензирование требований с аспектом на производительность.
- Рецензирование схем БД / хранимых процедур и т. д.
- Рецензирование архитектуры системы и сетевой инфраструктуры.
- Рецензирование критических участков кода.

Динамические активности НТ

Это процесс, при котором запускается какой-либо код.

Пирамида НТ


Приемочное НТ
Системное интеграционное НТ
Системное НТ (Внешние компоненты - заглушки)
НТ интеграции компонентов
Benchmarking
Статическое тестирование

Распространенные виды проблем производительности и их причины

1) Медленный отклик под любой нагрузкой.
   Возможные причины:
   - Архитектура.
   - Фоновая нагрузка.
   - Слишком слабый стенд.
   - Нехватка ресурсов сети.
2) Медленный отклик под умеренной и значительной нагрузкой.

Возможные причины:

- Исчерпание какого-то ресурса (см. закон Амдала).
- Медленный отклик внешней системы.

3) Деградация времени отклика с течением времени.

Возможные причины:

- Утечки памяти.
- Непредусмотренное разрастание БД.
- Фрагментация диска.
- Растущая нагрузка на сеть (если мы гоняем полные списки туда-сюда).
- Исчерпание места на файловом хранилище.

4) Некорректная или аварийная обработка ошибок под большой нагрузкой.

Возможные причины:

- Нехватка пулов ресурсов.
- Недостаточный размер очередей.
- И т. д.

Типичные метрики нагрузочного тестирования

- Метрики НТ.
  - Время отклика.
  - Интенсивность транзакций.
  - Коэффициент масштабируемости.
  - И др.
- Метрики системы.
  - Утилизация ресурсов (ЦПУ, РАМ, СЕТЬ и т. д.).
  - Количество ошибок.
  - Фоновая утилизация общего ресурса.
  - И др.
- Бизнес-метрики
  - Время обработки операций.
  - И др.
- USE/RED
- CLI/SLA

Обзор инструментов НТ

Yandex Tank

Комбайн на Python, созданный во времена ДО Docker. Задача — забрать на себя всю оснастку вокруг генераторов нагрузки. Работает как универсальный комбайн, который позволяет сделать единый формат управления различными генераторами нагрузки.
https://github.com/yandex/yandex-tank

Jmeter

Довольно распространенный инструмент нагрузочного тестирования. Имеет большое комьюнити и широко применяется в очень большом количестве проектов. Иногда достаточно быстр, но имеет большой порог входа: необходимо потратить время на изучение принципов и подходов. Есть GUI, но он весьма запутанный. В Ozon применяется, но редко и только для создания сложных сценариев нагрузки.
Большие сценарии крайне сложно поддерживать: они превращают тебя в того, кто прыгает по веткам xml-деревьев. Это полностью OpenSource инструмент.
Классический инструмент для сценарной нагрузки и закрытой модели. Гибкий, но в силу этого довольно сложно настраиваемый. Легко писать свои плагины на java.
https://jmeter.apache.org/
https://jmeter-plugins.org/
https://github.com/abstracta/jmeter-java-dsl

BFG

Генератор нагрузки на Python. Используется в некоторых проектах нашей компании. Поддерживает низкий уровень параллелизма — до 100 тредов.

Phantom

Пушка на C++ авторства Яндекса. Поддерживает только HTTP 1.1. В Ozon не используется.

Pandora

Основной генератор нагрузки Ozon. Написан на Golang в Яндекс. Обеспечивает весьма хорошую производительность. Сотни тысяч RPS. При определенном подходе позволяет писать сценарные тесты. Работает по принципу *1 запрос — 1 патрон*.
Плюсы:

- Быстрый, сравним с Phantom&.
- Кросс-платформенный.
- Любые go-библиотеки.
- Под свои нужды можно написать свои пушки.

Минусы:

- Для сложных кейсов надо писать свои пушки на Golang.
- Местами запутанный и не совсем оптимальный код.

https://github.com/yandex/pandora

Заключение

- Нагрузочное тестирование - большая тема
- Вы получили основу, которая потом поможет легче и быстрее разобраться в этом вопросе
- Если остались вопросы - пишите в чат
- До встречи в Озоне.

Литература

- ISTQB по нагрузочному тестированию: https://www.rstqb.org/ru/istqb-downloads.html?file=files/content/rstqb/downloads/ISTQB%20Downloads/ISTQB_CTFL_PT_Syllabus_2018_Russian.pdf&cid=28951
- Про пассинги: https://loadtestweb.info/2017/08/23/pacing/
- Про Яндекс-Танк: https://www.youtube.com/watch?v=1idebTeMTqY
- Анализ результатов нагрузки: https://www.youtube.com/watch?v=gws7L3EaeC0
- Про пандору: https://www.youtube.com/watch?v=lkusMkIniq0
- Gatling: https://www.youtube.com/watch?v=2wWiud1A7BM
- https://gitlab.com/tinkoffperfworkshop
- По Посгресу: https://www.youtube.com/watch?v=IoKwUNmU-0k
- По Jmeter: https://www.youtube.com/watch?v=rQCspOA30Bc
- Статья про подготовку к сезону: https://habr.com/ru/company/ozontech/blog/664472/
- Методологии метрик: https://www.itsumma.ru/knowledges/blog/alerts
- SLI/SLO: https://habr.com/ru/company/proto/blog/538966/
- Jmeter: https://jmeter.apache.org/
- Jmeter-plugins: https://jmeter-plugins.org/
- Yandex Tank: https://yandextank.readthedocs.io/en/latest/
- BFG: https://github.com/yandex-load/bfg

Чаты

- https://t.me/qa_load — тут помогут с вопросами по нагрузке
- https://t.me/metrics_ru — тут все о метриках