Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский университет
«Высшая школа экономики»

Факультет компьютерных наук
Основная образовательная программа
Прикладная математика и информатика

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Программный проект на тему

"Тестирование логов компонент поиска"

Выполнил студент группы 185, 4 курса,
Сафиуллин Зуфар Гумарович

Руководитель ВКР:
Кандидат технических наук, доцент, Сухорослов Олег Викторович

Соруководитель ВКР:
Руководитель группы инфраструктуры обработки данных, Агапитов Василий Владимирович

Москва 2022

Содержание

1 Введение

В последнее время микросервисная архитектура очень популярна для использования в высоконагруженных системах. По сравнению с монолитной схемой она имеет массу преимуществ, таких как скорость интеграции, безопасное внедрение изменений в отдельных частях сервиса. Кроме того, благодаря модульности разработка может разбиваться по зонам ответственности, что сильно облегчает погружение новых разработчиков в продукт.

Однако этот подход имеет ряд недостатков. Одним из них является сложность стабильного взаимодействия микросервисов между собой. Контракт между двумя компонентами может быть легко разорван при развертывании любой из них. Для проверки корректности работы сервиса с такой архитектурой необходимо иметь тесты, проверяющие взаимодействие разных частей внутри всего пайплана. В этой работе рассматривается инфраструктура для тестирования логов, позволяющая эмулировать пользовательские действия и прогонять запросы по всей последовательности микросервисов.

1.1 Актуальность и значимость

В микросервисной архитектуре ошибка при взаимодействии любой пары компонент может привести к цепочке сбоев в связанных частях и, как следствие, к потере данных, недоступности сервиса и несогласованности пользовательских данных. Если при этом сервис является бизнесом, всё вышеперечисленное может вести к денежным потерям. Известным подходом к решению этой проблемы являются интеграционное и end-to-end тестирования. С помощью них можно выявлять ошибки на стадии разработки.

Кроме того, если углубиться в техническую составляющую, разработка большого проекта требует значительных временных затрат и ресурсов на написание тестов. Написание фреймворка, позволяющего эмулировать пользовательский сценарий и прогонять его по последовательности сервисных компонент, в разы упрощает разработку тестов. Таким образом, разработчику

будет достаточно написать тест в одном файле, не задумываясь о том, как и где он будет запускаться.

1.2 Краткое описание работы

Рассмотрим систему, состоящую из микросервисов, которые участвуют в логировании. Необходимо уметь проверять, не нарушается ли логирование компонентами по отдельности и в связке.

В этой работе мы рассмотрим данную проблему на примере сервиса Яндекс.Поиск. Мы никак не опираемся на то, что сервис является поисковым. Аналогичная инфраструктура может быть развернута для любого сервиса, части которого вносят вклад в логи.

На данный момент, сервисы поиска Яндекса связаны так, что каждая компонента не логирует данные отдельно. Большая часть из них пропускает данные через «верхний» поиск, который пишет логи. Поэтому при релизе компонента не может независимо проверить, что она не поломает логирование.

Поскольку все компоненты являются микросервисами, малейшее изменение может помешать приёмочной стороне. В данной работе реализуется инфраструктура для тестирования логов, представляющая собой одну задачу-тест. Результатом выполнения этого теста является отчёт по изменению метрик в исходном состоянии сервиса и при замене одной из компонент на новую релизную версию.

1.3 Цели и задачи

Целью работы является написание инфраструктуры для интеграционного и end-to-end тестирования логов в сервисе Яндекс.Поиск.

Были поставлены следующие задачи:

- Исследовать существующую архитектуру Яндекс.Поиска

- Реализовать hermione тесты на основе selenium, обстреливающие поисковую страницу
- Реализовать процесс, откладывающий тестовые логи в отдельные таблички
- Реализовать процесс, запускающий A/B-тесты по полученным таблицам
- Написать sandbox задачу, запускающую пайплайн для двух кодовых версий компоненты и сравнивающую в конце соответствующие таблицы с помощью A/B-тестов.
- Интегрировать написанный тест в предрелизный этап для какой-либо компоненты

1.4 Структура работы

В главе 2 приводится обзор существующих решений задачи тестирования систем с микросервисной архитектурой.

В главе 3 подробно описывается архитектура Яндекс.Поиска, затрагивающая все участвующие в нашей работе микросервисы.

Глава 4 содержит детали реализации инфраструктуры для тестирования логов.

В заключительной главе 5 подводятся итоги и описываются направления дальнейшей работы.

2 Обзор существующих решений

Тестирование систем с микросервисной архитектурой это достаточно сложная задача. В силу архитектурных особенностей, проблемы могут возникать из-за временных выпадений сервисов, сетевых проблем и, разумеется, ошибок

в коде. В отличие от монолитной схемы, необходимо тестировать компоненты изолированно по отдельности, а также вместе в различных комбинациях. Кроме того, необходимы тесты, проверяющие работу всего пайплайна. При этом нужно принимать во внимание динамическое поведение сервисов и их сложность взаимодействия.

С повышением интереса к микросервисной архитектуре выросло кол-во научных и практических работ, исследующих различные методы её тестирования. Публикации [1][2][3][4][5] агрегируют существующие подходы, описывают проблемы и преимущества каждого из них.

2.1 Стратегии тестирования

В работе Israr Ghani и других[6] исследуются современные подходы к тестированию микросервисов. К ним относятся юнит тестирование, компонентное тестирование, интеграционное, а также контрактное и end-to-end тестирования.

Идеальной парадигмы тестирования не существует. Для лучших результатов необходимо комбинировать разные стратегии. Каждый подход можно охарактеризовать рядом признаков: долей покрытия, ресурсозатратностью, временем выполнения, масштабируемостью. Выбор способа тестирования необходимо принимать индивидуально, исходя из особенностей каждого из них и тестируемой системы.

- **Юнит тестирование.** Это универсальный инструмент - он используется как в монолитной, так и микросервисной архитектуре для тестирования небольших фрагментов кода. Как правило, юнит тесты пишут для проверки работы экземпляров классов и функций. Преимущество этого способа в том, что тесты являются легковесными, выполняются быстро и ими можно охватить мелкие детали функциональности. Однако написание большого кол-ва тестов для широкого покрытия является времязатратным процессом. Помимо прочего, нельзя обходиться толь-

ко этим методом тестирования, так как он не умеет проверять работу сервиса как единое целое.

- **Компонетное тестирование.** Используется для тестирования одной компоненты как сервиса. В микросервисной архитектуре данный тест является приёмочным.

- **Интеграционное тестирование**. Данный способ проверяет взаимодействие группы микросервисов между собой. При этом одновременно могут валидироваться и приемочные стороны, и каналы общения сервисов(сеть), а также глобальные входы и выходы пайплайна. Преимущество интеграционных тестов заключается в том, что они проверяют близкое к продакшну взаимодействие частей сервиса и являются более гибко конфигурируемыми, чем end-to-end тесты, так как тест может поднимать микросервисы небольшими группами. Как описывается в работе D.I. Savchenko и других [7] интеграционные тесты включают в себя проверку всех видов связи между микросервисами - протоколов и форматов данных, наличие взаимоблокировок, разделение общих ресурсов а также последовательность доставки сообщений.

- **End-to-end тестирование.** С помощью данного теста проверятся функциональность всей системы. Этот способ является самым дорогим по затрачиваемым ресурсам и самым долгим по времени выполнения, так как во время его прогона поднимаются все микросервисы пайплайна. Помимо прочего, написание end-to-end тестов - сложный процесс, так как разработчику необходимо разобраться во всей архитектуре сервиса.

3 Архитектура Яндекс.Поиска

3.1 Обзор

Яндекс.Поиск — крупнейшая поисковая система в России и одна из крупнейших в мире. Она обрабатывает десятки миллионов запросов ежедневно и может отвечать на них за доли секунды.

Первая версия Яндекс Поиска появилась в 1997 году и с тех пор претерпела множество изменений. Над этой сложной системой одновременно работают сотни комманд из около 3000 разработчиков. Безусловно, в таком большом проекте, микросервисная архитектура - лучший вариант для безопасной и быстрой разработки.

3.2 Пользовательский интерфейс

Когда пользователь вводит что-либо в строку поиска в поисковом приложении или браузере, запрос отправляется на сервер. После обработки запроса на сервере, результаты отображаются под поисковой строкой. Страница результатов поисковой системы (SERP) обычно состоит из 10 фрагментов, отсортированных по релевантности. После их отображения пользователь может нажать на подходящий по его мнению элемент. Как правило, сеанс пользователя может иметь несколько ветвей. Например, он может содержать просмотр других результатов поиска, новые запросы или другие нетривиальные сценарии.

3.3 Пользовательские логи

С точки зрения разработки клик по результату выдачи - один из типов пользовательских действий. К базовым типам действий также относятся запросы, зумы и драги. Последние живут в рамках подсервиса Яндекс.Карты. Зумы - любые приближения мышью или тачпадом, драги - перетаскивания карты. Все перечисленные действия имеют прямую связь с физическими дей-

ствиями пользователя. Помимо этого, существуют также служебные и технические события. Они могут содержать, например, информацию о факте показа SERP-а или время рендеринга страницы.

Многие сервисы Яндекса, а также других компаний, пишут логи некоторых происходящих событий. Большой интерес для улучшения сервисов представляют логи, сохраняющие действия пользователей. Обработка логов осуществляется в MapReduce кластерах: YT и в Real-Time-Map-Reduce (RTMR). Опишем подробнее необходимые для нашей работы логи:

- `reqans_log` - запросы и ответы поиска.
- `redir_log` - пользовательские действия, включающие служебные/технические события.

Интерес к логам в нашей работе заключается в том, что они являются основным источником информации, по которой можно понять, были ли проблемы во всём пайплайне. Например, при ошибке в какой-либо компоненте, может перестать писаться поле-счётчик с кол-вом кликов, который очень важен для аналитиков и бизнес-части поиска. Соответственно, в нашей работе мы хотим отлавливать выполненные пользователем действия и накапливать их в логах для дальнейшей проверки корректности данных.

3.4 Архитектура Яндекс.Поиска

На рис. 3.1 мы видим общую схему зависимостей микросервисов в Яндекс Поиске. Давайте рассмотрим её детальнее.

Когда пользователь делает запрос, это приводит к вызову функции-обработчика балансировщика L7 `«/search»` с параметром `«text»`, равным поисковому запросу, и необязательным параметром `«lr»`, описывающим регион. Например, запрос https://yandex.ru/search/?text=погода&lr=16 выдаст информацию о погоде в Ярославле. На уровне балансровщика также обрабатываются cookie. Проанализировав их мы можем легко идентифицировать пользователя или

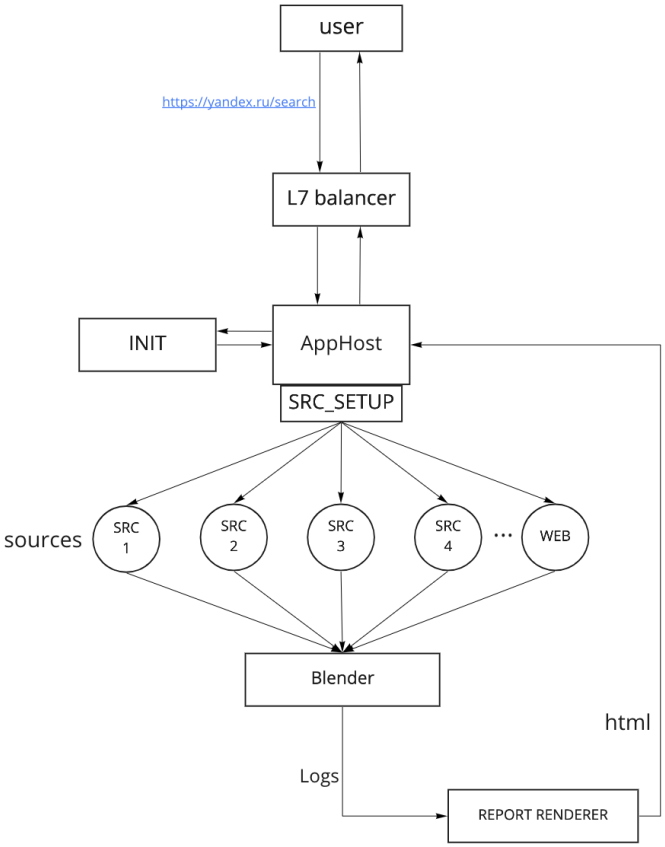

Рис. 3.1: Архитектура Яндекс.Поиска.

понять, что он является новым и мы должны установить временный cookie файл и запомнить его на какое-то время.

Далее обработка запросов управляется графом AppHost. AppHost — это механизм выполнения графа, который на основе запроса знает, какой именно граф следует запускать. Он имеет точку входа с именем `INIT` и несколько `SRC_SETUP`, настраивающих примерно 100 источников, изображенных кружками на рис. 3.1. Источник поиска — это сущность, которая может дополнять страницу результатов URL-адресами, информацией о погоде, календаре и т. д.

Корректные запросы ко всем из них формируются на уровне AppHost-а. В каждом источнике поиска конфигурация однозначно задает точки сбора информации. Каждый источник представляет собой систему микросервисов с собственными бэкендами и сложными зависимостями. Самыми большими и важными являются серверные части отвечающие за веб-поиск. В нашей

терминологии мы будем называть их метапоиском.

После этого ответы от источников направляются на узел Blender. Он объединяет результаты и упорядочивает их в финальные 10, которые будут отображаться на веб-странице. Например, запрос с названием города может возвращать туристическую рекламу, географическое местоположение и некоторую общую информацию о нем. Результаты поиска, в свою очередь, должны быть упорядочены по релевантности с использованием алгоритмов машинного обучения.

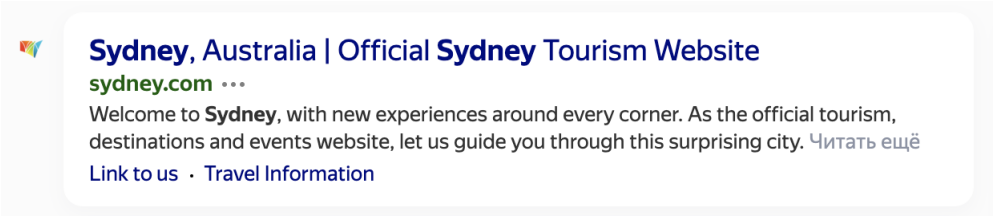

Рис. 3.2: Стандартный сниппет выдачи.

В конце нам нужно нарисовать строку поиска и сформировать блоки с результатами, называемые сниппетами. Стандартный сниппет, как показано на рисунке 3.2, состоит из кликабельного URL-адреса, небольшого логотипа веб-страницы слева и краткой выжимки, содержащей ключевые слова ниже. Таким образом, последняя вершина с именем Report Renderer отвечает за визуализацию. Она отправляет сгенерированный node js-ом HTML-код обратно в AppHost, и затем он отображается пользователю.

4 Детали реализации.

4.1 Общее описание

Наша работа реализована на примере Яндекс.Поиска. Мы опираемся только на то, что система представляет собой микросервисную архитектуру и имеет пользователей, совершающих логируемые действия. Эти абстракции соответствуют большому количеству продуктовых пользовательских сервисов.

У любого пользователя есть идентификатор. Технически наша задача сводится к умению брать действия произвольного юзера и проверять, насколько валидно они были залогированы.

Глобально есть набор некоторых тестовых кейсов, которые мы хотим проверять. Для этого мы прогоняем их на тестируемой компоненте с соответствующими идентификаторами. С другой стороны готовки сессий мы получаем записи по соответствующим идентификаторам и проверяем, что полученные результаты, а именно сессии, являются валидными.

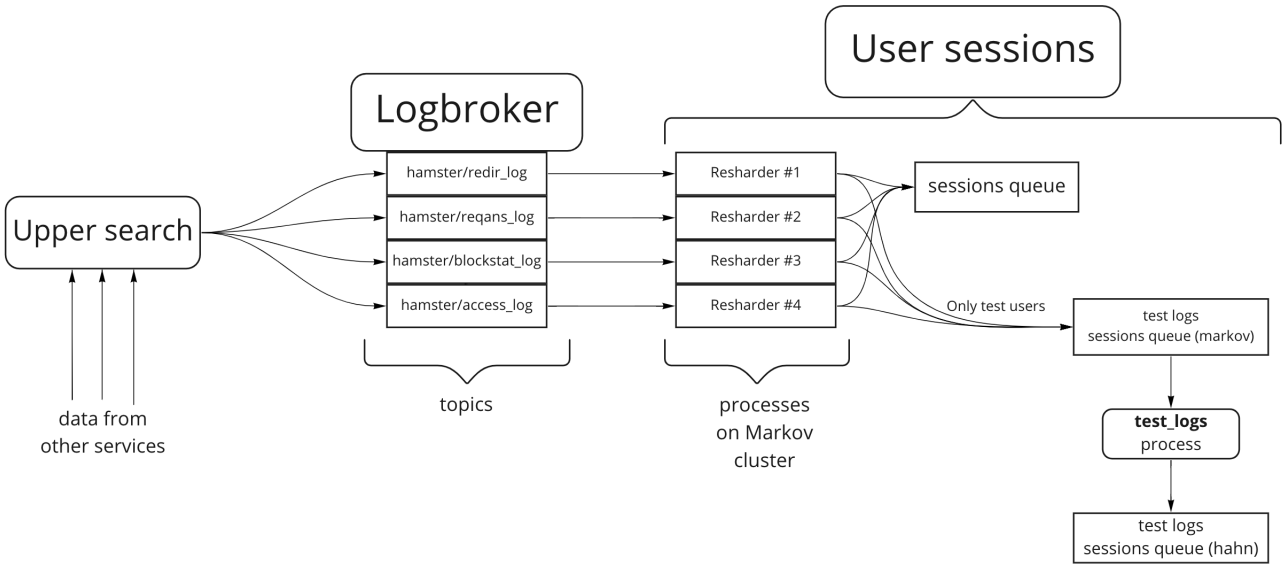

Рис. 4.1: Схема взаимодействия бэкенд сервисов, необходимых для тестирования логов.

На рис. 4.1 изображено взаимодействие бэкенд сервисов, участвующих в нашем тестировании. Как было упомянуто выше, большая часть логов пишется «верхним» поиском. Его выходними источниками являются топики в Logbroker-е [9]. Они представляют собой персистентные очереди с ttl в 36 часов. Как правило эти топики могут потребляться некоторым кол-вом сервисов - для этого достаточно подключить к топику своего читателя. На стороне пользовательских сессий для каждого лога существует resharder процесс - real-time процессинг, поднимающий десятки/сотни инстансов, читающих входные топики, и умеющий автоматически перебалансироваться и перезапускаться при неравномерных нагрузках или проблемах.

Решардер выполняет роль mapper-а. Он группирует записи по пользователям и отправляет их в несколько выходов для сервисов-потребителей. Одним из выходов является очередь `sessions` - она объединяет по пользователям данные разных логов. Основными логами являются `redir_log` и `reqans_log`, так как они содержат достаточную для склейки информацию о запросе, показе и кликах. Соответственно очередь `sessions` содержит обогащённые неким внутренним процессом клики и почищенные от префетчей запросы со всеми полями.

Важно учитывать, что различные логи, относящиеся к одному запросу, пишутся независимо, так как в общем случае пользователь может физически оторваться от сессии, создав тем самым временной разрыв или могут быть лаги в одном из доставляющих микросервисов. Поэтому финальным форматом хранения пользовательских данных является статическая таблица. В ней данные сгруппированны по пользователям (представленных неким идентификатором), а данные по одному пользователю отсортированы в свою очередь по времени (по unix timestamp). Таким образом, сессии готовятся из логов, по которым предварительно произведены обогащения, отфильтрованы роботные запросы и проверено, что данные в них валидны, а именно - ключевые поля не пусты и имеют разумные значения.

4.2 Гермиона тесты и выгрузка reqid-ов

Входной точкой нашего пайплайна является эмуляция пользовательских сценариев или, иначе говоря, тестовых кейсов. Известным инструментом для тестирования web приложений является Selenium. В Яндексе же есть аналог, называемый Гермиона [8], предоставляющий возможность простым кодом описывать пользовательские сценарии. Гермионой пользуется большая часть frontend разработки Яндекса. Этот инструмент очень удобен в использовании из-за простоты установки, возможности параллельного прогона тестов в нескольких браузерах и поддержки пользовательских плагинов. Гермиона

поддерживает интеграционные и end-to-end тесты, все они описываются на javascript и запускаются в реальных браузерах.

Так как цель нашей работы - написание инфраструктуры для тестирования, для проверки работы нам достаточно одного базового теста. Мы описали 3 простых действия - пользователь открывает поисковую страницу, ищет некую строку и затем кликает по второму результату на SERP-е. Однако в тесте необходимы несколько дополнительных инструкций для связки с другими частями пайплайна. Они реализованы с помощью написанных нами плагинов.

Для выделения тестовых пользователей было решено взять всех пользователей с префиксом yandex user id равным `42424242`. Этот идентификатор является внутренним, но из него может быть получена cookie, по которой происходит валидация пользователя в браузере. По этой причине перед открытием страницы в гермиона тесте с помощью плагина `yaGetICookieData` подтягивается cookie, соответствующая случайно сгенерированному id с нужным префиксом. Дальнейшие действия (запросы, клики, скроллы) выполняются с выставленной кукой.

Когда мы запускаем конкретный тест, в сессиях мы хотим получить из всех тестовых записей только те, которые соответствуют Selenium запросам из нашего теста. Для этого нам понадобится фильтр по идентификатору запроса (далее `reqid` или `request id`), который мы будем использовать для сбора необходимых записей в отдельную таблицу. Так как передавать id запроса по какому-то каналу между сервисами трудозатратно, мы можем выгрузить его в облако как временный ресурс и через некоторое время загрузить и использовать на стороне сессий. Для этого в Яндексе существует сервис Sandbox. В свою очередь, в гермионе выгрузка осуществляется плагином `yaUploadReqid`. Для экономии ресурсов и времени по умолчанию этот плагин выключен и включается только для прогона нащих тестов специальной переменной окружения. В дальнейшем в рамках тестирования логов за один прогон будут запускаться десятки тестов, и у каждого из них будет свой

reqid. Поэтому разумной идеей является выгрузка всех идентификаторов одной пачкой в коцне прогона. Средствами hermione это делается описанием события RUNNER_END. Во время выполнения кйса в плагин отправляются идентификатор теста - его уникальное название и request id. Эти пары накапливаются в одном файле и в конце прогона выгружаются в sandbox. Далее в пайплайне это необходимо для сравнения только общих успешно запущенных тестов. То есть если в одной из кодовых версий по какой-то причине тест не был запущен, мы не имеем права сравнивать таблицы, учитывающие соответствующие записи.

Таким образом, алгоритм гермиона теста выглядит следующим образом:

1 Открыть веб-страницу поиска.

2 Заменить cookie на тестовую, полученную с помощью плагина yaGetICookieData

3 Выполнить запрос, задав какое-то словосочетание в поисковой строке.

4 Нажать на один из результатов. (В нашей реализации - на второй).

5 Выгрузить идентификатор запроса в sandbox.

4.3 Описание процесса test_logs на стороне пользовательских сессий

На стороне пользовательских сессий есть 2 контура. Один является real-time процессингом, а другой batch процессингом, т.е. отвечает за ежедневную выгрузку сессий по 30-минуткам и суточным данным. В нашей работе нас интересует только ряд процессов из RT части, а в частности решардеры и новый процесс под названием test_logs. Решардеры читают логи из топиков Logbroker-а, которые пишутся туда разными сервисами. Например blockstat-log пишет report-renderer, access-log пишется Apphost-ом.

Данные пользователей хранятся в динамических таблицах - стейтах, представляющих собой базы данных с ключевыми колонками `user_id` и `timestamp` и колонкой `val` с соответстующими действиями(клики, показы), и очередях - базах данных, состоящих из упорядоченных последовательностей строк. Данные между процессами в основном передаются вторым способом.

Для нашей работы было принято решение выделить некий набор идентификаторов пользователей, который мы будем считать тестовым. А именно, к тестовым относятся все `yuid`, начинающиеся на `42424242`. Как показано на рис. 4.1, для отделения всех тестовых записей, в решардеры была добавлена логика, проверяющая пользователей на соответствие этому префиксу и пишет их данные не в общую продакшн очередь `sessions`, а в новую с названием `test_logs`.

Важно, что все процессы в real-time контуре, взаимодействующие с очередями работают по схеме с 3 датацентрами. Реплицированные очереди находятся на кросс-дц кластере Markov, 2 реплики - синхронная и асинхронная живут в кластерах Hahn и Arnold. После успешной записи на реплики данные с Markov удаляются. Такая схема выбрана неспроста - при падении одного из кластеров-реплик запись будет успешно осуществляться не накапливая лагов обработки, но с ростом потребления памяти. Очередь `test_logs` достаточно маленькая - её размер не превышает сотен Мбайт, поэтому эту динамическую таблицу мы храним как обычную очередь и дублируем все записи на одну реплику на Hahn.

4.4 Описание Sandbox

Sandbox - среда сборки, выполнения задач и обработки данных. Может также использоваться для хранения пакетов, бинарных файлов. Задачи в Sandbox описываются в виде программ на языке Python и исполняются в изолированной среде аналогично docker контентейнерам. В задачах можно настраивать окружение, задав список библиотек и зависимостей. Все фай-

лы, директории или бинарные файлы, выгруженные в облако называются ресурсами.

По умолчанию из sandbox задач нет доступа до всей кодовой базы Яндекса и внешних библиотек. В простых случаях это решается установкой библиотек через `pip`, однако на практике в задачах с большим количеством зависимостей возникают конфликты и трудности. Для решения этой проблемы можно использовать бинарные задачи. Технически они представляют собой бинарные исполняемые файлы, собранные в конкретном репозитории локально. В файле сборки это описывается с помощью макроса `SANDBOX_TASK`, в коде же нужно отнаследовать класс задачи от `binary_task.LastBinaryTaskRelease`. После сборки бинарник выгружается в sandbox передав аргументом описанный тип задачи.

4.5 Основная sandbox задача test_logs. A/B тестирование.

Наши тесты будут запускаться на предрелизном этапе микросервисов. Поэтому, как правило, перед нами есть 2 версии компоненты - продакшн и новая ветка, ожидающая релиз.

Главной задачей, последовательно запускающей процессы всего пайплайна является Sandbox задача `TEST_LOGS`. Глобально она состоит из следующих шагов:

- запустить генерацию `cookie` задачей `GENERATE_ICOOKIE`.
- запустить hermione-e2e тесты для каждой из версий, подложив входную cookie из пункта выше.
- Дождаться доезда логов до очереди `test_logs` на Hahn-е.
- Собрать все выгруженные из hermione задачи ресурсы с `reqid`-ами.

- Заморозить очередь, скопировать из неё данные в статическую табличку, разморозить очередь. Отсортировать получившуюся очередь по ключевым колонкам.

- запустить A/B тесты по статической таблице с сессиями для двух версий - продовой и релизной.

- Сравнить базовые метрики (общее кол-во кликов, кол-во кликов по элементам выдачи, время провождения на странице) на расхождение.

- Выгрузить результаты прогонов и выставить вердикт нашего теста.

Рассмотрим все этапы подробнее.

Для поисковой системы запросы, выполненные из hermione тестов, рассматриваются как запросы каких-то пользователей. Поэтому изначально мы генерируем `yuid` (yandex user id) как `42424242 + random int`. Далее этот id передается в sandbox задачу `GENERATE_ICOOKIE`, которая выгружает сгенерированную cookie в ресурс `ICOOKIE_ARTIFACT`.

Большая часть frontend тестов web поиска прогоняется запуском крупной задачи `SANDBOX_CI_WEB4_HERMIONE_E2E`. Она содержит в себе и и интеграционные и end-to-end тесты, однако для тестирования логов мы планируем проверять только крайние случаи, и для этого в задачу можно передать фильтр, перечислив уникальные названия нужных тестов. За счёт этого время выполения сокращается примерно на 15 минут.

После генерации кук мы мы должны запустить Hermione с 2 версиями компоненты. Для этого существует параметр `beta_domain`, в который передается url обстреливаемой беты поиска. В Яндексе существует инструмент для запуска и верификации бета-версий под названием `yappy`. Он умеет поднимать все микросервисы поиска, заменяя некоторые компоненты на другие версии. Для запуска гермиона тестов мы сначала находим и клонируем (копируем вместе с параметрами) продакшн версию задачи
`SANDBOX_CI_WEB4_HERMIONE_E2E` и меняем в ней часть полей, а именно уби-

раем релизные тэги, в фильтре `custom_opts` перечисляем названия тестов, в `beta_domain` передаём ссылку на бету.

Также мы добавляем переменную окружения `user_sessions_logs_resource_uploader_enabled=true` для включения плагина, выгружающего `reqid`-ы.

```
                'HERMIONE_URL_QUERY_EXP_FLAGS': 'csp_disable',
                'HERMIONE_URL_QUERY_CLCK_HOST': 'hamster.yandex.ru/clck?
l7rwr=clck:hamster-clickdaemon-sas-yp-2.sas.yp-c.yandex.net:80/clck'
```

Рис. 4.2: Переменные окружения, перенаправляющие логи с кликами по произвольной бете в hamster хосты.

В Яндексе существует полная тестовая копия поисковика - hamster, живущая по адресу hamster.yandex.ru. Зачастую тесты прогоняются на ней или на бетах, созданных в pull-request-ах, или в release machine. При этом клики, сделаные по web странице поиска, приводят к вызову ручки «/clck» на основном url-е. Ответственным за это сервисом является Clickdaemon. По умолчанию клики, произведенные по не hamster бетам, не перенаправляются клик-демоном в `redir` топики Logbroker-а. Однако в нашей задаче необходимо, чтобы эти данные доезжали до сессий. Для этого с помощью переменных окружения как показано на рис. 4.2 можно направить логи с кликами по произвольной бете в хосты обработчики hamster-а. А они, в свою очередь, запишут данные в топик `hamster/redir_log`, который подключен к real-time контуру пользовательских сессий.

После прогона всех тестов мы ставим задачу на ожидание в 5 минут, чтобы гарантированно дождаться доезда логов в очередь `test_logs`. Время было выбрано как 99.9 перцентиль времени, проходящего от физического клика до появления данных о нём в сессиях.

Как было описано в главе 4.2 гермиона тесты выгружают свои `request id`. После мы собираем их в 2 файла - по одному для каждой версии.

Следующим шагом является запуск A/B тестов. Kati - инструмент расчета AB-метрик с возможностью их последующего сравнения. На вход он ожидает статические отсортированные таблицы с данными и фильтр со списком

`reqid`-ов. Kati состоит из двух бинарей - `stat_collector`-а, собирающего фичи по нужному срезу, и `stat_fetcher`-а, считающего метрики. Соответственно в задаче `TEST_LOGS` далее осуществляется dump данных из очереди в стат табличку и дважды запускается Kati с разными идентификаторами запросов. На выходе мы получаем 2 json файла с большим кол-вом метрик (порядка 6000). Из них базовыми являются 177, однако в силу простоты наших тестов мы проверяем только 20. К ним, например, относятся общее кол-во кликов, запросов, проведённое на SERP-е время.

Сравнение всех количественных метрик, кроме времени провождения на web странице детерминированно сравнивается на равенство между собой и вшитым в тест значением. Время провождения, в силу потенциальной погрешности сравнивается с точностью до 4 знака после запятой. В случае неравенства кол-ва показов или кликов заранее заданному значению с большой вероятностью проблема является общей и заключается не в компоненте, а в нашем пайплайне или смежном сервисе. Первое время при внедрении мы выставим тест в состояние `muted`, что означает игнорирование его результата и будем наблюдать за стабильностью прокраски теста в зеленый цвет.

На данный момент проведено тестирование написанной инфраструктуры на корректность при работе с 2 стабильно работающими бетами, но не проведено тестирование на выброс ошибки при проверке намеренно сломанной беты.

5 Заключение

В результате работы в поисковый сервис Яндекса была добавлена инфраструктура для тестирования логов, позволяющая значительно уменьшить количество потенциальных поломок. Кроме того, инфраструктура позволяет быстро и удобно дописывать новые тестовые сценарии. В рамках работы была реализована Sandbox задача, запускающая hermione тесты по поднятым на этапе релиза бетам с исходной и новой версиями. Далее она собирает те-

стовые логи в таблицы с пользовательскими сессиями и запускает на них расчёты A/B-метрик.

Данная работа имеет широкую область применения, так как современные сервисы часто пишутся на микросервисной архитектуре. Как пример сервиса с логированием мы рассматривали Яндекс.Поиск, однако нет никаких ограничений для разворачивания аналогичной инфраструктуры в рекламном сервисе, биллинговой системе или любом другом сервисе, собирающем и обрабатывающем пользовательские данные.

В качестве дальнейшей работы можно выделить дописание hermione тестов, наличие которых могло уберечь Яндекс.Поиск от уже случавшихся в прошлом инцидентов, а также исследование других крайних случаев, потенциально приводящих к нарушениям связи между микросервисами. Также направлением дальнейшей работы можно назвать интеграцию Sandbox `TEST_LOGS` задачи в основные компоненты поиска Яндекса.

Список источников

1. Alshuqayran, Nuha & Ali, Nour & Evans, Roger. (2016). A Systematic Mapping Study in Microservice Architecture. 44-51. 10.1109/SOCA.2016.15.

2. Taibi, Davide & Lenarduzzi, Valentina & Pahl, Claus. (2018). Architectural Patterns for Microservices: A Systematic Mapping Study. 10.5220/0006798302210232.

3. Soldani, Jacopo & Tamburri, Damian & Heuvel, Willem-Jan. (2018). The Pains and Gains of Microservices: A Systematic Grey Literature Review. Journal of Systems and Software. 146. 10.1016/j.jss.2018.09.082.

4. Francesco, Paolo & Lago, Patricia & Malavolta, Ivano. (2019). Architecting with Microservices: a Systematic Mapping Study. Journal of Systems and Software. 150. 10.1016/j.jss.2019.01.001.

5. Márquez, Gastón & Osses, Felipe & Astudillo, Hernán. (2018). Review of Architectural Patterns and Tactics for Microservices in Academic and Industrial Literature (Spanish). IEEE Latin America Transactions. 16. 2321-2327. 10.1109/TLA.2018.8789551.

6. Ghani, Israr & Wan Kadir, Wan Mohd Nasir & Mustafa, Ahmad & Babir, Muhammad. (2019). Microservice Testing Approaches: A Systematic Literature Review. 11. 75-80. 10.30880/ijie.2019.11.08.008.

7. Savchenko, D & Radchenko, Gleb & Taipale, Ossi. (2015). Microservices validation: Mjolnirr platform case study. 10.1109/MIPRO.2015.7160271.

8. Hermione. Utility for integration testing of web pages.

9. Logbroker: сбор и поставка больших объемов данных в Яндексе.