

Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

Работа допущена к защите
Директор ВШПИ
П. Д. Дробинцев
« » 2025 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
магистерская диссертация

**РАЗРАБОТКА И АРХИТЕКТУРНЫЕ ОСОБЕННОСТИ POLYMAP: ПЛАТФОРМА ДЛЯ
ПРОЕКТИРОВАНИЯ И РАЗМЕЩЕНИЯ ИНТЕРАКТИВНЫХ КАРТ ПОМЕЩЕНИЙ И
ТЕРРИТОРИЙ. ПРИМЕНЕНИЕ SERVERLESS-ПОДХОДА.**

по направлению подготовки (специальности)

09.04.04 Программная инженерия

Направленность (профиль)

**09.04.04_01 Технология разработки и сопровождения качественного программного про-
дукта**

Выполнил студент гр.
5140904/20102

/ Сопрачев А. К. /

Руководитель доцент,
к.т.н

/ Дробинцев П. Д. /

Консультант по
нормоконтролю

/ Локшина Е . Г. /

Санкт-Петербург
2025

Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и кибербезопасности
Высшая школа программной инженерии

УТВЕРЖДАЮ

Директор ВШПИ

П. Д. Дробинцев

« »

2025 г.

ЗАДАНИЕ

на выполнение выпускной квалификационной работы

студенту Сопрачеву Андрею Константиновичу, группа 5140904/20102

1. **Тема работы:** Разработка и архитектурные особенности PolyMap: платформа для проектирования и размещения интерактивных карт помещений и территорий. Применение Serverless-подхода.
2. **Срок сдачи студентом законченной работы:** 15.05.2025
3. **Исходные данные по работе:** Исходный код iOS приложения PolyMap
4. **Содержание работы (перечень подлежащих разработке вопросов):**
 - Актуальность исследования
 - Анализ существующих решений
 - Требования к разработке
 - Архитектура сервиса
 - Реализация сервиса
 - Анализ результата
5. **Перечень графического материала (с указанием обязательных чертежей):** -
6. **Перечень используемых информационных технологий, в том числе программное обеспечение, облачные сервисы, базы данных и прочие сквозные цифровые технологии (при наличии):** VSCode, Yandex Cloud, Cloudflare
7. **Консультанты по работе:** -
8. **Дата выдачи задания:** 01.04.2025

Руководитель ВКР

/ Дробинцев П. Д. /

Задание принял к исполнению 01.04.2025

Студент

/ Сопрачев А. К. /

Реферат

На 82 с., 0 рис., 0 табл.

КЛЮЧЕВЫЕ СЛОВА:

Abstract

82 pages, 0 figures, 0 tables.

KEYWORDS:

Содержание

Введение	6
Глава 1. Обзор предметной области	6
1.1 Актуальность	6
1.2 Цели и задачи (В введении)	6
1.2.1 Цели	6
1.2.2 Задачи	6
Глава 2. Архитектура. Serverless подход	6
2.1 Анализ требований и проектирование	7
2.1.1 Ролевая модель. Сценарии использования	7
2.2 Клиентская часть	9
2.3 Конструктор карт	10
2.4 Серверная часть	10
2.5 Serverless подход	10
2.5.1 Преимущества и недостатки	11
2.5.2 Cloud Native	12
2.5.3 Выбор облачного провайдера с применением СППР	12
2.5.4 Обзор Serverless компонентов Яндекс Облака	13
2.5.5 Content Delivery Network (CDN)	14
2.5.6 Жизненный цикл запроса	15
2.5.7 Ценообразование	16
2.5.8 Сравнение затрат на Serverless и Kubernetes	16
2.6 Поисковая оптимизация (SEO)	18
2.6.1 Применение SEO в PolyMap	18
2.7 Детальная архитектура	19
2.7.1 Клиенты	21
2.7.2 Микросервисы	21
2.8 Вывод	24
Глава 3. Реализация	25
3.1 Система контроля версий	25
3.1.1 Выбор системы контроля версий	25
3.2 Веб-карта	27
3.2.1 Выбор основного фреймворка	27
3.2.2 Отображение карты	29
3.2.3 Отображение аннотаций	31
3.2.4 Система анимаций	35
3.2.5 Поиск маршрутов	37
3.2.6 Энергоэффективность	39
3.2.7 Хостинг	39
3.3 Серверная часть	41
3.3.1 Стек технологий	41
3.3.2 Шаблон микросервиса	43
3.3.3 Раздача карта и CDN	43
3.4 Мониторинг	44
3.4.1 Логирование	44
3.4.2 Мониторинг	45
3.4.3 Алертинг	45
3.5 Вывод	46

Глава 4. Тестирование и оценка качества кода	47
4.1 Непрерывная интеграция и развёртывание (CI/CD)	47
4.1.1 GitHub Actions	47
4.1.2 Infrastructure as Code (IAC)	47
4.1.3 Terraform	47
4.1.4 Преимущества Serverless в CI/CD	49
4.1.5 Итоговый CI/CD Pipeline в GitHub Actions	52
4.2 Оценка качества кода	55
4.2.1 Применение инструментов оценки качества кода в PolyMap	56
4.3 Модульное тестирование	57
4.4 UI тестирование	58
4.4.1 Инструменты визуального тестирования	59
4.5 Интеграционное и End to End тестирование	62
4.6 Нагрузочное тестирование	64
4.7 Вывод	64
Заключение	64
Список литературы	65
Приложение А. Отчёт СППР о выборе облачного провайдера	66
Приложение Б. Отчёт СППР о выборе фреймворка для веб-карты	72
Приложение В. Реализация функции пружинной интерполяции	78
Приложение Г. Тестирование времени ответа в Serverless режиме	80

Введение

Глава 1. Обзор предметной области

TODO: Подробнее расписать эту главу с табличкой сравнительной и показателями популярности polymap ios

1.1 Актуальность

Обусловлена популярностью пилотной версии PolyMap. В первую неделю учебного семестра, приложением пользовалось более 6000 студентов, что составляет около 80% от всей возможной аудитории.

1.2 Цели и задачи (В введении)

Сервис PolyMap является продолжением бакалаврской работы, в которой была реализована пилотная версия приложения, она обладала следующими ограничениями:

- Поддерживалась только iOS платформа
- Поддерживалась только одна карта Политеха, которая была жёстко закодирована в приложении
- Распространялось в виде приложения, которое требовалось устанавливать на устройство

1.2.1 Цели

Цели магистерской работы вытекают из ограничений пилотной версии приложения:

1. Разработать кроссплатформенное решение, которое будет доступно прямо в браузере, и будет адаптировано под управление как с помощью мыши на компьютере, так и с помощью сенсорного экрана на мобильных устройствах.
2. Реализовать возможность динамического просмотра разных карт, которые будут загружаться из удалённого сервера по запросу пользователя.
3. Серверная часть приложения должна справляться с вариативными нагрузками с высокими пиками. Должно быть быстрое время ответа в разных регионах мира.

1.2.2 Задачи

Для достижения поставленных целей необходимо решить следующие задачи:

1. Спроектировать и реализовать гибкую клиент-серверную архитектуру приложения
2. Разработать веб-приложение, которое будет отображать интерактивные карты в формате Extended-IMDF (формат карт, используемый в приложении PolyMap)
 - 2.а. Реализовать мобильную и компьютерную версии приложений.
 - 2.б. Интерфейс должен быть адаптирован под разные устройства.
 - 2.в. Управление картой на мобильных устройствах должно поддерживать жесты несколькими пальцами (для вращения и приближения карты).
3. Реализовать серверную часть, которая будет хранить карты и предоставлять их пользователю по запросу, а так же обрабатывать сопутствующие запросы веб-приложения (функция поделиться, генерация QR кодов, сокращение ссылок).
4. Добавить в конструктор карт возможность загрузки карты на сервер.

Глава 2. Архитектура. Serverless подход

В этой главе будет рассмотрена верхнеуровневая архитектура сервиса PolyMap, будет подобран оптимальный подход к разработке серверной части, а так же будет рассмотрен выбор оптимального облачного провайдера для реализации Serverless архитектуры.

2.1 Анализ требований и проектирование

Архитектура сервиса в первую очередь определяется требованиями и задачами, которые перед ней ставятся. Одним из подходов к постановке технического задания является использования пользовательских сценариев. Они позволяют описать требования к системе с точки зрения пользователя, что позволяет лучше понять, как система будет использоваться в реальной жизни.

2.1.1 Ролевая модель. Сценарии использования

В сервисе PolyMap выделяются три основные роли:

- **Заказчик** – это человек, который заказывает разработку карты.
- **Пользователь** – это человек, который использует уже созданную карту.
- **Художник** – это человек, который по поручению заказчика оцифровывает карту в конструкторе сервиса.

При этом заказчик и художник могут быть одним и тем же человеком, так как в большинстве случаев, заказчик сам будет оцифровывать карту. Однако, в некоторых случаях, заказчик может поручить эту задачу художнику как своему представителю, так и заказать обрисовку карты у PolyMap.

2.1.1.1 Для заказчика

1. Как **Заказчик**, я хочу смотреть примеры других карт, чтобы определиться с выбором и качеством сервиса.
2. Как **Заказчик**, я хочу иметь возможность заказать размещение, чтобы получить карту.
3. Как **Заказчик**, я хочу делегировать задачу отрисовки карты чтобы самому не тратить на это время.
4. Как **Заказчик**, я хочу смотреть на карту во время отрисовки, чтобы контролировать процесс.
5. Как **Заказчик**, я хочу выбрать кастомный URL на котором будет карта, чтобы его было легко запомнить (например polymap.ru/spbstu).
6. Как **Заказчик**, я хочу иметь возможность создавать приглашения на мероприятия с закодированным маршрутом, чтобы рассыпать их по почте и пользователю не надо было ничего дополнительно делать.
7. Как **Заказчик**, я хочу иметь возможность создавать векторные QR-коды приглашений, чтобы использовать их в печатной продукции.
8. Как **Заказчик**, я хочу чтобы сервера сервисы были в России, чтобы снизить инфраструктурные риски.
9. Как **Заказчик**, я хочу чтобы мои карты индексировались в поисковых движках, чтобы пользователи могли найти карту заведения с использованием обычного поиска Google или Яндекс.

2.1.1.2 Для пользователя

1. Как **Пользователь**, я хочу , чтобы .
2. Как **Пользователь**, я хочу открывать карту прямо в браузере, чтобы не скачивать её как отдельно приложение.
3. Как **Пользователь**, я хочу чтобы карта быстро загружалась, чтобы не приходилось ждать.
4. Как **Пользователь**, я хочу чтобы при повторном открытии карта не скачивалась заново, чтобы не ждать загрузку при повторном открытии.
5. Как **Пользователь**, я хочу мобильный интерфейс, чтобы удобно управлять картой жестами, приближать и отдалять её.

6. Как **Пользователь**, я хочу компьютерный интерфейс, чтобы можно было изучить карту на компьютере.
7. Как **Пользователь**, я хочу автоматическое переключение между ПК и мобильной версией, чтобы не задумываться об этом.
8. Как **Пользователь**, я хочу просматривать не только планировку этажей, а и прилежащую территорию, чтобы изучать взаимное расположение корпусов.
9. Как **Пользователь**, я хочу создавать приглашения и QR коды как и заказчик, чтобы приглашать на студенческие события без излишней бюрократии (например собрание профсоюза).
10. Как **Пользователь**, я хочу строить бесшовные маршруты от любой одной аннотации до любой другой аннотации, даже если это кабинеты в разных корпусах, чтобы лучше планировать маршрут.
11. Как **Пользователь**, я хочу расчёт времени на маршрут с учётом лестниц и входов в здание, чтобы следить за временем и не опаздывать.
12. Как **Пользователь**, я хочу текстовый поиск по аннотациям, чтобы ввести номер кабинета, нажать на него и он отобразился на карте.
13. Как **Пользователь**, я хочу видеть информацию о кабинете, если такая есть, например расписание столовой или любую текстовую заметку, чтобы получать из карты ещё больше полезной информации.

2.1.1.3 Для художника

Большая часть задач художника уже была решена в рамках бакалаврской работы, ниже приведён список недостающих задач.

1. Как **Художник**, я хочу удобную привязку к сетке, чтобы не приходилось вручную выравнивать стены.
2. Как **Художник**, я хочу прямо из конструктора отправлять карту на сервер, чтобы не заниматься бесполезной работой по экспорту в файл.
3. Как **Художник**, я хочу создавать тестовые версии карт, чтобы во время процесса отрисовки смотреть как карта будет выглядеть в реальном приложении.
4. Как **Художник**, я хочу более удобным чем Git механизм совместной работы над картой, чтобы синхронизировать прогресс без навыков использовать Git.

Таким образом, верхнеуровневую архитектуру сервиса можно представить в виде трёх основных компонентов:

1. Клиентская часть – приложение отображающее интерактивную карту.
2. Конструктор карт – доступен для заказчиков карт и художников, позволяет создавать и редактировать карты, а так же загружать их на сервер.
3. Серверная часть – отвечает за хранение карт и сопутствующих данных, а так же за обработку запросов от клиентской части.

Необходимо предусмотреть возможность расширения клиентской части до нативных мобильных приложения на iOS и Android, а так же возможность в будущем, предоставлять доступ к конструктору карт неограниченному числу лиц.

Так же важным требованием является размещение серверной части в России, что позволит снизить инфраструктурные риски, связанных с политической ситуацией в мире.

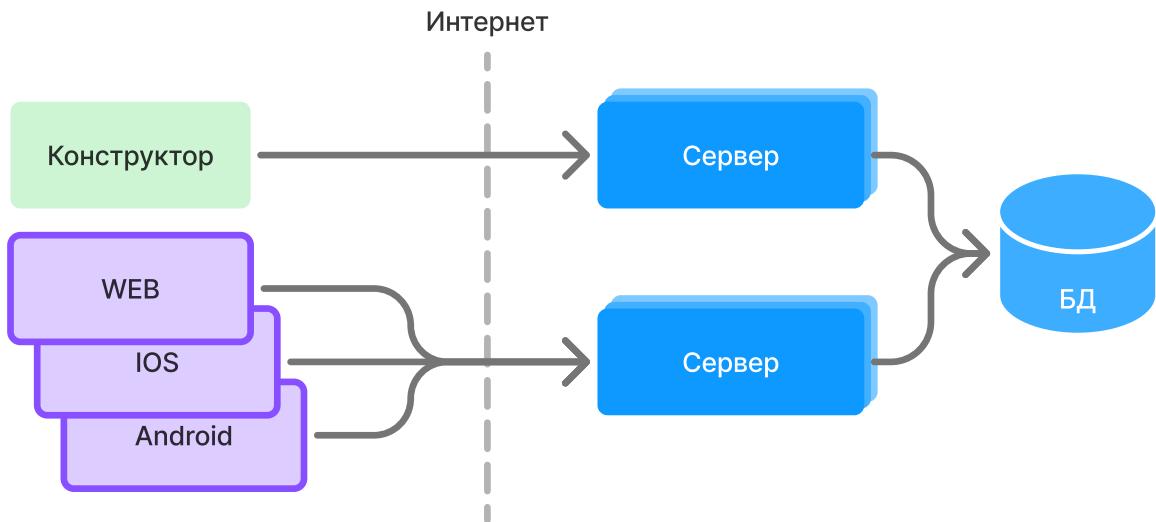


Рис. 1. Общая схема архитектуры приложения

2.2 Клиентская часть

Для соответствия требованиям кроссплатформенности, и скорости загрузки, было решено использовать веб-технологии для разработки клиентской части. Это позволит использовать единое веб-приложение как на мобильных устройствах, так и на компьютерах. Кроме того, это позволит избежать необходимости установки приложения на устройство, что значительно упростит процесс использования сервиса и даст пользователям более комфортный опыт (открыть сайт с картой быстрее и безопаснее для пользователя, чем скачивать и устанавливать приложения).

Клиентское веб-приложение всё ещё требует адаптации под разные устройства и способы пользовательского ввода. Для этого необходимо использовать адаптивный интерфейс, который будет подстраиваться под размер экрана устройства, а так же поддерживать способы управления картой мышкой и сенсорным экраном.

Клиентское приложение должно поддерживать следующие функции:

- Отображение карты в формате Extended-IMDF
 - Просмотр карты
 - Приближение и вращение карты
 - Просмотр планировок этажей
 - Переключения между этажами текущего здания
- Отображение аннотаций на карте и этажах
- Поиск по аннотациями на карте
- Просмотр информации об аннотации
 - Текстовое описание аннотации
 - Расположение аннотации на карте
- Построением маршрута по карте
 - Маршрут между двумя аннотациями
 - Маршрут от стандартной точки начала до аннотации (точка начала своя для каждой карты, их может быть несколько)
 - Маршрут должен бесшовно переходить от прилежащей территории к зданию и между этажами
- Должна быть возможность поделиться маршрутом с другими пользователями
 - С помощью постоянной URL ссылки
 - С помощью QR-кода

Подробная реализация будет рассмотрена в Раздел 3.2.

2.3 Конструктор карт

Конструктор карт является отдельным приложением, которое доступно для художников и заказчиков карт после авторизации. В нём можно создавать и редактировать карты, выгрузка их на сервер осуществляется из специального меню конструктора, которое будет доступно только для авторизованных пользователей. При планировании архитектуры сервиса, необходимо предусмотреть будущие развитие конструктора до публичной веб-версии, к которой будет иметь доступ любой желающий. Общее видение проекта подразумевает бизнес-модель аналогичную конструкторам сайтов, таким как Tilda или Wix. То есть, любой желающий сможет спроектировать карту, загрузить её на сервер и просмотреть, однако, для того, чтобы сделать её общедоступной, необходимо будет оплатить подписку.

Реализация веб-версии конструктора выходит за рамки текущий работы, однако планируется в дальнейших обновлениях сервиса, что накладывает ряд дополнительных требований на архитектуру:

- Весь процесс взаимодействия и обработки новых карт должен быть автоматизирован.
- Необходимо предусмотреть возможность добавления сервера синхронизации для совместной работы над одной картой.
- Необходимо заложить возможность масштабирования серверной части, отвечающей за обработку новых и изменения старых карт.

2.4 Серверная часть

Сервис будет сталкиваться с неравномерной нагрузкой с высокими пиками, которые будут возникать в связи со следующими причинами:

- Повышенный спрос на карту университетов в начале учебного семестра
- Повышенный спрос на карту университетов в начале каждого дня. По утрам, в 10:00 и в 12:00, перед началом пар, студенты будут открывать карту, чтобы найти нужную аудиторию. Что подтверждается статистикой использования пилотной версии.

TODO: Добавить график использования пилотной версии PolyMap

- При использовании карты на временных конференциях и выставках, большинство пользователей будут открывать карту одновременно, что создаст пиковую нагрузку на сервер.

Для решения этих проблем необходимо предусмотреть горизонтальное масштабирование серверной части, для этого **был выбран микросервисных подход** к разработке.

2.5 Serverless подход

Для оркестрации микросервисов наиболее распространено использовать Kubernetes, однако последние несколько лет, растёт популярность альтернативного подхода – Serverless. Он позволяет запускать и масштабировать микросервисы без необходимости взаимодействовать с виртуальными или выделенными серверами. Вместо этого, Serverless предлагает арендовать у облачных провайдеров только вычислительные ресурсы, которые фактически были использованы.

Основными недостатками Kubernetes выделяют сложность настройки качественной инфраструктуры, высокий риск ошибок при ручной настройке, сложность управления версиями, а так же высокую стоимость кластера, особенно на начальных этапах разработки и тестовых средах.

2.5.1 Преимущества и недостатки

Как и любой другой подход, Serverless имеет свои плюсы и минусы. Плюсы Serverless вытекают из минусов Kubernetes:

1. Простота настройки и управления – при использовании Serverless, разработчик не взаимодействует с виртуальными серверами, а только с облачными ресурсами, которые предоставляют достаточно простой и однозначный интерфейс управления.
2. Низкая вероятность ошибок – так как разработчик не взаимодействует с виртуальными серверами, то вероятность ошибок при настройке инфраструктуры значительно снижается. Кроме того, облачные провайдеры предоставляют такие нестабильные, которые не позволяют допустить существенную ошибку.
3. Низкая стоимость – Serverless распространяется по модели pay-as-you-go, то есть необходимо оплачивать только те ресурсы, которые были фактически использованы, что хорошо видно на Рис. 2. Это позволяет значительно снизить затраты на инфраструктуру.

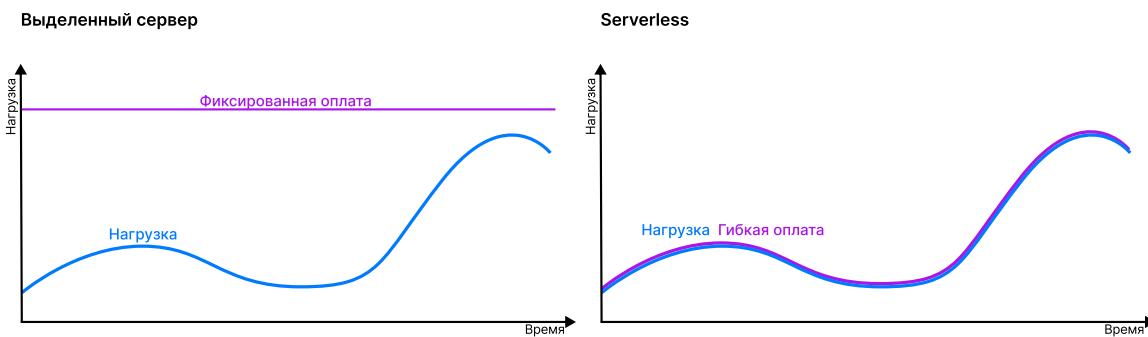


Рис. 2. Сравнение затрат на инфраструктуру

За счёт того, что микросервисы в Serverless подходе запускаются только во время запроса пользователя и отключаются после завершения обработки, они потребляют ресурсы облака только в момент фактической работы, в отличии от Kubernetes, в котором вы арендуете виртуальные сервера вне зависимости от их использования. При переменной нагрузке, Kubernetes будет простаивать значительное время, что приведёт к нерациональным затратам, которых позволяет избежать Serverless.

Однако Serverless имеет и свои недостатки:

1. Требования чистой архитектуры – Serverless подход требует от разработчиков строгого соблюдения принципов чистой архитектуры:
 - микросервисы должны быть независимыми друг от друга
 - связи должны быть сделаны через очереди сообщений
 - микросервисы должны быть State Less – то есть не должно храниться внутреннее состояние, оно будет потеряно при перезапуске микросервиса
2. Привязанность к облачному провайдеру – Serverless – это у большинства крупных облачных провайдеров, однако конечная реализация может отличаться, что сильно усложняет миграцию между облачными провайдерами.
3. Ограниченностей возможностей – не любую микросервисную архитектуру удастся реализовать с помощью Serverless. Например, большинство баз данных не поддерживают Serverless. Однако облака предоставляют доступ к Managed версиям баз данных, к которым можно обращаться из Serverless микросервисов. Кроме того, не все технологии смогут корректно работать в условиях короткого жизненного цикла.
4. Должно быть обеспечено быстрое время холодного старта – так как микросервисы запускаются только во время запроса, то время их запуска должно быть минимальным. Это

накладывает ограничения на используемые технологии, подробнее о выборе подходящего технологического стека рассмотрено в Раздел 3.3.1.

2.5.2 Cloud Native

В современном мире, для разработки и развёртывания приложений, всё чаще используется Cloud Native подход. Он подразумевает разработку приложений, которые будут работать в облаке, используя облачные ресурсы для хранения данных и вычислений. Это позволяет в некоторых аспектах упростить разработку и развёртывание приложений, а так же значительно снизить затраты на поддержку и гарантии доступности.

Serverless является дочерним подходом к Cloud Native разработке. Именно этот подход и был выбран для разработки серверной части приложения PolyMap.

2.5.3 Выбор облачного провайдера с применением СППР

Наиболее крупным игроком в сфере Serverless является Amazon Web Services (AWS), однако официально в России он не доступен, что создаёт дополнительные инфраструктурные риски. По этому, в качестве облачного провайдера необходимо было выбрать отечественную альтернативу. На текущий момент в России существует несколько крупных облачных провайдеров предоставляющих Serverless решения:

- Яндекс Облако – крупнейший в России облачный провайдер, который активно развивает как Serverless, так и Cloud Native решения. Яндекс Облако предоставляет широкий спектр облачных услуг, включая Serverless Container, API Gateway, YDB и другие.
- Selectel – старый игрок на российском рынке, в первую очередь ориентируется на выделенные серверы, однако Serverless решения тоже присутствуют.
- Сбер Облако (переименовали в CloudRu) – относительно новый провайдер, активно развивается, имеет очень выгодные тарифы и бесплатные квоты.
- VK Cloud – является ещё одним новым провайдером, Serverless решения не являются приоритетом.

Стоит отметить, что все эти хостинги соответствуют требованиям российского законодательства о хранение данных (ФЗ-152), что является важным фактором при выборе облачного провайдера.

Для выбора облачного провайдера было принято решение воспользоваться системой поддержки принятия решений (СППР), которая позволяет сравнить облачные провайдеры по заданным критериям. В качестве критерии были выбраны следующие:

- Вычисления – функции (FaaS), наличие serverless контейнеров, технические лимиты, квоты, удобство использования
- Интеграции – удобство вызовов функций/контейнеров (API Gateway), триггеры, очереди, CRON задачи
- БД, Очереди, Уведомления – управления данными – наличие serverless решений, удобство работы, поддерживаемые протоколы
- DevOps & DX – наличие DevOps инструментов, документация, удобство автоматизации процессов. Оценивается CLI, SDK, наличие Terraform провайдера, наличие собственных GitHub/GitLab интеграций, активное сообщество (популярность)
- Мониторинг, логи – наличие инструментов мониторинга и логирования, удобство работы с ними, внутренний функционал (создание своих графиков, алERTов, язык запросов и т.д.), а так же интеграция с внешними системами.

Кроме отечественных облачных провайдеров, в таблице представлены и зарубежные:

- AWS – на текущий момент является лидером в области Serverless, однако официально недоступен в России.
- Google Cloud Platform – второй по популярности облачный провайдер, который активно развивает Serverless решения.
- Azure – облачный провайдер от Microsoft, в нём тоже присутствует Serverless, однако на нём не делают акцент.

Для всех критериев были простираны оценки по шкале от 0 до 10:

0–3 – функционал полностью отсутствует, либо пользоваться крайне неудобно.

4–5 – функционал присутствует, но выполняет базовый минимум задач, серьёзно уступает альтернативам.

6–7 – работает, но есть заметные пробелы, требует ручной настройки.

8–9 – работает хорошо, покрывает все задачи, работать удобно, но есть небольшие недочёты.

10 – самый полный и удобный на рынке функционал в данной категории.

С использованием 5 алгоритмов был рассчитан общий балл для каждого облачного провайдера. В Приложение А. находится подробный отчёт СППР. Использовалась собственная реализация СППР с открытым исходным кодом.

Вариант	Дом	Блок	Тур	Sjp	Sjm	ИТОГО	Место
Yandex	5	6	5	5	6	27	3
Selectel	5	6	2	2	6	21	6
VK	5	6	1	1	6	19	7
Sber	5	6	3	3	6	23	5
AWS	7	7	7	7	7	35	1
Google	6	6	6	6	6	30	2
Azure	5	6	4	4	6	25	4

Таблица 1. Результат работы СППР

Как видно из таблицы 1, наибольшим требуемым функционалом обладает AWS, однако он, как и Google и Azure недоступны в России, наивысший балл среди отечественных сервисов получило Яндекс Облако, поэтому оно было выбрано в качестве облачного провайдера для проекта. Sber Cloud активно развивает свои Serverless решения, однако на момент написания работы, всё ещё сильно отставал от Яндекс Облака. Selectel и VK имеют очень урезанный Serverless функционал, который не позволяет реализовать проект в полном объёме.

2.5.4 Обзор Serverless компонентов Яндекс Облака

Подход к Serverless в Яндекс Облаке реализован с помощью следующих компонентов:

- Cloud Functions – позволяет запускать код в ответ на события, такие как HTTP запросы или CRON задачи.
- Serverless Container – позволяет запускать Docker контейнеры в ответ различные события.
- API Gateway – позволяет описывать REST API для микросервисов в формате OpenAPI. Является публичной точкой входа в микросервисы.
- Message Queue – позволяет организовать асинхронную связь между микросервисами. Совместим с Amazon SQS API.
- Yandex Data Base (YDB) – Serverless база данных, которая может работать и в реляционном и в документо-ориентированном режимах. Совместимо DynamoDB API.

- Object Storage – объектное хранилище, которое позволяет хранить и раздавать бинарные объекты по протоколу S3.
- Cloud Postbox – сервис позволяет организовывать Email рассылки. Совместим с Amazon SES API.

В проекте не используется Cloud Functions, так как все микросервисы реализованы в виде Docker контейнеров, это позволяет использовать более привычный подход к разработке и тестированию, а так уменьшает привязанность к облачному провайдеру. Архитектура состоящая полностью из контейнеров может быть легко перенесена как на локальную машину, так и в другие облака. Кроме того, использование контейнеров позволяет использовать больший набор технологий, в том числе и те, которые не поддерживаются в Cloud Functions.

Кроме непосредственное Serverless компонентов, Яндекс Облако предоставляет и другие сервисы, которые будут полезны в проекте:

- Container Registry – позволяет хранить Docker образы, которые будут использоваться в Serverless Container.
- Monitoring – позволяет хранить и отслеживать состояния работы проекта: число запросов, время ответов и другие метрики. Кроме того, позволяет настраивать Alerting, который будет уведомлять о проблемах в работе проекта при срабатывании настроенных условий.
- Cloud Logging – позволяет собирать, хранить, фильтровать и просматривать логи из всех микросервисов в одном месте.

Yandex Cloud активно развивает Serverless раздел облака, и регулярно добавляет новые компоненты и улучшения. Например, на момент написания магистерской работы, в облако был добавлен Notification Service, который позволяет отправлять персональные уведомления пользователям через SMS, Push и Email. Это позволит в будущем расширить функционал сервиса PolyMap, сильно упростив реализацию личного кабинета заказчика.

2.5.5 Content Delivery Network (CDN)

Для решения задачи географического масштабирования, необходимо использовать Content Delivery Network (CDN) сети. Они пропускают запросы пользователей через географически ближайший к ним узел, и в случае, если запрашиваемый ресурс разрешен для кеширования и уже есть в кеше, отдают его пользователю не перенаправляя запрос на основной сервер. Главным источником трафика в сервисе является раздача карт – это большие и тяжёлые файлы, которые будет скачивать каждый пользователь при каждом открытии карты. Сами карты являются статическими файлами, которые редко меняются, а так же, в большинстве случаев привязаны к конкретной локации. Поэтому их можно эффективно кешировать в CDN сети. Это позволит значительно снизить объём трафика и скорость открытия сайта, что положительно скажется на пользовательском опыте.

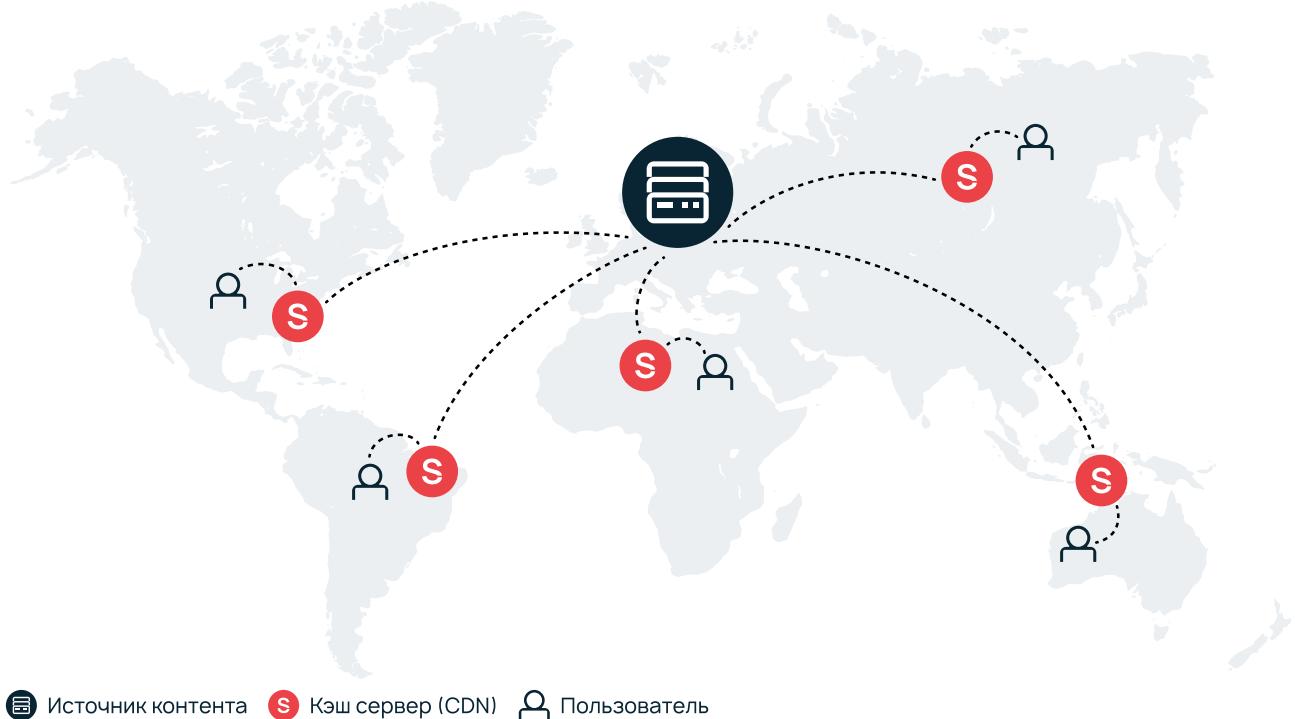


Рис. 3. Схема работы CDN

Общий принцип работы CDN состоит в передаче DNS управления к CDN провайдеру, после чего, провайдер будет разрешать запросы к ближайшему узлу.

2.5.6 Жизненный цикл запроса

При использовании Serverless в Yandex Cloud, жизненный цикл обработки запросы выглядит следующим образом:

1. Пользователь через DNS сеть CDN определяет ближайший к нему узел, и отправляет запрос на него.
2. Узел CDN проверяет, есть ли запрашиваемый ресурс в кеше. Если он есть, то отдает его пользователю, если нет, то проксирует запрос в Yandex Cloud.
3. Yandex Cloud получает запрос и по домену находит соответствующий API Gateway, который будет обрабатывать запрос.
4. API Gateway проводит часть проверок (авторизацию, валидацию параметров) и перенаправляет запрос на соответствующий Serverless Container, который будет обрабатывать запрос.
5. Если нет запущенного экземпляра Serverless Container, то Yandex Cloud создает новый экземпляр контейнера из локального Docker Registry, и перенаправляет запрос на него.
6. Serverless Container обрабатывает запрос, при этом имеет возможность обращаться к прочим ресурсам Облака, таким как базы данных, очереди, и т.д.
7. После обработки запроса, Serverless Container возвращает ответ обратно в API Gateway, который в свою очередь возвращает его пользователю через CDN Proxy.
8. Если указаны заголовки кеширования, то CDN Proxy кеширует ответ на региональном узле, и в дальнейшем будет отдавать его пользователям, не перенаправляя запрос на основной сервер.

В случае обновления статического контента, есть возможность вызвать инвалидацию CDN кеша с помощью специального API. Это позволит удалить из кеша старую карту, чтобы новая версия была доступна пользователям сразу после обновления.

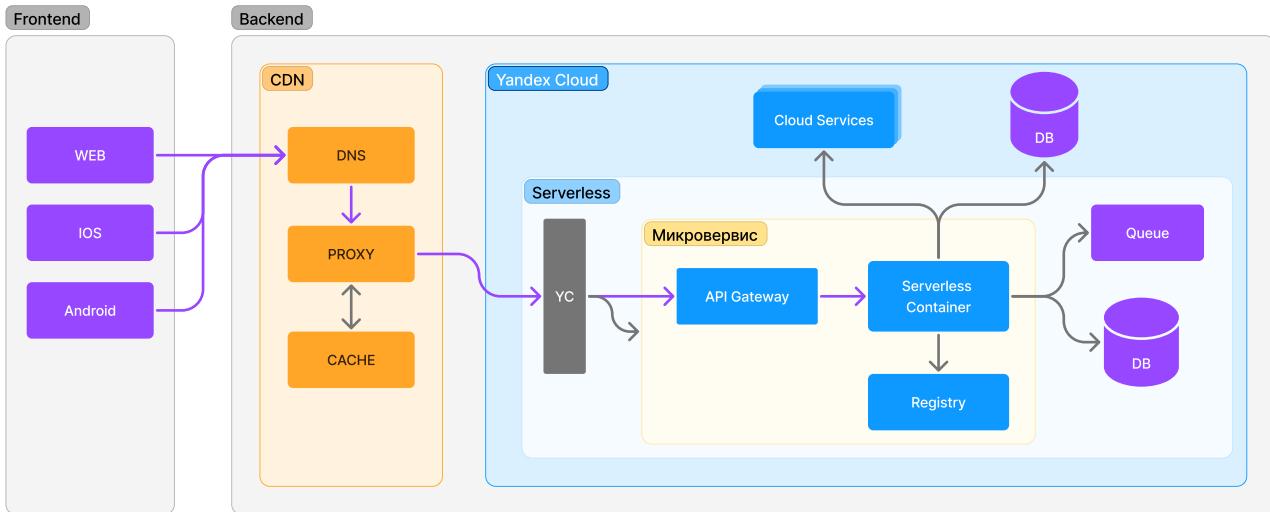


Рис. 4. Инфраструктурный путь запроса

2.5.7 Ценообразование

Основным преимуществом Serverless подхода является снижение инфраструктурных затрат относительно классических походов. Рассмотрим ценообразование в Yandex Cloud.

Важным фактором затрат на Serverless в Yandex Cloud является квоты на число бесплатно предоставляемых ресурсов. Эти квоты сбрасываются в начале каждого месяца, таким образом, низкая нагрузка в течение месяца, позволяет пользоваться инфраструктурой полностью бесплатно. После исчерпания квот, тарификация происходит по модели pay-as-you-go, то есть необходимо оплачивать только те ресурсы, которые были фактически использованы.

Ресурс	Стоимость	Бесплатно
Serverless Containers	120₽/1М запросов	Первые 100к в месяц
	16₽/1М вызовов	Первые 1М вызовов в месяц
	3.2₽/ГБ×час ОЗУ	Первые 1ГБ×час в месяц
	4.8₽/vCPU×час	Первые 1vCPU×час в месяц
Message Queue	48.76₽/1М запросов	Первые 100k в месяц
Yandex Data Base	21.38₽/1M Request Units	Первый 1M в месяц
	21.38₽/1ГБ×месяц	Первый 1ГБ в месяц
Object Storage (Standard)	GET	0.39₽/10000
	POST	0.48₽/1000
	DELETE	Бесплатно
Исходящий трафик	< 1ТБ	1.53₽/1ГБ
	< 50ТБ	1.28₽/1ГБ
	< 100ТБ	1.20₽/1ГБ
	> 100ТБ	1.15₽/1ГБ

Таблица 2. Стоимость используемых Serverless ресурсов в Yandex Cloud на 01.05.2025

2.5.8 Сравнение затрат на Serverless и Kubernetes

С использованием цен актуальных на момент написания работы (Таблица 2) можно ориентировочно рассчитать общие затраты на инфраструктуру в месяц. Возьмём ориентировочную нагрузку на пилотную версию PolyMap iOS (Таблица 3).

Ресурс	Объём
Открытие карты	600 000
Пользователей	8 000
Просмотров информации об аннотации	70 000
Построено маршрутов	10 000
Открыто приглашений	250

Таблица 3. Потребляемые ресурсы пилотной версии PolyMap iOS за сентябрь 2024

Стоит отметить, что приложение не имеет официальный статус, и его не использовали для приглашений на мероприятия. В связи с этим, нагрузка на открытия приглашений сильно занижена, при реальном использовании на конференциях, число открытых приглашений будет пропорционально пришедшим пользователям.

Перенесём статистику на примерный механизм работы Serverless:

- API Gateway: 600 000 открытых карт + 70 000 просмотров аннотаций + 10 000 построений маршрутов + 250 открытых приглашений = 680 250 запросов
- Serverless Containers: 680 250 вызовов (среднее потребление: 0.5 ГБ памяти, 0.2 с процессора на вызов)
- Message Queue: 680 250 запросов
- Object Storage GET: 680 250 операций
- Исходящий трафик: 600 000 загрузок карт по 2 МБ = 1 200 ГБ

Ресурс	Объём	Бесплатно	Платно	Стоимость, ₽
API Gateway	680k запросов	100k	580k @120 ₽/1 млн	≈ 69.6
Вызовы контейнеров	680k вызовов	1M	—	0
Память (ГБ×ч)	18.9	1	17.9 @3.2 ₽/ГБ×ч	≈ 57.3
vCPU (vCPU×ч)	10.5	1	9.5 @4.8 ₽/vCPU×ч	≈ 45.6
Message Queue	680k запросов	100k	580k @48.76 ₽/1 млн	≈ 28.3
YDB RU	680k	1M RU	—	0
Object Storage GET	680k операций	100k	580k @0.39 ₽/10k	≈ 22.6
Исходящий трафик	1 200 ГБ	100ГБ	1 100 @1.28 ₽/ГБ	≈ 1 408
ИТОГО				≈ 1 640

Таблица 4. Оценка ежемесячных затрат на Serverless ресурсы

Как видно из расчёта (Таблица 4) на самый нагруженный месяц – сентябрь, приблизительные затраты на Serverless инфраструктуру составят **1600 рублей**, и большая часть затрат это исходящий трафик, который порождается раздачей статичных файлов карты, однако, большинство этих запросов не будут доходить для серверов, а будут обрабатываться на узлах кеша CDN, по этому реальные затраты будут значительно ниже.

Сравним это с затратами на Kubernetes кластер. Так как нагрузка не очень высокая, возьмём минимальную production конфигурацию кластера, которая будет состоять из 3 нод с 4 vCPU и 4 ГБ RAM. Три ноды позволят обеспечить доступность и отказоустойчивость сервера, а 4 vCPU и 4 ГБ RAM будет достаточно для обработки аналогичной нагрузки. Такой кластер в Yandex Cloud будет стоить **33000 рублей** в месяц, что значительно выше, чем затраты на Serverless. Для сравнения, на Рис. 5.6 есть стоимость минимально возможного кластера в Selectel, он имеет меньшую отказоустойчивость, так как находится в одной зоне доступности,

однако обладает большими характеристиками: 8 vCPU и 16 ГБ RAM, такой кластер будет стоить **31000 рублей** в месяц.

Managed Service for Kubernetes®	33 201,27 ₽	Для production
Кластер Kubernetes® 1	33 201,27 ₽	3 мастер-ноды 2 ворк-ноды Балансировщик нагрузки
Managed Kubernetes. Regional Master - small	17 293,82 ₽	Характеристики ворк-ноды
Intel Broadwell. 100% vCPU	9 967,10 ₽	vCPU
Intel Broadwell. RAM	3 470,69 ₽	RAM
Быстрое сетевое хранилище (SSD)	2 469,66 ₽	SSD
Итого	33 201,27 ₽	31 988,77 ₽/мес.

(a) Yandex Cloud

(б) Selectel

Рис. 5. Стоимость Kubernetes кластера в Yandex Cloud и Selectel

Так же, не стоит забывать, что кроме production кластера, необходимо будет поддерживать тестовый кластер, для разработки и тестирования новых обновлений, что значительно увеличит затраты на инфраструктуру, в то время, как в Serverless подходе, тестовые среды даже не будут выходить из бесплатных квот. За три года активной разработки, я ни разу не использовал больше 10% от выделенных квот, что позволяет полностью бесплатно разрабатывать и тестировать новые обновления.

2.6 Поисковая оптимизация (SEO)

Важной частью любого веб-сервиса является поисковая оптимизация (SEO – Search Engine Optimization). Этим термином называют набор методов, которые позволяют улучшить видимость сайта в поисковых системах, таких как Google, Yandex и других. Это позволяет увеличить количество пользователей, которые будут органически находить ресурс через поиск. В качестве методов SEO можно выделить:

- Оптимизация мета-тегов – это теги в заголовке HTML страницы, которые содержат информацию о сайте: Название, описания, ключевые слова, язык
- Создание OpenGraph мета-тегов – это специальный подвид мета-тегов, в котором описывается как будет выглядеть предпросмотр ссылки на сайт в социальных сетях.

TODO: В идеале сюда добавить сравнительную картинку с OpenGraph тегами и без них

- Персонализация страниц – для SPA (Single Page Application) приложений, необходимо генерировать уникальные index.html страницы в зависимости от контента, это можно сделать с помощью серверного рендеринга (SSR – Server Side Rendering) или статической генерации страниц (SSG).
- robots.txt – специальный файл, который позволяет указать поисковым системам, какие страницы сайта не нужно индексировать.
- Sitemap.xml – специальный файл, в котором описывается структура сайта, по которой поисковые системы могут понять, какие страницы сайта нужно индексировать, а какие нет.

2.6.1 Применение SEO в PolyMap

В сервисе PolyMap есть два основных направления SEO:

1. Лендинг страница с описанием возможностей сервиса, его преимуществами и примерами использования.
2. Веб-версия карт, которая отображает карты заказчиков.

В случае с лендинг страницей, особых сложностей в SEO нет, так как это статичный веб-сайт, который будет использовать SSG подход, что позволит поисковым движкам с лёгкостью его анализировать.

Веб-версия карт куда сложнее, это SPA приложение, которое в зависимости от URL будет подгружать карту уже после открытия сайта. При этом сами карты являются динамичным контентом, они могут добавляться новые, а старые исчезать. SPA приложения работают следующим образом: с сервера загружается пустой index.html файл, в котором объявлен основной JavaScript файл, после скачивания которого, приложение запустит фреймворк и динамически нарисует страницу.

Для поисковой оптимизации SPA с динамическим контентом, принято используют SSR подход, при нём, на каждый запрос пользователя, сервер запускает JavaScript код приложения, и создаёт полную HTML страницу, после чего, прикрепляет состояния фреймворка в виде JSON объекта и отвечает пользователю. На стороне пользователя запускается процесс гидратации (hydration) – это процесс, при котором фреймворк берет HTML страницу и JSON объект, и запускает приложение с уже готовым состоянием. Однако этот подход значительно увеличивает нагрузку на сервер, скорость ответа, а так же время до первого взаимодействия (Time to First Interaction – TTFI), это происходит из-за медленного процесса гидрации.

Для PolyMap был выбран другой подход. Благодаря тому, что карты не являются текстовым контентом, можно использовать SSG только для мета-тегов, а всё остальное генерировать на стороне клиента классическим SPA подходом. Для генерации мета-тегов достаточно подменить их в index.html файле, получаемом от фреймворка, никакие другие скрипты в процессе участвовать не будут. В проекте для этого используется отдельный микросервис web-map-back, который в зависимости от URL будет генерировать нужную index.html страницу, а так же, по запросу поисковых движков, будет генерировать актуальные sitemap.xml и robots.txt файлы. Подробнее о его работе будет рассказано в разделе 2.7.2.9.

2.7 Детальная архитектура

Детальная архитектурная схема сервиса PolyMap представлена на рисунке 6. Её можно разделить на несколько основных типов компонентов:

- Клиенты – это приложения с которыми взаимодействует пользователь, заказчик и художник. Они могут быть как веб-приложениями, так и нативными программами. На схеме они представлены блоками фиолетового цвета.
- Внешние сервисы – не являются частью PolyMap, однако взаимодействуют с сервисом. Например внешние сервисы аналитики или запросы OpenGraph Meta. На схеме они представлены блоками тёмно серого цвета.
- Микросервисы – это основные вычислительные компоненты. На схеме они представлены блоками белого цвета.
- CDN – это прослойка между клиентами и микросервисами, которая позволяет кэшировать статические файлы. На схеме она представлена блоками оранжевого цвета.
- Системы хранения – это места хранения данных, на схеме представлены:
 - Синим цветом, префикс **SQL** – SQL базы данных, например PostgreSQL или YDB в SQL режиме.

- Зелёным цветом, префикс **DOC** – документо-ориентированные базы данных, например YDB в NoSQL режиме или MongoDB.
- Оранжевым цветом, префикс **S3** – объектные хранилища S3.
- Синим цветом, префикс **OLAP** – Аналитические базы данных, например ClickHouse.

TODO: исправленная схема в фигме

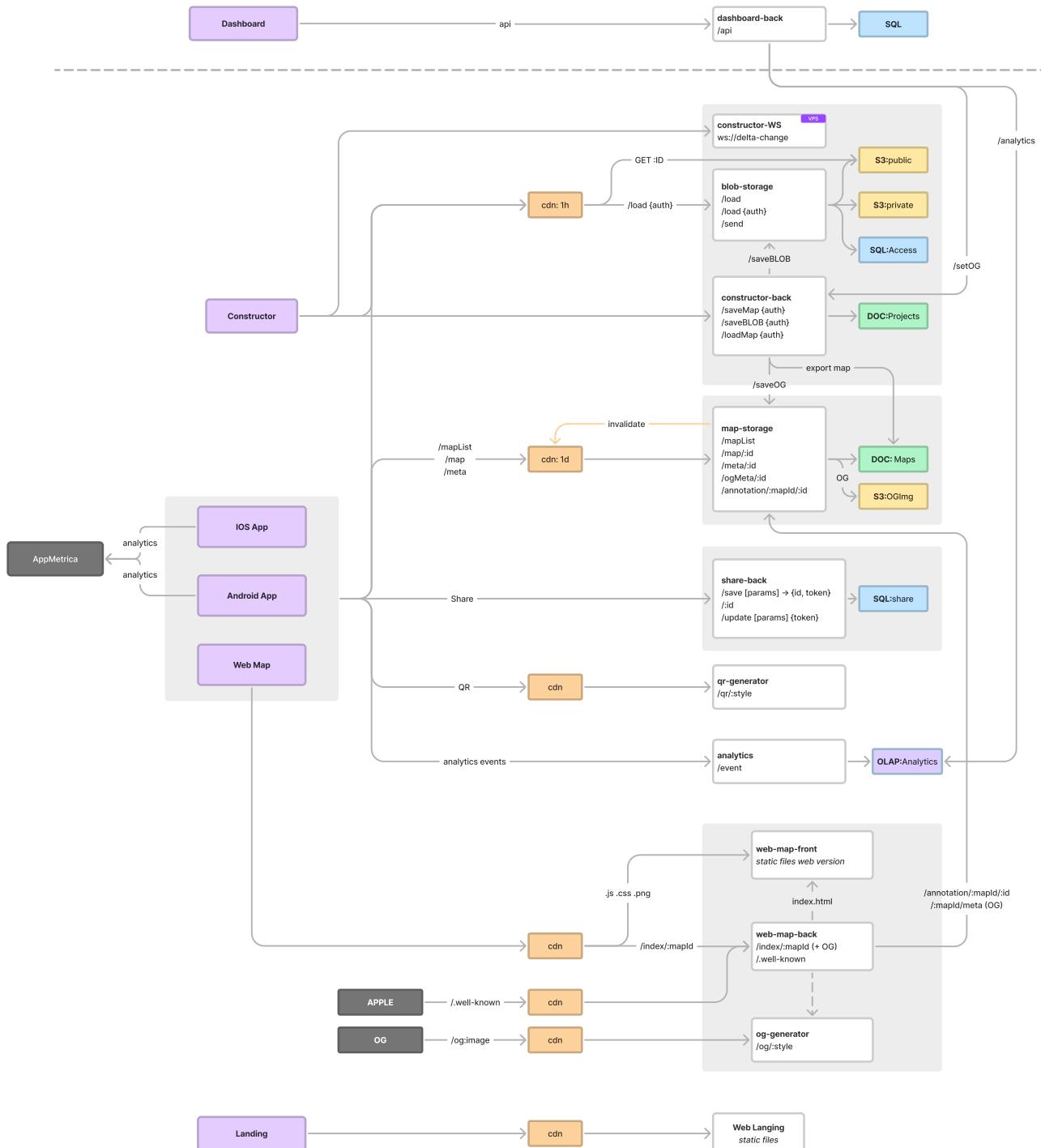


Рис. 6. Детальная схема архитектуры PolyMap

Далее будут компоненты архитектуры будут рассмотрены подробнее, будет определено их назначение и обоснована значимость для сервиса.

2.7.1 Клиенты

Клиентами являются приложения, с которыми взаимодействует человек, такие приложения подразумевают графический пользовательский интерфейс. При проектировании архитектуры сервиса PolyMap, необходимо учитывать следующие клиенты:

- Dashboard – панель управления для заказчиков.
- Constructor – приложение для художников, которое позволяет создавать карты.
- iOS App, Android App, Web Map – нативное приложения, которое позволяет просматривать карты.
- Landing – сайт с описанием сервиса, его возможностей и преимуществ.

2.7.1.1 Dashboard

Архитектурный компонент запланированный на будущее. Это веб-приложение с панелью управления, доступ к которому будет выдаваться каждому заказчику. В нём будет доступен список карт, которые были созданы этим заказчиком. Будет возможность оплаты и подробная информация о каждой карте.

Для каждой карты можно будет просматривать аналитику (подробнее Раздел 2.7.2.8), настраивать доступ к карте, управлять художниками, переходить к конструктору карты. Настраивать внешние интеграции (такие как расписание), просматривать список приглашений и их статистику.

Данный компонент не является критическим для работы сервиса, и его реализация не является приоритетной задачей. Он будет реализован в будущем, когда сервис будет готов к коммерческому использованию.

2.7.1.2 Constructor

Это компьютерное приложение, которое позволяет создавать карты. В будущем оно будет перенесено в веб-приложение с системой разграничения доступа. В текущий момент используется старая версия конструктора, разработанная в рамках бакалаврской ВКР. Оно написано на базе движка Unity3D с добавлением кастомных окон.

2.7.1.3 iOS App, Android App, Web Map

Клиентские приложения непосредственно отображающие карты. В рамках текущей работы, реализуется только веб-версия приложения, однако планирование архитектуры требует учёта всех версий, которые будут реализованы в будущем.

2.7.1.4 Landing

Это веб-сайт на котором будет размещена общая информация о сервисе, его преимуществах, возможностях и ценах. На сайте можно будет оставить заявку на создание карты. Сайт реализован в виде статического сайта, который будет раздаваться через CDN. Серверной части не требует, с другими микросервисами не взаимодействует.

2.7.2 Микросервисы

Микросервисами являются приложения, с которыми взаимодействуют клиенты (приложения) или другие микросервисы, такие приложения не подразумевают графический интерфейс, взаимодействие с ними происходит через API.

2.7.2.1 Dashboard-back

Этот микросервис будет реализован в будущем вместе с панелью управления. Он будет отвечать за взаимодействие с панелью управления, регистраций заказчиков, хранением сервисных настроек карт, управлением картами, расчётом аналитических графиков.

Так же, через него можно будет задать OpenGraph метаданные для карт. Он будет взаимодействовать с другими микросервисами: `map-storage`, `analytics`.

2.7.2.2 Constructor-WS

Это микросервис, который будет отвечать за синхронизацию совместной работы в конструкторе карты. Он будет использовать pub/sub WebSocket, клиенты конструктора будут отправлять delta-синхронизации. На схеме архитектуры (Рис. 6) этот компонент помечен тегом `VPS`, что означает, что он не является частью Serverless. Это связано с тем, что на момент написания работы, в Yandex Cloud тарификация WebSocket соединений оплачивается за каждое сообщение, по цене эквивалентной обычным запросам, что делает является невыгодным. При этом, данный сервер синхронизации не требует большой вычислительной мощности и гибкого масштабирования, так как будет доступен только художникам карты только в момент редактирования карты. Поломка механизма синхронизации не приведёт к критичным последствиям. По этому он будет реализован в виде обычного виртуального сервера, на котором будет Nginx и Docker контейнер с сервисом синхронизации.

2.7.2.3 Blob-storage

Микросервис хранения бинарных объектов. Он может использоваться разными сервисами, но в первую очередь предназначен для конструктора карт, чтобы художники могли загружать свои собственные изображения аннотаций, планы этажей и т.д. Непосредственно хранение будет осуществляться в объектном хранилище S3, сервис будет отвечать за управление доступом. Некоторые ассеты конструктора должны быть приватными и доступны только внутри конструктора и только для определённой карты (например инженерные планировки), а некоторые должны быть публичными, такие как изображения аннотаций.

Загружать новые ассеты в хранилище смогут только художники, уровень доступа ассе-тов будет определяться картой из которой они были загружены.

2.7.2.4 Constructor-back

Микросервис обслуживающий конструктор карт. Он будет отвечать за сохранение карт из конструктора, а так же за дополнительную обработку карты и её экспорт в `map-storage`. Будет предоставлять все ассеты карты в редактируемом виде (для конструктора) с учётом авторизации художника.

2.7.2.5 Map-storage

Самый важный микросервис для проекта, отвечает за хранение и раздачу карт. Кроме самих файлов карт для клиентов, он будет позволять получить информацию об аннотации не скачивая всю карту целиком. Будет хранить метаданные карты, и OpenGraph метаданные в том числе картинки. Этот же микросервис будет отвечать за инвалидацию кеша CDN при обновлениях карт.

Получать информацию об аннотациях нужно будет из микросервиса `web-map-back` для генерации OpenGraph метаданных для SEO оптимизации страниц аннотаций. Например при открытие страницы `/spbstu/annotation/uuid` будет получать информацию об аннотации с `uuid` для карты `spbstu`, и генерировать OpenGraph метаданные для страницы, для конкретной

аннотации. Это позволит поисковым движкам лучше индексировать страницы популярных аннотаций, и выдавать их в результатах поиска.

2.7.2.6 Share-back

Вспомогательный микросервис для клиентов, который отвечает за создание приглашений на карту. Приглашения состоят из аннотации на карте или маршрута и названия с описанием. Необходимо сохранить эту информацию в базе данных и вернуть пользователю уникальный короткий URL. По реализации этот сервис очень схож с классической задачей сокращение ссылок. Ничего специфичного в нём нет.

2.7.2.7 Qr-generator

Один из самых дискуссионных микросервисов. Он отвечает за генерацию QR-кодов, которые нужны клиентам для создания приглашений. Эти QR-коды должны быть стилизованными, должны поддаваться настройке по цветам и виду, а так же предоставляться в виде растровой png картинки или векторного svg изображения.

Может показаться, что его реализация в виде микросервиса нецелесообразна, и куда эффективнее было бы использовать клиентскую библиотеку, которая бы генерировала QR-коды прямо на клиенте в принципе без необходимости использования сервера. Однако, в дальнейшем, сервис планируется развивать до нативных мобильных приложений на других языках программирования, а стилизация QR-кодов везде должна быть одинаковой. Кроссплатформенных библиотек для стилизации QR-кодов соответствующей требованиям не существует, и пришлось бы разрабатывать её самому. Кроме того, в случае с мобильными приложениями, обновление этой библиотеки потребовало бы обновления приложения, а далеко не все пользователи обновляют их своевременно. Ещё одним аргументом в пользу микросервиса является то, что генерация QR кода может быть востребована другим сервером, например для Email рассылок, в случае с которыми, пришлось бы не просто использовать библиотеку на сервере, а ещё и сохранять изображения QR-кодов в S3.

В случае с использованием микросервиса, все сложности по стилизации QR-кодов нивелируются. Сервис предоставляет REST API, в котором все параметры QR-кода определяются в виде query параметров, а в результате возвращается изображение. Этот подход позволяет:

- Использовать один и тот же сервис для всех платформ
- Для вложений использовать просто URL ссылку на QR-код, который будет генерироваться в момент запроса (Что полезно для Email рассылок)
- Не зависеть от обновлений библиотек, так как сервис будет поддерживаться отдельно от клиентских приложений
- Не сохранять QR-коды в S3, а генерировать их в момент запроса, что позволяет избежать лишних затрат на хранение.
- Кэшировать QR-коды на CDN, что позволит избежать лишних запросов (например для рассылок)

TODO: В идеале сюда добавить пример URL для генерации QR-кода и сам QR-код

2.7.2.8 Analytics

Сервис для сбора аналитики от клиентов. Будет собирать информацию о событиях на картах и анонимизированную информацию о пользователях. Это нужно для того, что бы заказчики карт могли видеть, как пользователи взаимодействуют с картами, какие аннотации открываются чаще всего, какие маршруты строятся, как часто пользователи открывают карту,

сколько их, оценивать удержание. Сервис будет сохранять информацию в ClickHouse – это колоночная СУБД предназначена специально для аналитики. Собственное хранилище аналитики позволяет избежать использования сторонних сервисов, таких как Google Analytics, которые могут собирать информацию о пользователях и передавать её третьим лицам. Это важно для соблюдения GDPR, ФЗ-152 и других законов о защите персональных данных. Кроме того, собственное хранение позволит бесплатно и в любых количествах фильтровать аналитику по карте и выдавать эту информацию заказчикам. Сторонние сервисы ограничивают запросы к API, особенно с дополнительными фильтрами, так как их базы данных не предназначены для этого.

2.7.2.9 Web-map-back

Этот микросервис крайне важен для SEO оптимизации, он занимается генерацией index.html страницы для web-версии карт. В зависимости от URL запроса, он будет добавлять в сгенерированный UI фреймворком index.html файл, мета-теги, которые будут использоваться поисковыми движками для индексации страницы, а так же обслуживать нужны поисковых движков, возвращая корректные

- `sitemap.xml` – структура сайта, в ней будут URL всех карт, которые отмечены как публичные. Эти карты будут индексироваться поисковыми движками. Например и на запрос «Карта Политеха» в поисковой системе, будет отображаться сайт PolyMap с соответствующей картой. В будущем, можно будет попробовать добавить ещё и все аннотации на каждой карте, чтобы прямо в поисковике можно было искать «Политех корпус 1, аудитория 101», и получать ссылку на карту с соответствующей аннотацией.
- `robots.txt` – служебный файл, который позволяет указать поисковым системам, какие страницы сайта не нужно индексировать, будет использоваться для приватных карт, которые не должны индексироваться поисковыми системами.
- `.well-known/apple-app-site-association` – специальный файл, который позволяет iOS устройствам открывать ссылки на карты в нативном приложении, а не в браузере.

2.7.2.10 Og-generator

Микросервис по функционалу аналогичный qr-generator, но генерирует не стилизацию qr кодов, а OpenGraph изображения, которые будут использоваться социальными сетями для предпросмотра ссылок на карты. Микросервис web-map-back будет создавать мета-теги, в том числе по протоколу OpenGraph, помещая в `og:image` специальные URL на микросервис og-generator, который по GET запросу на этот URL будет генерировать изображение. Изображение кешируется в CDN. Все параметры изображения определяются в `query` параметрах ссылки. Будет поддерживаться генерация названия карты, аннотации, возможно, предпросмотра маршрута. Кроме того, заказчики смогут кастомизировать свои OpenGraph изображения, например добавлять логотипы, в этом случае, web-map-back будет добавлять ссылку на эти изображения в S3:OGImg. Это позволит сделать og-generator полностью изолированным от других микросервисов, что считается хорошей архитектурной практикой.

TODO: В идеале сюда добавить картинку OpenGraph

2.8 Вывод

В данной главе были проанализированы требования к сервисы, определен Serverless подход к архитектуре, с помощью системы поддержки принятия решений, выбран оптимальный облачный провайдер Yandex Cloud. Так же, была подробно рассмотрена детальная

архитектура, разбивка на клиенты и микросервисы, обоснована их целесообразность и взаимодействия. Рассмотрена оптимизация поисковых запросов.

В результате получилась хорошая чистая архитектура, в которой все компоненты слабо связаны между собой.

Глава 3. Реализация

В этой главе будут рассмотрены некоторые детали реализации сервиса PolyMap, такие как система контроля версий, непрерывная интеграция и развёртывание (CI/CD), а так же некоторые детали реализации некоторых микросервисов. Будут выбраны языки программирования, фреймворки и основные библиотеки, которые будут составлять технический стек проекта. Так же, будет рассмотрена система мониторинга и логирования, которая будет использоваться в проекте.

3.1 Система контроля версий

При разработке проекта в микросервисном подходе выделяют два способа организации системы контроля версий:

1. Монорепозиторий – все микросервисы хранятся в одном репозитории. Плюсом этого подхода является сквозное версионирование всех микросервисов, каждый коммит порождает новую общую версию приложения, которая гарантирована не нарушить совместимость. Минусом такого подхода является сильная связанность между микросервисами, что усложняет их независимую разработку, тестирование и развёртывание.
2. Полирепозиторий – каждый микросервис хранится в своём репозитории. Плюсом такого подхода является независимость разработки, тестирования и развёртывания каждого микросервиса. Минусом такого подхода является сложность в управлении версиями. Может случиться так, что при обновление одного микросервиса сломается совместимость с другим.

Был выбран второй подход, так как он лучше позволяет разделить кодовые базы и вести независимую разработку. Сложность версионирования решается гарантиями обратной совместимости – ни одна новая версия не должна ломать совместимость с предыдущими версиями. Это требование и так необходимо соблюдать, что бы корректно работало горизонтальное масштабирование, при котором в один момент времени могут работать несколько версий одного микросервиса.

3.1.1 Выбор системы контроля версий

В качестве хранилища системы контроля версий Git можно использовать несколько систем:

- GitHub
- GitLab
- Bitbucket

Каждая из них обладает своими плюсами и минусами. Для выбора была составлена сравнительная таблица функционала, который потребуется для разработки проекта.

	GitHub	GitLab	Bitbucket
Модель хостинга	Только облачный	Облачный + самостоятельный (открытый исходный код)	Только облачный

	GitHub	GitLab	Bitbucket
Приватные репозитории	Да, бесплатно	Да, бесплатно	Да, бесплатно
CI/CD	Есть, GitHub Actions, большой Marketplace расширений	GitLab CI/CD, можно подключать свои Runner, нет готовых плагинов	Pipelines, Pipes от Atlassian (не так много)
Code Review	Pull Requests, есть AI проверка	Merge Requests	Pull Requests
Issue-трекинг	Встроенный	Встроенный	Встроенный
Интеграции	Широкая инфраструктура, есть GitHub Projects для полноценного ведения проекта	Есть аналог GitHub Projects (Issue Boards)	Нет
Управление доступом	Роли	Группы и уровни	Гибкие права
Безопасность и проверка кода	Dependabot, секреты	SAST/DAST, секреты	Сканирование, секреты
Сообщество и популярность	Самый крупный	Набирает популярность, в основном корпоративно	Теряет популярность

Таблица 5. Сравнение GitHub, GitLab и Bitbucket

Как видно из таблицы 5, GitHub и GitLab имеют весь необходимый функционал и не отстают друг от друга, Bitbucket сильно отстает от конкурентов. GitLab имеет важное преимущество – возможность размещать сервис полностью на своей инфраструктуре (self-hosted), что полезно для крупных компаний с корпоративными секретами, однако не имеет особого смысла в небольших проектах. В свою очередь, облачная версия GitHub является более удобной, сильно популярнее любых конкурентов, CI/CD процессы обладают огромным количеством готовых плагинов в GitHub Marketplace. Ещё одним важным направлением развития GitHub является интеграция нейросетей на всех этапах, GitHub Copilot добавляет подсказки кода в IDE, позволяет автоматически проверять и комментировать PullRequest'ы, и даже в beta режиме полностью самостоятельно писать код по поставленным задачам.

В GitHub была создана отдельная организация, в которой расположены все репозитории связанные с проектом.

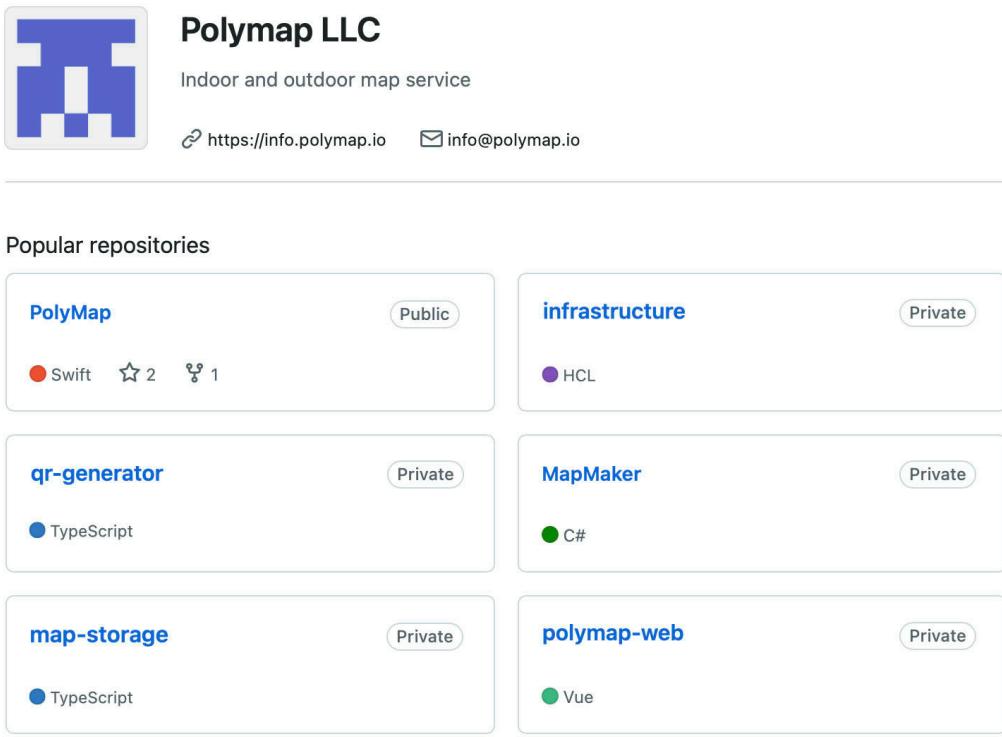


Рис. 7. Организация Polymap в GitHub

3.2 Веб-карта

Визитной карточкой всего сервиса является веб-карта, именно с ней будет взаимодействовать большинство пользователей и именно с ней будет начинаться знакомство с сервисом новых заказчиков. По этому крайне важно, реализовать её максимально качественно. Основная задача веб-карты – это динамически отображать карты разных кампусов в формате Extended-IMDF (баз разработан в рамках бакалаврской ВКР), а так же предоставлять пользователям возможность взаимодействовать с картами, строить маршруты, открывать аннотации, предоставлять поиск по аннотациям. Важной задачей является сделать поддержку не только компьютерной но и мобильной версии с адаптированным интерфейсом и управлением.

3.2.1 Выбор основного фреймворка

Карта является однострочным веб-приложение (SPA), и имеет много интерактивных элементов, что означает, что для её реализации необходимо использовать веб-фреймворк, ввиду того, что карта будет активно обновляться и развиваться, к этому выбору стоит подойти с особой тщательностью. К основным фреймворкам, которые специализируются на разработке SPA относятся:

- React – самый популярный фреймворк на текущий момент. Имеет огромное количество готовых библиотек, компонентов, плагинов и поддержку сообщества. Позиционируется как библиотека, что несколько снижает удобство разработки, что компенсируется сторонними расширениями. Разработан и поддерживается крупной мировой компанией Meta (Facebook).
- Vue – второй по популярности фреймворк, который активно развивается и является главным конкурентом React. С самого начала позиционируется как фреймворк, что позволило сделать более удобную работу с компонентами. Разрабатывается сообществом. Имеет много готовых библиотек и компонентов, однако их количество значительно уступает React.
- Angular – устаревший фреймворк, который был одним из первых, но на текущий момент сильно уступает React и Vue. Создавать новые проекты на нём не рекомендуется.

- Svelte – новый фреймворк, который активно развивается и имеет много интересных идей. Популярность и количество библиотек сильно уступает React и Vue, большой акцент делается на производительность и простоту использования. Больше предназначен для небольших и быстрых сайтов, нежели для крупных веб-приложений.

На первый взгляд, очевидным выбором будет React, однако это не совсем так, и стоит углубиться в детали. Важным преимуществом Vue является система реактивности, и начиная с третьей версии был добавлен `composition api`, который вывел использования рекактивности на новый уровень. С `composition api` система реактивности больше не привязана к компонентам, что позволяет использовать её в любых частях приложения и качественно декомпозировать код. Кроме того, в отличие от React, в котором изменение реактивного состояния приводит к перерисовки (redraw) компонента в котором это состояние объявлено и всех его дочерних в глубину, изменение состояния Vue приводит к перерисовке только тех компонентов, которые это состояние используют, что в разы эффективнее с точки зрения производительности. В React в данный момент тоже добавляют аналогичный функционал (React Signals), однако, на момент написания работы, он ещё в экспериментальной стадии и библиотеки его не поддерживают. В свою очередь, по остальному функционалу, Vue не отстает от React, и выбор между этими двумя инструментами становится более сложным. Для решения этой задачи, было принято решения воспользоваться системой поддержки принятия решений, которая позволяет выбрать оптимальный инструмент для решения задачи.

Фреймворки оценивались по следующим критериям:

- Производительность рендеринга – насколько эффективно фреймворк обновляет DOM. Использование Virtual DOM, скорость перерисовки, способность обрабатывать большое количество изменений.
- Размер бандла – какой размер занимает фреймворк после компиляции. Важен для скорости загрузки страницы.
- TypeScript – насколько хорошо фреймворк поддерживает TypeScript, написан ли он сам на TS.
- Встроенные возможности – наличие встроенных возможностей, таких как маршрутизация, управление состоянием. Можно ли пользоваться фреймворком без сторонних библиотек.
- Реактивность (model) – удобство и эффективность работы реактивности
- Минимальные зависимости – сколько дополнительных зависимостей (библиотек) используется во фреймворке. Чем меньше зависимостей, тем меньше вероятность появления ошибок и уязвимостей.
- Стабильность API – устойчивость фреймворка в долгосрочной перспективе, совместимость версий. Открытый или закрытый проект.
- Инструменты разработки – наличие и удобство инструментов разработки, таких просмотр компонентов и состояний.
- Документация, сообщество – насколько хорошо написана документация, размер и активность сообщества, количество библиотек.

С использованием 5 алгоритмов был рассчитан общий балл для каждого облачного провайдера. В Приложение Б. находится подробный отчёт СППР.

Вариант	Дом	Блок	Тур	Sjp	Sjm	ИТОГО	Место
Vue3	4	3	4	4	3	18	1
React	3	2	3	3	2	13	3
Angular	1	1	1	1	1	5	4
Svelte	2	4	2	2	4	14	2

Таблица 6. Результат работы СППР выбора фреймворка

Как видно из таблицы 6, Vue3 является самым оптимальным выбором для реализации веб-карты с большим отрывом. React и Svelte набрали примерно одинаковое количество баллов, Svelte вышел вперёд за счёт высокой производительности и отсутствию внешних зависимостей. Angular сильно отстает от всех остальных фреймворков, и его использование не рекомендуется.

Таким образом, для реализации интерфейса веб-карты был выбран Vue3.

3.2.2 Отображение карты

Выбранный фреймворк Vue3 позволяет удобно работать с компонентами пользовательского интерфейса, однако кроме интерфейса, приложение должно отображать интерактивные карты. Для этого можно использовать готовую картографическую библиотеку, такую как MapboxGL, Leaflet, OpenLayers и другие. Однако, у всех библиотек очень ограничен функционал добавления наложения (overlay) слоёв и аннотаций. В основном, функционал наложения в таких библиотеках предназначен для отображения небольших полигонов или одиночных маршрутов, в случае с PolyMap, карты состоят из тысяч полигонов, которые необходимо отображать одновременно. Для такого необходимо использовать графический ускоритель (GPU), из браузера это можно сделать с помощью WebGL. Поддержку WebGL наложений поддерживает всего несколько библиотек: MapboxGL, MapLibreGL, GoogleMaps (позволяет получить контекст WebGL и рисовать самостоятельно). Однако, свои аннотации со своими анимациями делать крайне проблематично на всех библиотеках.

Исходя из того, что основным функционалом PolyMap является отображение карты, было принято решение не использовать сторонние библиотеки, а разработать своё решение адаптированное под нужды и задачи PolyMap, которая будет наиболее эффективно выполнять задачи сервиса. Кроме того, своё собственное решение позволит не добавлять лишний функционал, который замедляет общую работу приложения.

Для оптимального отображения карты необходимо использовать WebGL, так как он позволяет использовать графический ускоритель (GPU). Работать с WebGL можно напрямую через контекст и низкоуровневый API, а можно использовать минимальные библиотеки, которые уже реализуют базовый графический Pipeline. Наиболее популярной и лёгкой библиотекой является ThreeJS – она позволяет работать с 3D графикой, а так же предоставляет удобный интерфейс для работы с WebGL. В текущей версии PolyMap, карта имеет 2D отображение, чего можно достигнуть ортографической проекцией сверху вниз. В будущих обновлениях, планируется добавить поддержку 3D отображения карты, что является ещё одним аргументом использования 3D pipeline'а с самого начала.

Общий процесс отрисовки 3D графики в WebGL состоит из следующих этапов:

1. В видеокарту загружаются

- вершинный шейдер (vertex shader) – программа на GLSL, которая выполняется на GPU и отвечает за преобразование вершин в экранные координаты. Вызывается для каждой вершины от буфера вершин (vertex buffer) в пространственных координатах
- фрагментный шейдер (fragment shader) – программа на GLSL, которая выполняется на GPU и отвечает за определение цвета пикселя на экране. Вызывается для каждого пикселя внутри треугольника в экранных координатах
- юниформы (uniforms) – глобальные переменные, которые передаются в шейдеры и не меняются в процессе отрисовки. Например, матрица проекции, матрица вида, текстуры, и другие пользовательские данные, определённые в шейдере

2. В видеокарту загружается буффер вершин (vertex buffer), который может быть индексированным или не индексированным. Индексированный буффер состоит из массива параметров вершин и индексов, которые указывает на вершины для объединения в треугольники. У каждой вершины может быть сколько угодно параметров, которые попадут на вход вершинного шейдера.
3. Выполняется вызов отрисовки (draw call), который указывает, что необходимо запустить отрисовку с использованием загруженных шейдеров и буферов.
4. Весь процесс повторяется для следующего набора вершин и шейдеров.

В ThreeJS, для упрощения работы с WebGL, эти шаги автоматизированы и низкоуровневые сущности представленные в виде более высокоуровневых абстракций:

- **Material** – материал, он инкапсулирует в себе вершинный и фрагментный шейдеры и все их параметры (uniforms)
- **Geometry** – геометрия, она определяет набор вершин и их параметры
- **Mesh** – меш, он объединяется **Material** и **Geometry**, с него и начинается процесс отрисовки.

Кроме того, ThreeJS предоставляет абстракцию **Scene**, в которую необходимо добавить все **Mesh**, которые необходимо отрисовать. После чего, вызвав функцию **render**, для всех **Mesh** в **Scene** поочерёдно будет вызвана описанная выше последовательность шагов отрисовки.

С точки зрения производительности, бутылочным горлышком процесса отрисовки является количество вызовов отрисовки (draw calls). Каждый вызов отрисовки требует переключения контекста GPU, очисткой старой памяти и загрузкой новой (переход данных от CPU к GPU, инициализация этого процесса медленная), что является дорогостоящей операцией. Главным способом уменьшения количества вызовов отрисовки является батчинг (batching) – объединение нескольких небольших объектов в один большой. В ThreeJS это можно сделать с помощью **InstancedMesh**, который позволяет отрисовать несколько одинаковых объектов за один вызов отрисовки. Такое объединение выполняется на уровне **Geometry** и накладывает ограничение на использование только одного материала (у всех объектов должны быть одни и те же шейдеры, текстуры и параметры). В PolyMap реализован батчинг по категориям Unit'ов, то есть, все многоугольники одного типа (например газон, дороги, здания) будут объединены в одну **Geometry** и присвоены одному **Mesh**. Это возможно благодаря тому, что они отображаются одинаковым цветом, а их пересечения могут накладываться друг на друга и отображаться за один проход. На Рис. 8 показан процесс отрисовки карты по шагам, на практике, каждый шаг накладывается поверх предыдущего, и в результате получается общая карта. Вся карта отображается за 10-15 шагов в зависимости от наличия многоугольников определённых типов на карте. Благодаря батчингу, количество вызовов снижено с примерно 3000 (в зависимости от размера карты) до постоянных 10-15 (в зависимости от количества категорий).

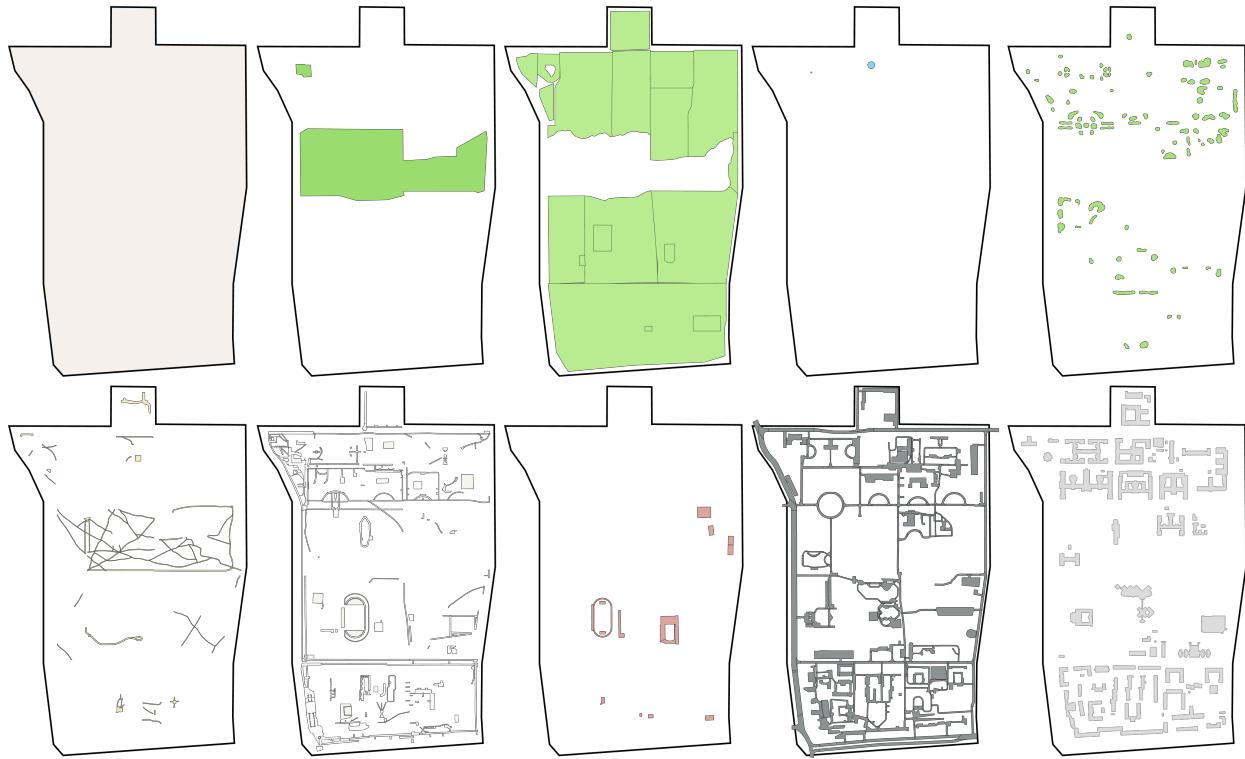


Рис. 8. Процесс отрисовки карты по шагам DrawCalls

Такая система отрисовки показала себя крайне эффективной, время на один кадр на компьютере 2019 года (с видеокартой Radeon Pro Vega 48) в FullHD разрешении составляет 1.29мс, при лимите в 16.67мс на кадр (60 FPS). Измерения проводились с помощью встроенных инструментов Chrome DevTools, на Рис. 9 видно, что время вызова функции `requestAnimationFrame`, которая каждый кадр запускает отрисовку карты, составляет 1.29мс.

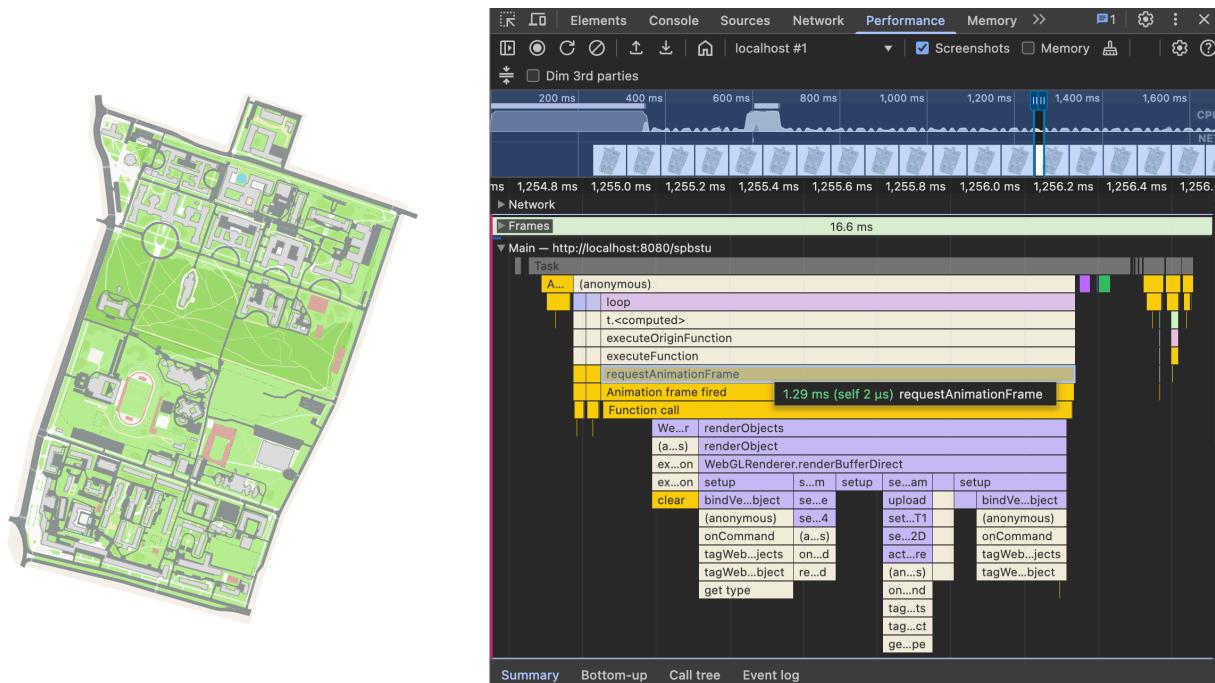


Рис. 9. Диаграмма времени отрисовки одного кадра карты

3.2.3 Отображение аннотаций

После общей геометрии карты, по значимости идут аннотации, они добавляют на карту информативности, с их помощью подписываются здания, входы, парковки, номера кабинетов,

столовые. По аннотациям пользователь сможет проводить поиск и строить маршруты между ними. В отличие от статичной геометрии карты, аннотации являются частью пользовательского интерфейса. Каждая аннотация привязана к точке на карта в экранном пространстве. Размер и горизонт аннотации является неизменным при масштабировании и вращении карты. На аннотациях присутствует текст, по аннотациям пользователь может кликать. Все эти факторы накладывают сильные ограничения на систему отрисовки аннотаций.

Основной способ отображение 2D пользовательского интерфейса в вебе – это использование HTML элементов, в случае с аннотациями, нужно было бы использовать аннотации в абсолютных координатах, которые бы обновлялись каждый кадр. Однако, это крайне неэффективно, так как время отображение HTML элементов экспоненциально растёт с количеством элементов. По этому, в PolyMap аннотации отображаются с помощью Canvas API, который позволяет императивно рисовать 2D графику на растровом холсте, в этом случае, производительность зависит от количества элементов линейно.

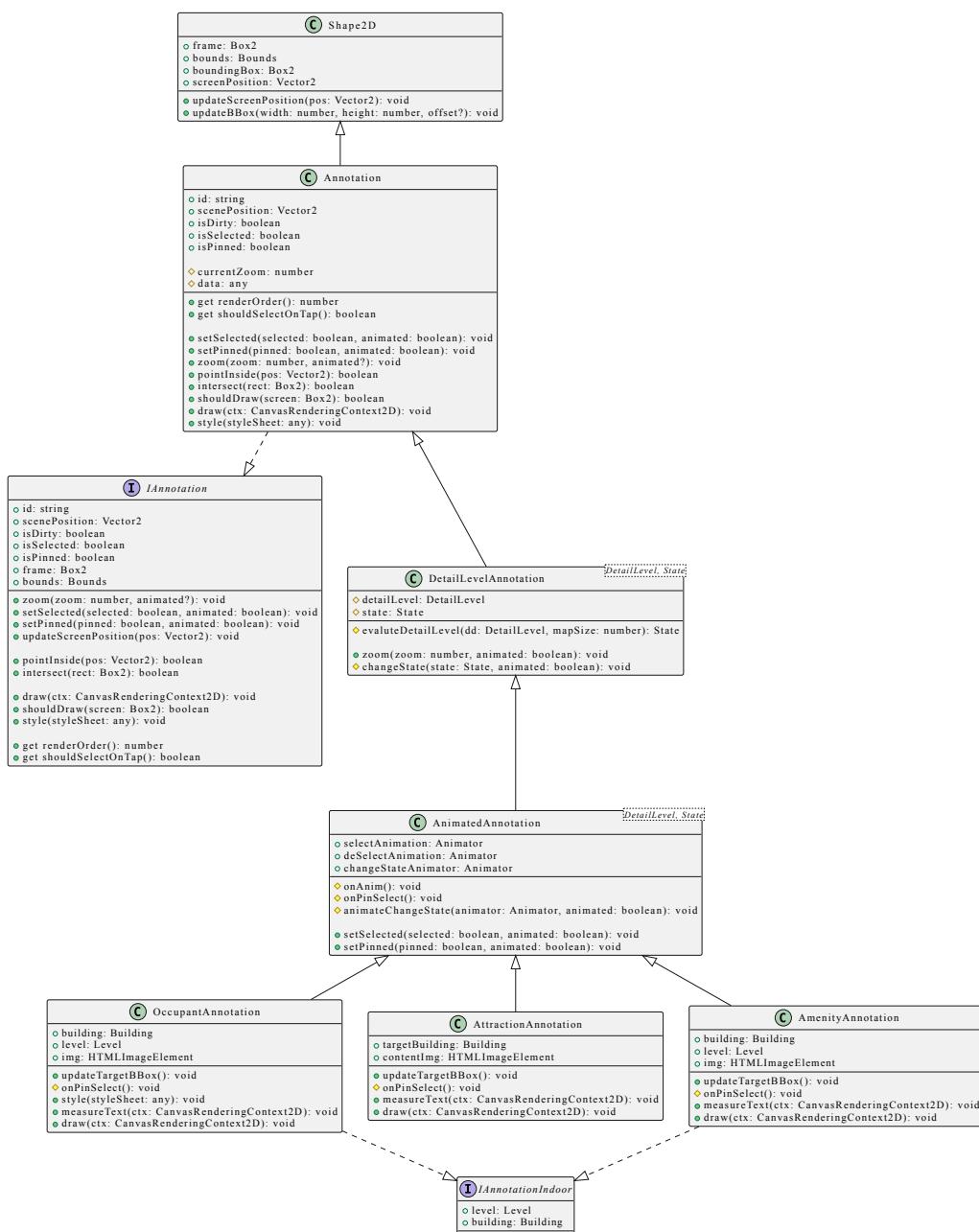


Рис. 10. Диаграмма классов системы аннотаций

В сервисе присутствует три вида аннотаций:

- Occupant Annotation – самая базовая аннотация, с её помощью отображаются предназначение кабинетов внутри здания. Например номер лектория, название столовой, конференц-зала, и т.д.
- Amenity Annotation – аннотация, которая отображает дополнительные удобства внутри здания. Например, лестницы, лифты, входы и выходы, а также удобства вне здания, такие как парковки, велопарковки, спортивные площадки и т.д.
- Attraction Annotation – аннотация, которая отображает точки интереса вне здания. В первую очередь, такими аннотациями помечаются учебные корпуса.



Рис. 11. Виды аннотаций

У каждой из аннотаций есть несколько состояний, между которыми она может переключаться, например, аннотация может быть активной (выделенной пользователем), закреплённой (в случае прокладывания маршрута) или в минимизированном состоянии (отображается маленькой точкой). Все аннотации наследуются от класса `AnimatedAnnotation`, который упрощает переходы между состояниями. У аннотаций есть уровень детализации, который определяет значимость аннотации. Базовый класс `DetailLevelAnnotation` управляет состоянием в зависимости от текущего контекста карты, учитываются состояния аннотации, масштаб карты, наличие аннотаций поблизости. В конечном итоге, все аннотации являются потомками класса `Annotation`, который реализует интерфейс `IAnnotation`, через который, аннотации добавляются на карту.



Рис. 12. Виды состояний аннотаций

Ответственность за отображение аннотаций в разных состояниях лежит на конечной реализации трёх видов аннотаций, которые, переопределяют метод `draw(ctx: Context2D)`, кроме этого, каждая аннотация должна определить свой `BoundingBox` – прямоугольник, внутри которого будет отрисовываться и `frame` – прямоугольник, по нажатию на который, аннотация будет активироваться. Механизмы наследования позволяет переопределить метод `pointInside(position: Vector2D)` для определения является ли точка внутри аннотации для её активации, однако, в текущей реализации, достаточно использовать `frame` и стандартную реализацию метода `pointInside`.

Процедурная отрисовка графики на процессоре медленнее чем отрисовка графики на видеокарте, особенно медленной операцией является отрисовка текста. Результат такой отрисовки будет в разы качественнее, чем любая альтернатива на видеокарте, именно процессорные алгоритмы позволяют получить наиболее качественное сглаживание. При масштабировании карты, аннотации двигаются друг относительно друга, а это означает, что каждую аннотацию нужно каждый кадр перерисовывать в новых координатах.



Рис. 13. Пример двух текстурных атласов

Для оптимизации процесса отрисовки было применено кеширование (запекание) аннотаций и текстовых подписей в текстурные атласы (Рис. 13). При повторной перерисовке, аннотация не выполняет медленную отрисовку текста и картинки, а просто копирует область пикселей из текстурного атласа в нужные координаты целевого холста (Рис. 14), такое копирование является крайне быстрой операцией. Если места на атласе недостаточно, то создается новый. Красной рамкой на Рис. 13 и Рис. 14 отображаются границы отдельных областей на атласе. У аннотаций в активном и закрепленном состоянии, а так же, во время анимации перехода между состояниями, отрисовка происходит в обычном режиме, так как в таких состояниях не отображаются много аннотаций одновременно, и медленная отрисовка не будет заметна пользователю. Кешировать необходимо только те состояния, которые отображаются на экране в больших количествах.

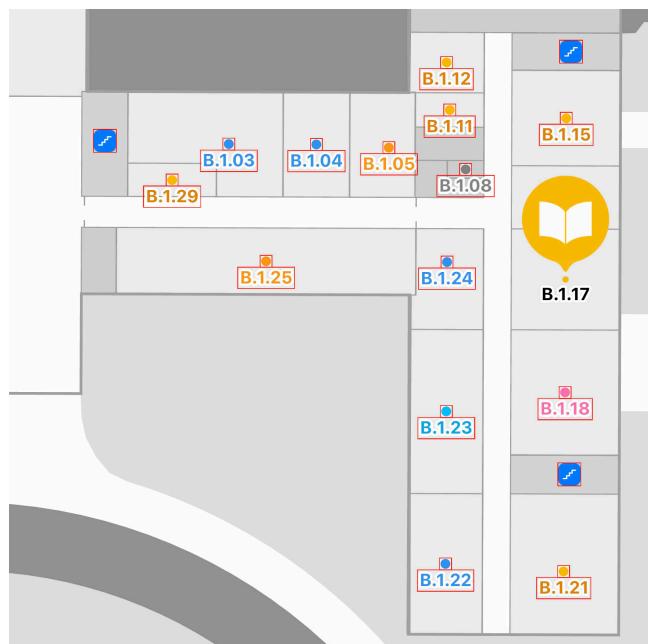


Рис. 14. Пример отображения закешированных аннотаций

3.2.4 Система анимаций

Между состояниями аннотации переходят с анимацией. При процедурной отрисовке, анимации тоже необходимо делать процедурно, то есть менять параметры отрисовки и перерисовывать каждый кадр анимации (по умолчанию 60 FPS). Для этого, в PolyMap применяется твиннер (tweener) – это класс, который управляет анимациями. Твиннер позволяет декларативно задать анимацию путём указания конечного состояния параметра, времени анимации и функции интерполяции, а после, в произвольный момент времени, запустить анимацию из текущего состояния. Такая система позволяет перейти из любого состояния в любое другое, и даже во время проигрывания одной анимации, её можно прервать на половину и запустить другую. Например, пользователь выбрал аннотацию, и до завершения анимации перехода, отменил выделение, вместо ожидания завершения анимации открытия и только после запуска анимации закрытия, твиннер прервёт анимацию открытия на том моменте, на котором она находилась, и из этого стартового положения запустит анимацию закрытия. Это делает карту более отзывчивой для пользователя.

В качестве функции интерполяции, в PolyMap используется механизм пружинных анимаций (spring animation). При этом подходе, анимация проигрывается так, как будто объект прикреплён к пружине, которая тянет его к конечному состоянию. В этом случае, траектория движения наиболее приближена к реальности, что делает анимацию более естественной. К сожалению, невозможно одной формулой описать пружинную анимацию, для каждого набора параметров, необходимо генерировать новую формулу интерполяции.

Пружинная анимация подчиняется классическому дифференциальному уравнению затухающего гармонического осциллятора для массы $m = 1$

$$m\ddot{x} + c\dot{x} + kx = 0, m = 1 \quad (1)$$

$$\ddot{x} + c\dot{x} + kx = 0 \quad (2)$$

Введём две величины: собственная частота без затухания $\omega = \sqrt{k}$ (частота отклика), коэффициент демпфирования: $\zeta = \frac{c}{2\sqrt{k}}$

Подставим $c = 2\zeta\omega$ и $k = \omega^2$:

$$\ddot{x} + 2\zeta\omega + \omega^2x = 0 \quad (3)$$

Найдём решение в форме характеристического уравнения $x(t) = e^{rt}$

$$r^2 + 2\zeta\omega r + \omega^2 = 0 \Rightarrow r_{1,2} = -\zeta\omega \pm \omega\sqrt{\zeta^2 - 1} \quad (4)$$

Величина $\Delta = \zeta^2 - 1$ даёт нам 3 решения:

Недодемпированный $\zeta < 1$:

$$\omega_d = \omega\sqrt{1 - \zeta^2}, x(t) = e^{-\zeta\omega t} \left[x_0 \cos(\omega_d t) + \frac{v_0 + \zeta\omega x_0}{\omega_d} \sin(\omega_d t) \right] \quad (5)$$

Критически демпфированный $\zeta = 1$:

$$x(t) = e^{-\omega t} [x_0 + (v_0 + \omega x_0)t] \quad (6)$$

Передемпфирированный $\zeta > 1$: Реальные отрицательные корни:

$$r_{1,2} = -\zeta\omega \pm i\omega\sqrt{\zeta^2 - 1}, (r_1 < r_2 < 0) \quad (7)$$

Получим:

$$w_d = \omega\sqrt{\zeta^2 - 1}, x(t) = e^{-\zeta\omega t}[c_1 e^{w_d t} + c_2 e^{-w_d t}] \quad (8)$$

где

$$c_1 = \frac{v_0 + x_0(\zeta\omega + w_d)}{2w_d}, c_2 = x_0 - c_1 \quad (9)$$

Для упрощения в коде введём: $\lambda = \frac{c}{2m} = \zeta\omega$, полный исходный код реализующий пружинную анимацию на TypeScript (Приложение В.).

Примеры работы функции пружинной анимации приведены на Рис. 15 и Рис. 16. На них показано, как меняется кривая анимации в зависимости от изменения коэффициента демпфирования и частоты отклика. Как видно из графиков, продолжительность анимации зависит от частоты отклика, а амплитуда колебаний («пружинность») от коэффициента демпфирования. При передачи функции интерполяции в твиннер, необходимо учитывать, что пружинная анимация является бесконечно затухающей, а значит, нужно подбирать время анимации так, чтобы обрезать колебания в моменте, когда они становятся менее одного пикселя.

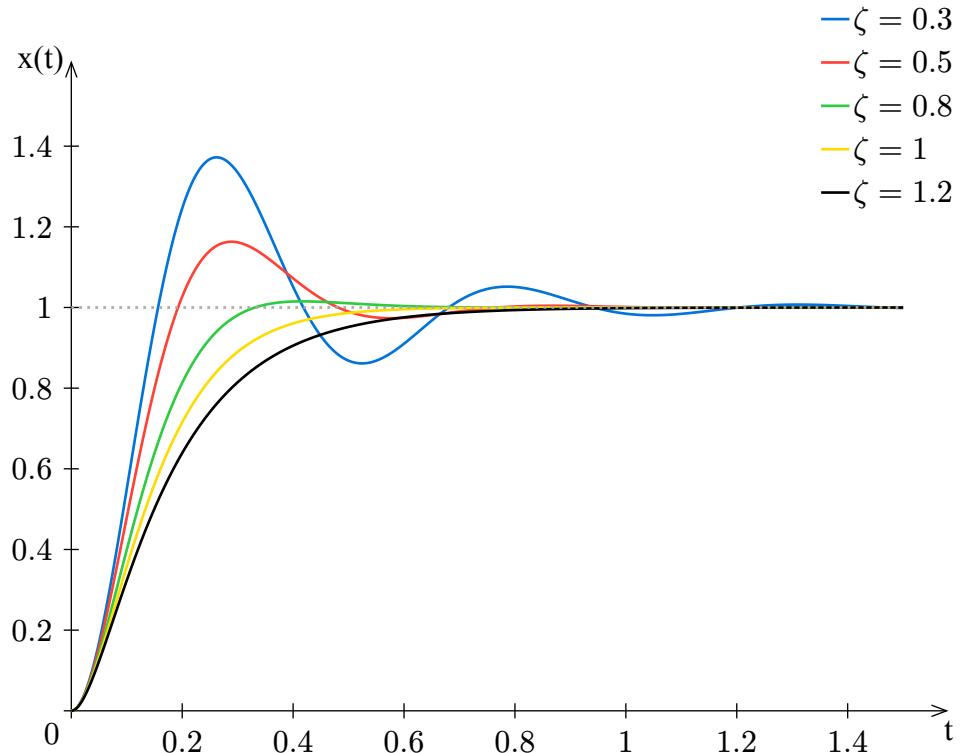


Рис. 15. Изменение коэффициент демпфирования ζ при частоте отклика $\omega = 0.5$

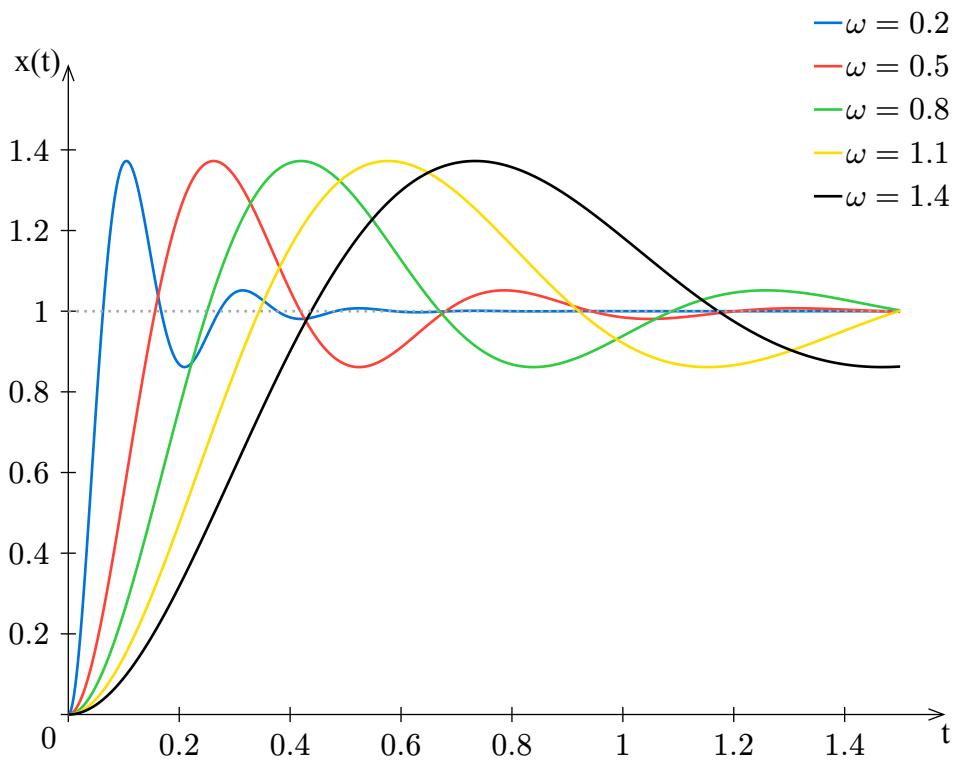


Рис. 16. Изменение частоты отклика ω при коэффициенте демпфирования $\zeta = 0.3$

3.2.5 Поиск маршрутов

Поиск маршрута в PolyMap осуществляется по графу маршрутов с весами (Рис. 17). Поиск по графу можно осуществлять с помощью разных алгоритмов, наиболее оптимальным и популярным является алгоритм A* (A-star), однако, на практике он плохо работает с неравномерными в пространстве графами с весами (хорошо работает по сетке). По этому, в PolyMap для поиска маршрута используется алгоритм Дейкстры, который позволяет находить гарантированно кратчайший путь по графу между двумя точками. Размер графа маршрутов внутри кампуса достаточно мал, по этому проблем с производительностью не возникает.

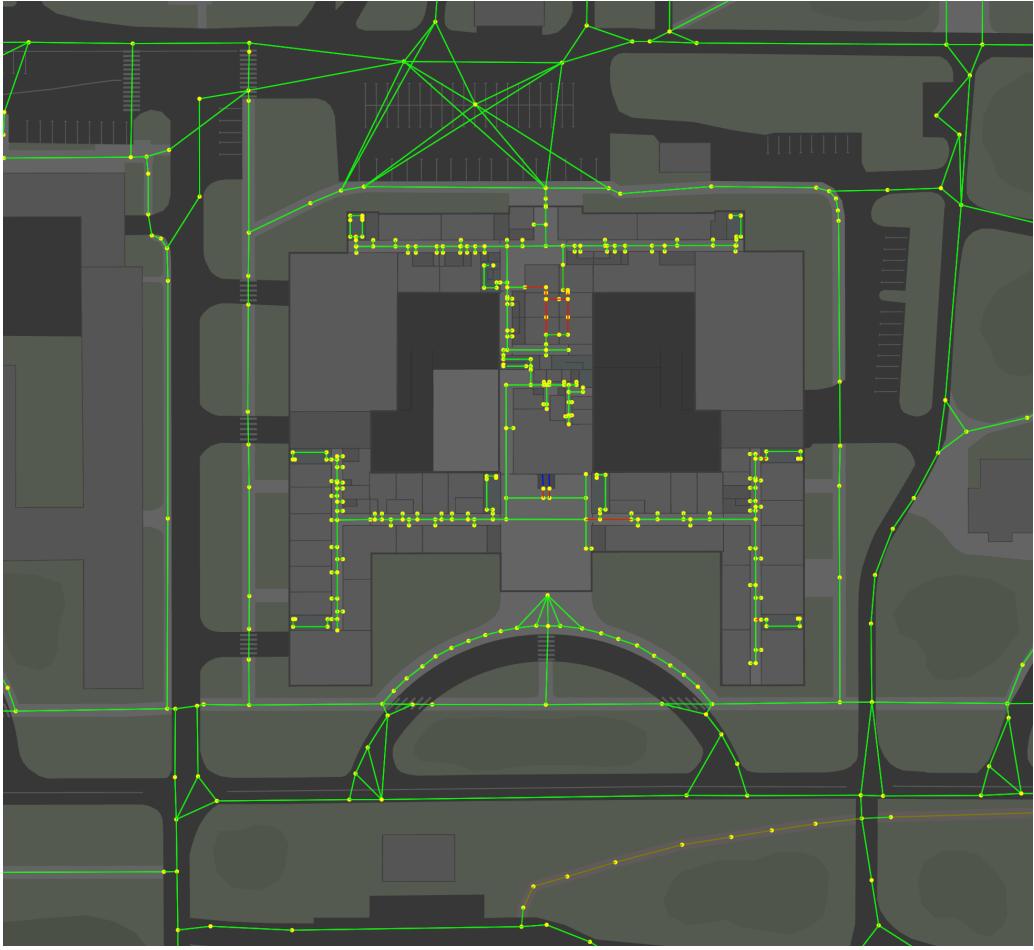


Рис. 17. Пример графа маршрутов

При поиске маршрута с улицы в помещение, поиск происходит в два этапа:

1. Поиск маршрута от точки на улице до входа в здание
2. Поиск маршрута от входа в здание до конечной точки внутри здания

Это позволяет снизить число вершин в графе и ускорить поиск, даже если на одной карте будет много зданий, то при поиске маршрута будут учитываться только граф вне здания, и график внутри целевого здания. Если здание имеет несколько входов, то будут перепробованы все входы, и отображаться будет наиболее оптимальный маршрут. Отображение маршрута происходит в системе аннотаций на Canvas, так как маршрут всего один, значительную нагрузку на CPU он не оказывает и в оптимизации отрисовки не нуждается.

Для расчёта времени маршрута применяется формула 10, где T – время в секундах, w_k – вес отрезка, d_k – длина отрезка, $\text{type}[k]$ – тип отрезка (внутри здания, вне здания, лестница), v – скорость передвижения по отрезку определённого типа C – количество переходов между улицей и зданием, t_C – время перехода между улицей и зданием.

$$T = \sum_{k=0}^n \left[\frac{w_k * d_k}{v_{\text{type}[k]}} \right] + C * t_C \quad (10)$$

Экспериментально выяснено, что скорость передвижения вне здания составляет 4.5 км/ч, внутри здания 3.5 км/ч, по лестнице 1.5 км/ч, а время перехода между улицей и зданием составляет $t_C = 60$ с.

3.2.6 Энергоэффективность

Так как приложение предназначено не только для компьютеров, но и для мобильных устройств, необходимо учитывать энергопотребление. Использование графический ускорителей на мобильных устройствах задействует высокопроизводительные ядра системы на чипе, что приводит к повышенному энергопотреблению. Для снижения энергопотребления, в PolyMap реализована условная перерисовка карты: во время взаимодействия с картой и анимаций, например при прокрутке или изменение масштаба, карта перерисовывается с частотой 60FPS, а в статичном состоянии, карта не перерисовывается, а оставляет последний кадр, что позволяет снизить число обращений к графическому ускорителю примерно в 2-3 раза (на основе практических измерений взаимодействия с картой). Так же, в сценариях поиска по аннотациям и использования прочего пользовательского интерфейса, число вызовов отрисовки снижается в 10 раз.

На Рис. 18 показан профайлер Chrome DevTools, на котором отображается 8 секунд использования приложения, за это время, было проведено два панорамирования карты и одно приближение. На верхней шкале отображается общая нагрузка, из графика видно, что нагрузка присутствует во время взаимодействия с картой, и практически отсутствует между действиями. Из шкаллы Frames видно, что между взаимодействиями, частота кадров замораживалась, а во время взаимодействия, частота кадров была 60 FPS (график зелёный, в случае превышение нагрузки и падения ниже 60 FPS, кадры отображаются оранжевым цветом). Аналогичная картина видна и на шкале использования основного ядра Main и графического ускорителя GPU. Хоть на них и присутствуют деления, они связаны не с отрисовкой карты, а с композицией HTML интерфейса (compositing), который оставался динамическим всё время использования приложения.

TODO: Возможно тут надо использовать картинку, а не PDF, тк фреймрейта не видно тк он скрин внутри pdf

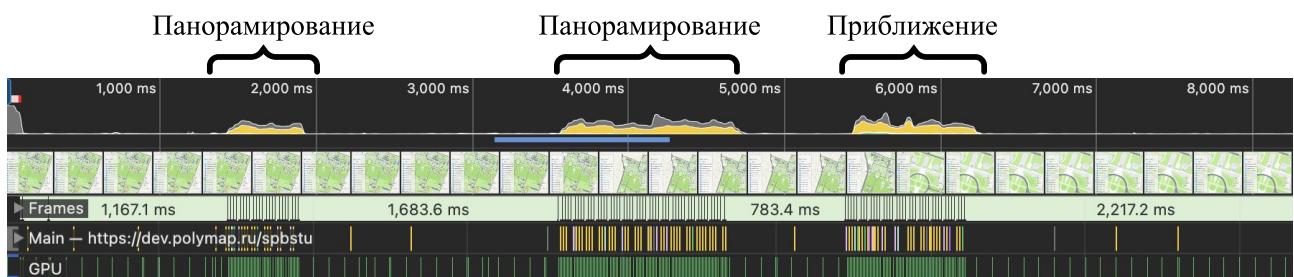


Рис. 18. Диаграмма нагрузки во время использования приложения

3.2.7 Хостинг

Несмотря на то, что веб-карта является веб приложением (фронтентом), его всё ещё необходимо хранить на сервере и раздавать пользователям по GET запросам на домен карты. В выбранной инфраструктуре Yandex Cloud, для хостинга статичных сайтов, можно использовать Yandex Object Storage (S3). Он позволяет настроить бакет в режиме HTTP сервера (Рис. 19), в этом случае, бакет станет доступен из интернета по служебному домену, после чего, в настройках DNS нужно будет добавить CNAME запись, которая будет указывать на бакет (Рис. 20). В этом случае даже не нужно использовать API Gateway, так как S3 будет раздавать статичные файлы по HTTP запросам.

Настройки

The screenshot shows the 'Website' tab in the 'Settings' section of the Yandex Cloud Object Storage interface. It includes fields for 'Main page' (index.html), 'Error page' (index.html), and a 'Link' (http://polymap.io.website.yandexcloud.net). The 'Hosting' tab is selected.

Рис. 19. Настройки Object Storage в режиме веб сайта

The screenshot shows the DNS configuration for a CNAME record. It maps 'polymap.io' to 'polymap.io.website.yandexcloud.net'. The 'Proxy status' is set to 'Enabled' and 'TTL' is 'Auto'.

Рис. 20. Настройки DNS для работы веб сайта на Object Storage

Чтобы не создавать излишнюю нагрузку на Object Storage (каждый запрос тарифицируется), рационально использовать CDN, для этого нужно загружать статичные файлы в бакет с указанием заголовков Cache-Control, в этом случае, Object Storage будет добавлять эти заголовки к ответам, и CDN будет кешировать файлы.

```
1 aws --endpoint-url=https://storage.yandexcloud.net/ \
2   s3 cp .output/public s3://polymap.io/ \
3   --recursive \
4   --cache-control 'max-age=864000' \
5   --profile polymap
```

Листинг 1. Пример загрузки в Object Storage с указанием заголовков Cache-Control

Такой хостинг получается крайне дешевым, а в большинстве случаев и бесплатным, так как используемые ресурсы не превышают бесплатный лимит. При этом, Object Storage гарантирует высокую доступность и отказоустойчивость, а также возможность масштабирования. В дополнение к этому, Yandex Cloud предоставляет продвинутый мониторинг полностью бесплатно, он настроен по умолчанию и позволяет отслеживать количество запросов, ошибки, время ответа.

Мониторинг

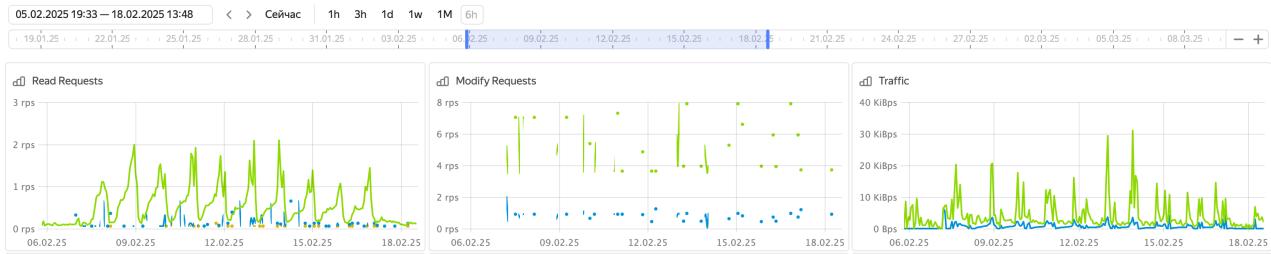


Рис. 21. Пример работы мониторинга Object Storage

3.3 Серверная часть

В сервисе используется микросервисная архитектура, и благодаря хорошей декомпозиции (Раздел 2.7.2), реализация каждого микросервиса в отдельности не представляет сложности и не обладает уникальными особенностями, о которых можно было бы рассказать. Однако, есть некоторые общие аспекты, которые применимы ко всем реализованным микросервисам. Рассмотрим их в этом разделе.

3.3.1 Стек технологий

Как упоминалось ранее, одним из ограничений Serverless подхода, является требование к быстрому холодному старту (время между началом запуска Docker контейнера и моментом, когда сервер будет готов слушать указанный порт). Это создаёт дополнительные нестандартные условия при выборе технологического стека. Благодаря использованию Serverless Containers, можно использовать любой стек который подойдёт для реализации, не ограничиваясь поддерживаемыми языками облака.

Наиболее популярные стеки для бекенда в 2025 году:

- Java + Spring Boot – постепенно теряет популярность, высокое время холодного старта
- C# + ASP.NET Core – популярный стек, высокое время холодного старта
- Python + Flask – набирает популярность, применяют в Serverless, но мало инструментов
- Node.js + Express – популярный стек, быстрый холодный старт, много инструментов адаптированных под Serverless

В качестве языка программирования для бекенда выбран TypeScript, это крайне популярный язык, у него много специалистов, хорошая поддержка, наибольшее число библиотек, он статически типизированный и обладает комфорным синтаксисом. При работе с бекенном, запускать TypeScript код принято через Node.js предварительно скомпилировав его в JavaScript. Существует множество фреймворков для разработки веб-сервера на Node.js. Благодаря хорошей модульности и изолированности частей проекта, никаких специфичных требований для построения больших архитектур к фреймворку не предъявляется. Основной критерий выбора – скорость холодного старта и поддержка сообщества (популярность).

Для начала рассмотрим рантайм выполнения JavaScript, на текущий момент существует три основных движка:

- Node.js – самый первый и наиболее популярный, использует движок V8
- Deno – новый движок, который использует движок V8, но с поддержкой ES модулей и TypeScript из коробки
- Bun – новый движок, с новым подходом, своей целью является максимальная производительность выполнения кода.

Было принято решение попробовать использовать современный и многообещающий Bun. Производительность достигается за счёт использования движка JavaScriptCore разработа-

ного Apple для Safari, движок с открытым исходным кодом, а так же, за счёт использования языка программирования Zig с применением низкоуровневых оптимизаций. Кроме того, Bun представляет из себя инструмент всё в одном, который включает в себя пакетный менеджер, сборщик, рантайм имеет поддержку тестов, запускает TypeScript и JSX из коробки, имеет ряд полезных встроенных библиотек (PostgreSQL driver, SQLite driver, S3 Cloud Storage driver, Redis client, WebSocket server (including pub/sub), HTTP server, HTTP router), каждая из этих библиотек является прослойкой к низкоуровневому и очень хорошо оптимизированному коду на Zig, в результате, все базовые возможности, работает в разы быстрее аналогов JS библиотек для Node или Deno.

Рантайм практически полностью совместим с API Node.js и подавляющее число Node.js библиотек будут корректно работать в Bun, все популярные библиотеки адаптированы под Bun, кроме того, вокруг Bun сложилась экосистема библиотек и инструментов, которые были разработаны с философией максимальной производительности, и достигают этого за счёт использования высокооптимизированный API Bun.

Тест	Bun	Deno	Node
Express.js 'hello world', rps	59 026	25 335	19 039
PostgreSQL 100 rows x 100 parallel queries, qps	50 251	11 821	14 398
WebSocket, messages sent per second	2 536 227	1 320 525	435 099

Таблица 7. Сравнение производительности Bun Node и Deno

Кроме рантайма нужно ещё выбрать фреймворк для удобства создания REST API сервера, большинство фреймворков предоставляют очень похожий синтаксис, так что наиболее важным критерием является производительность и скорость холодного старта. Наиболее популярные быстрые фреймворки:

- Fastify – один из самых быстрых фреймворков, поддерживает все популярные плагины, имеет хорошую документацию и сообщество
- Express – самый популярный фреймворк и наиболее медленный фреймворк
- Hono – наиболее быстрый фреймворк разработанный специально для Bun и Serverless
- Elysia – быстрый новый фреймворк, разработанный для Bun

Для выбора фреймворка, было проведено тестирование на предмет времени ответа в разных сценариях для холодного старта и горячего. Тестирование проводилось на реальной инфраструктуре Yandex Cloud, с использованием Serverless Containers. Результаты тестирования приведены в Приложение Г..

В ходе тестирования, было выяснено, что кроме фреймворка и рантайма, на время холодного старта влияет базовый образ Docker контейнера (а точнее его размер), а так же, применение бандлера и сам бандлер. Бандлер позволяет собрать весь исходный код в один файл (в том числе библиотеки), а так же минимизировать его. Это позволяет сократить время загрузки и парсинга исходного кода. Современные бандлеры используют механизм Tree Shaking, который позволяет удалить неиспользуемый код из бандла, что крайне эффективно при использовании библиотек. При сравнении были проведены тесты с использованием разных бандлеров (ESBuild, Parcel, Rollup, Webpack, Bun bundler) и без них. Тесты проводились на микросервисе генерации QR кодов и на тестовом микросервисе с большим числом библиотек, что делает результаты более репрезентативными.

В результате был выбран следующих подход:

- Bun в качестве рантайма

- TypeScript в качестве языка программирования
- Hono в качестве фреймворка
- Bun bundler в качестве бандлера
- Distroless в качестве базового образа Docker контейнера (является наиболее урезанным образом Linux, в котором нет ничего лишнего. Базовый образ занимает всего 9Мб вместо 16Мб у Alpine)

3.3.2 Шаблон микросервиса

Для удобства создания микросервисов, был создан шаблон микросервиса, который можно копировать и использовать в качестве основы для нового микросервиса. Шаблон включает в себя:

- Настроенный Bun проект, с пакетным менеджером, сборщиком
- Установленный Hono фреймворк и echo пример (отвечает на запросы GET /api/echp/:message сообщением Hello :message)
- Настроенный Dockerfile, который собирает проект в Docker образ
- Настроенный GitHub Actions, подробнее описан в Раздел 4.1.5
- Настроенный Terraform проект, который разворачивает микросервис в Yandex Cloud
 - Компиляция микросервиса в Docker образ
 - Загрузка Docker образа в Yandex Container Registry
 - Создание Serverless Container в Yandex Cloud
 - Создание API Gateway в Yandex Cloud. Описан шаблон OpenAPI спецификации.
- Настроенные ESLint для оценки качества кода, подробнее в Раздел 4.2.1
- Настроенные модульные тесты с использованием bun test, подробнее в Раздел 4.3
- Настроенные Cypress E2E тесты, подробнее в Раздел 4.5

Таким образом, для создания нового микросервиса, достаточно скопировать шаблон, переименовать название проекта, и подключить его к организации в GitHub. Сразу после первого коммита, запустится CI/CD процесс, который развернёт этот микросервис в тестовом окружении.

После чего, программисту, остаётся только писать реализацию кода, модульные и E2E тесты, и вносить изменения в OpenAPI спецификацию. Все настройки, конфигурации и инфраструктура уже готова в шаблоне и не требует дополнительных действий. Такой подход значительно ускоряет разработку, не требует повторения рутинных действий, и гарантирует единообразие кода по всему проекту.

3.3.3 Раздача карты и CDN

Одной из особенностей архитектуры PolyMap, является оптимизация раздачи статических файлов через CDN. В случае с фронтеном, фреймворк Vue, в момент компиляции генерирует скрипты, с названиями в виде хуш-функций от их содержимого, что делает их полностью статическими файлами, никаких сложностей с использованием CDN не возникает, эти файлы хранятся в Yandex Object Storage с указанием заголовков Cache-Control, на время которого и кешируются в CDN. При обновлении фронта, случается его перекомпиляция, и изменённые файлы результата имеют другое название, что позволяет отличить их от старых закешированных версий.

Однако, кроме фронта, значимое потребление трафика происходит при раздаче файлов карт, этот процесс тоже оптимизирован через CDN, но карты не являются полностью статическими файлами. В случае изменения (обновления) карты, название файла остается прежним, и необходим способ максимально быстро доставить новую карту до пользователей.

Один из подходов сделать это – использовать короткое время кеширования, например 1 час, в этом случае, при обновлении карты, пользователи получат новую версию в течение одного часа после обновления. Такой подход решит проблему, но создадут излишнюю нагрузку на сервер и замедлит часть запросов, которые будут идти в обход кеша, в большинстве случаев, карты меняются сильно реже одного часа. По этому, в PolyMap используется другой подход, при обновлении карты: после обновления, микросервис `map-storage`, который отвечает за управление картами, отправляет в CDN систему по API запрос с указанием URL карты, кеш которой нужно инвалидировать. CDN система удаляет кеш для этого URL, и при следующем запросе, будет получена и закеширована новая версия карты. Это позволяет мгновенно доставлять до пользователей обновления, при этом оставляя большое время кеширования.

```
1 curl -X POST \
2   "https://api.cloudflare.com/client/v4/zones/$ZONE_ID/purge_cache" \
3   -H "Authorization: Bearer $CF_API_TOKEN" \
4   -H "Content-Type: application/json" \
5   --data '{ "files": [ "https://map-storage.polymap.io/$MAP_ID" ] }'
```

Shell

Листинг 2. Пример запроса на инвалидирование кеша в CDN

3.4 Мониторинг

Мониторинг – важная часть любого программного продукта, крайне важно не только опубликовать систему, а ещё и отслеживать её состояние, время ответов, ошибки, число пользователей и нагрузку на систему. В случае возникновения ошибок, полезно иметь логирование, по которому можно будет понять, что именно привело к ошибке.

При использовании Serverless подхода в Yandex Cloud, мониторинг и логирование уже встроены в облако и не требуют дополнительных действий для настройки. Кроме того, мониторинг является полностью бесплатным, логирование тарифицируется по количеству хранимых записей, в большинстве случаев, хватает бесплатных квот.

3.4.1 Логирование

Для логирования используются Yandex Cloud Logging, которые позволяют:

- Хранить логи от всех внутренних ресурсов Yandex Cloud
- Создавать лог-группы, в которые можно отправлять логи из внешних источников
- Использовать фильтры по логам
- Поддерживает структурированные JSON логи
- Логи хранятся в Object Storage
- Есть интеграция с Grafana
- Можно настроить права доступа к логам от разных групп

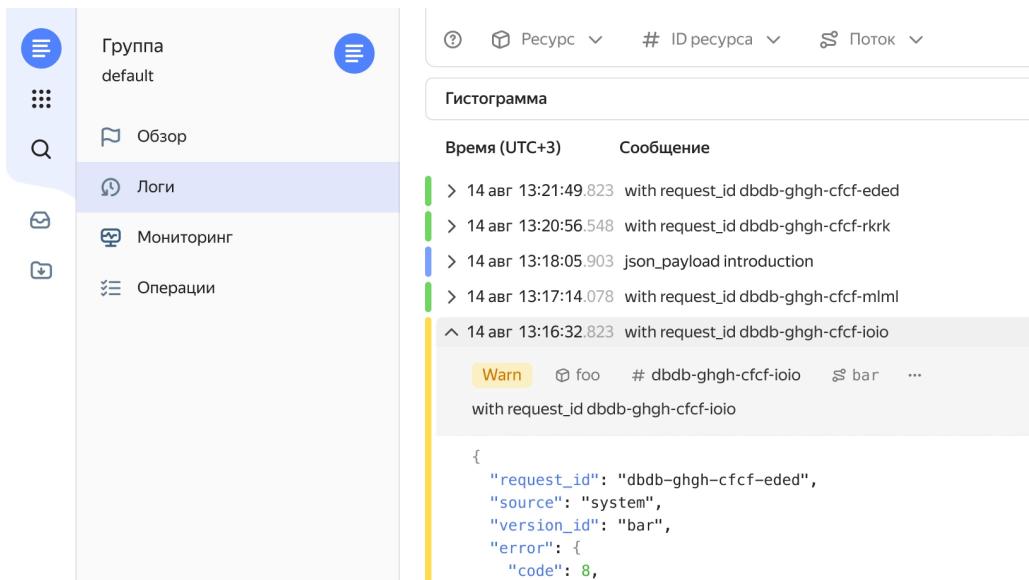


Рис. 22. Пример логов в Yandex Cloud Logging

3.4.2 Мониторинг

Для мониторинга используется Yandex Monitoring, позволяет собирать, хранить и отображать метрики. По дизайну и функционалу близок к Grafana и Prometheus. Позволяет:

- Автоматически собирать метрики от всех ресурсов Yandex Cloud
- Создавать свои графики и дашборды
- Есть уже настроенные дашборды для внутренних ресурсов Yandex Cloud
- API для выгрузки данных
- Создание своих метрик

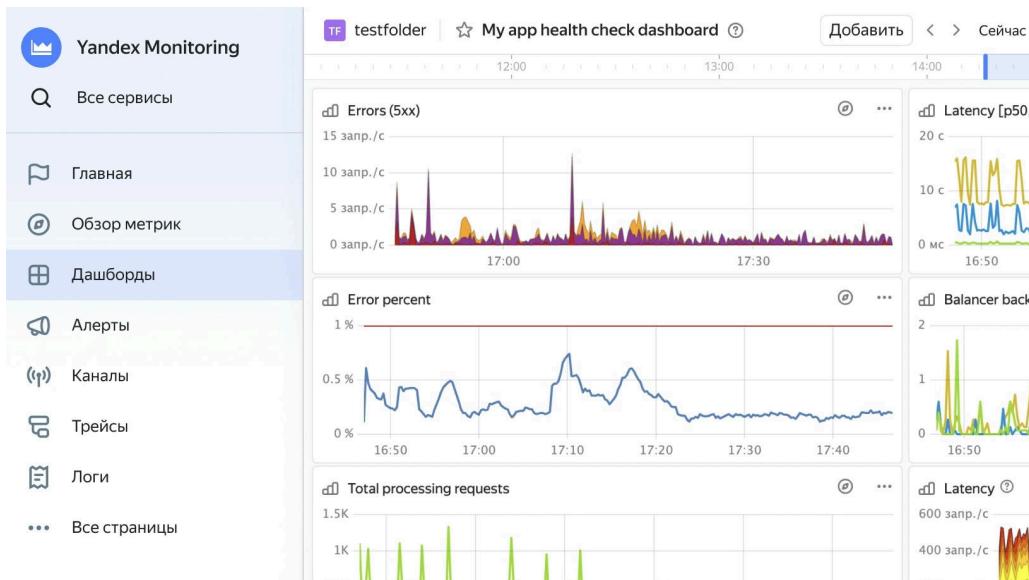


Рис. 23. Пример работы Yandex Monitoring

3.4.3 Алертинг

Сервис мониторинга позволяет настраивать уведомления на достижение пороговых значений метрик. Уведомления могут приходить через различные каналы: email, Telegram, SMS. Алертинг позволяет максимально быстро реагировать на нестандартное поведение системы, своевременно исправлять ошибки и узнавать о неожиданных ситуациях.

На Рис. 24 показан пример уведомления от Yandex Monitoring о превышении порога числа запросов в секунду на сервис API Gateway. Превышение было достигнуто во время проведения нагружочного тестирования. Уведомление приходит в момент превышение порога, а также, после нормализации. В отчёте о нормализации, можно увидеть график числа запросов, время когда он превысил установленный порог и вернулся в норму.



Рис. 24. Пример уведомления от Yandex Monitoring о превышении порога числа запросов в секунду

3.5 Вывод

В данной главе были рассмотрены основные аспекты реализации сервиса, в качестве системы управления версиями выбран GitHub, определён стек технологий для клиентской части – Vue 3 + TypeScript + Vite, для серверной части – Bun + TypeScript + Hono.

Рассмотрены некоторые детали реализации основного функционала проекта – веб-версии карты, описан процесс батчинга геометрии для оптимизации GPU, рассмотрен процесс запекания аннотаций, а так же их система пружинных анимаций, описан механизм достижения энергопотребления с помощью динамической частоты кадров. Описан хостинг статичных файлов в Yandex Object Storage, с использованием CDN для оптимизации раздачи файлов и система инвалидации.

Рассмотрен процесс разработки микросервисов с использованием разработанного шаблона. Описана система мониторинга, включающая в себя логирование, мониторинг и алертинг.

Глава 4. Тестирование и оценка качества кода

4.1 Непрерывная интеграция и развёртывание (CI/CD)

При разработке по микросервисной архитектуре крайне важно на самых ранних этапах автоматизировать процесс сборки и развёртывания приложения. Это обусловлено тем, что при таком подходе появляется множество проектов с частыми обновлениями, автоматизация рутинных процессов позволяет снизить количество ошибок, которые могут возникнуть при ручном развёртывании. А так же позволяет добавить автоматизированные тесты для контроля качества кода с самых ранних этапов.

4.1.1 GitHub Actions

В выбранной системе контроля версий GitHub, для автоматизации процессов сборки и развёртывания, используется встроенный инструмент GitHub Actions. С его помощью можно настроить различные автоматические процессы, которые будут запускаться при определённых действиях в репозитории. Процессы могут состоять из нескольких последовательных и параллельных шагов.

Процессы запускаются на определённом GitHub Runner'е, который может быть как облачным, так и локальным. В моём случае используются облачные GitHub Runner'ы, в качестве операционной системы используется Ubuntu 22.04. GitHub Actions позволяет запускать процессы на Windows и MacOS, однако в этом нет необходимости, весь мой стек технологий прекрасно работает на Linux.

Кроме того, GitHub Actions обладают большим количеством готовых шагов, которые можно найти в GitHub Marketplace и использовать в своих процессах автоматизации.

Процессы состоят из задач (`jobs`), каждая задача состоит из шагов (`steps`). Шаги запускаются последовательно друг за другом, задачи параллельно, однако, задачам можно указать зависимости, и следующая задача будет запускаться только после завершения всех зависимостей.

4.1.2 Infrastructure as Code (IAC)

Для автоматизации развёртывания приложений в CloudNative среде можно использовать IAC (Infrastructure as Code) подход. Он позволяет декларативно описать инфраструктуру в виде кода, который можно хранить рядом с кодом приложения в системе контроля версий. Это позволяет версионировать инфраструктуру вместе с кодом приложения, что в свою очередь позволяет откатить инфраструктуру к предыдущей версии, если в ней были внесены ошибки. Так же такой подход сильно упрощает развёртывание для Serverless приложений, который состоит из множества ресурсов, которые необходимо связать между собой.

4.1.3 Terraform

Для реализации IAC подхода наиболее популярным инструментом является Terraform. Все облачные провайдеры в первую очередь добавляют поддержку именно этого инструмента, в том числе и Yandex Cloud который используется в моём случае.

В Terraform инфраструктура описывается кодом на специальном языке HCL (HashiCorp Configuration Language). Базовой сущностью в Terraform является ресурс, который описывает отдельный компонент инфраструктуры (Api GateWay, Serverless Container, Docker Registry).

Внутри блока ресурса описываются его параметры. Внутри одного Terraform проекта можно ссылаться на другие ресурсы. Например, ресурс Serverless Container должен в своих параметрах ссылаться на ресурс Docker Registry, в котором хранится образ контейнера.

```
1 resource "yandex_container_registry" "registry" {
2   folder_id = var.folder_id
3 }
4
5 resource "docker_image" "main" {
6   name = "cr.yandex/${yandex_container_registry.registry.id}/main:latest"
7   build { context = abspath(local.app_path) }
8   triggers = { hash = local.project_files_hash
9 }
10
11 resource "docker_registry_image" "registry" { name = docker_image.main.name }
12
13 resource "yandex_serverless_container" "container" {
14   name          = local.project_name
15   folder_id     = var.folder_id
16   memory        = 128
17   cores         = 1
18   core_fraction = 5
19   concurrency   = 16
20
21   image {
22     url = docker_registry_image.registry.name
23     digest = docker_registry_image.registry.sha256_digest
24     environment = merge(var.container_env, & "LOG_VARIANT" = "JSON" )
25   }
26
27   depends_on = [...]
28 }
29
30 resource "yandex_api_gateway" "gateway" {
31   name      = local.project_name
32   folder_id = var.folder_id
33
34   spec = templatefile("${path.module}/gateway.yaml.tftpl", {
35     container_id = yandex_serverless_container.container.id
36   })
37 }
```

Листинг 3. Пример использования Terraform

В примере выше (Листинг 3) описано пять ресурсов, которых полностью достаточно для микросервиса в Serverless:

1. `yandex_container_registry` – описывает реестр Docker образов в Yandex Cloud, в котором будут храниться образы контейнеров
2. `docker_image` – описывает образ Docker контейнера, который будет скомпилирован из `local.app_path` директории при изменении хеша от файлов проекта и будет загружен в вышеописанный реестр с тегом `latest`
3. `docker_registry_image` – отвечает за выгрузку образа в реестр (выгрузка определяется по имени образа, который зависит от `id` реестра)
4. `yandex_serverless_container` – описывает Serverless контейнер, к которому будет привязан `latest` образ из реестра `yandex_container_registry`, задаёт ему настройки памяти, количеству ядер, и лимиту параллельных запросов (в данном случае, контейнер не сможет обрабатывать больше 16 запросов одновременно, после выхода за пределы, будут создаваться новые экземпляры). А так же, задаются переменные окружения, которые берутся из параметров запуска Terraform, и добавляется дополнительная переменная `LOG VARIANT`, которая отвечает за формат логов.
5. `yandex_api_gateway` – описывает точку входа в микросервис по спецификации OpenAPI, которая находится в отдельном `gateway.yaml.tftpl` файле, он имеет формат `yaml` с поддержкой шаблонизации Terraform. В качестве значений шаблона, в спецификацию передаётся `container_id`.

4.1.4 Преимущества Serverless в CI/CD

Совместно с использованием Terraform, Serverless подход обладает важным преимуществом – возможностью **бесплатно** создавать сколько угодно тестовых окружений, которые будут полностью идентичны производственным, но будут полностью изолированы от внешней инфраструктуры. Это позволяет запускать тестирования на самых ранних этапах разработки, и проводить тесты на инфраструктурной среде неотличимой от производственной. Это позволяет существенно снизить количество ошибок, которые возникают из-за разницы в тестовом и производственном окружении. Например, при Kubernetes развёртывании, тестовое окружение обычно запускают на одном локальном сервере, из-за чего, при тестировании не учитываются сетевые задержки между нодами, которые неизбежно возникнут при распределённой системе.

В Yandex Cloud для разделения окружение используются **директории** – это способ изолировать ресурсы друг от друга внутри одного облака. Ресурсы находятся в разных подсетях, к ним разные права доступа, разные IAM роли, и нет возможность взаимодействовать между ними.

Однако, на момент написания работы, процесс создания и удаления директорий занимает продолжительное время (около 5 минут), что существенно замедляло CI/CD процесс. Чтобы избежать этого ожидания, был использован классический подход – создание Worker Pool'ов – это группы директорий, которые создаются один раз, а потом переиспользуются по надобности. Для реализации этого подхода, был разработан и опубликован в GitHub Marketplace специальный GitHub Action `yandex-cloud-worker-folder-manager`.

TODO: Лучше использовать <https://typst.app/universe/package/chronos/>

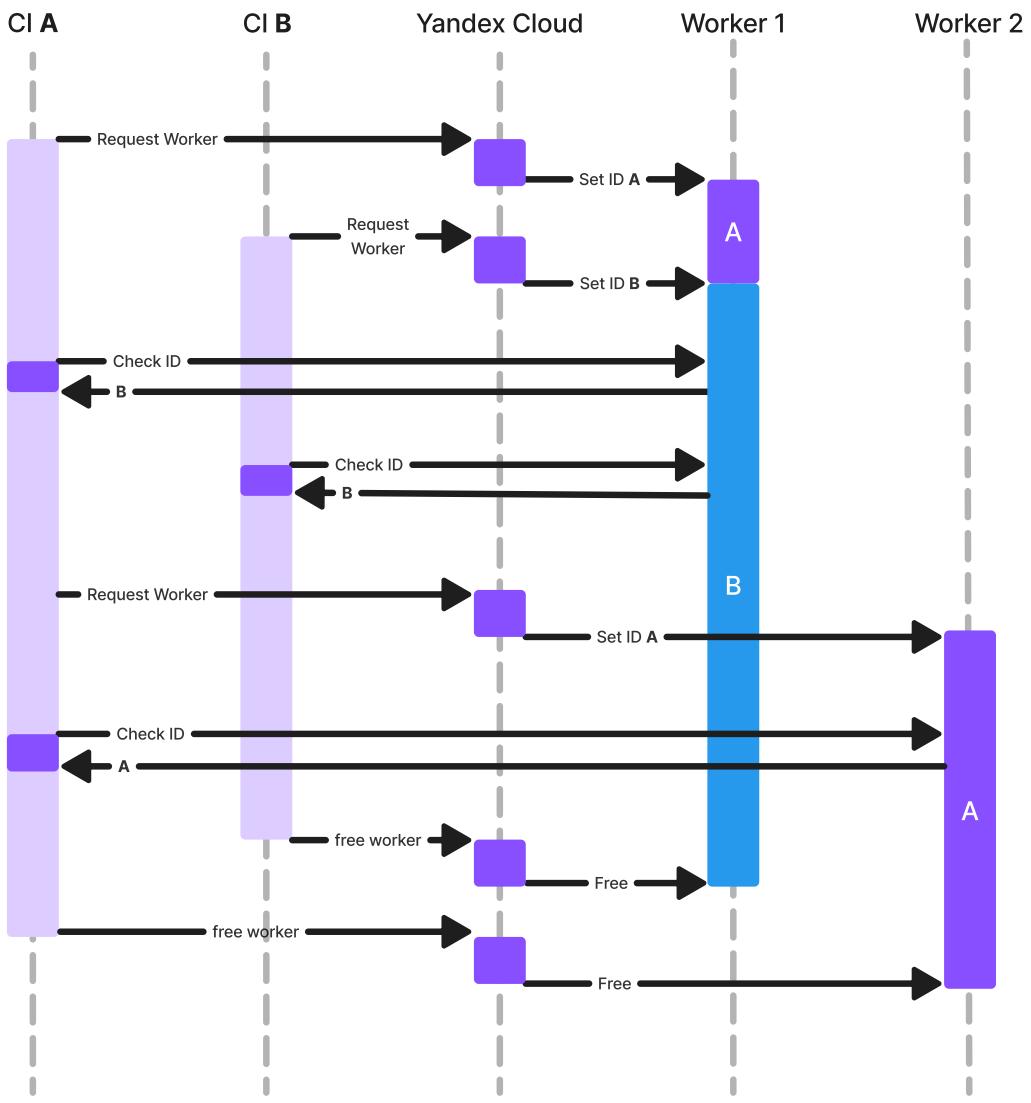


Рис. 25. Диаграмма последовательности для запроса рабочих директорий

Рассмотрим механизм его работы в ситуации, когда два микросервиса одновременно запрашивают новое тестовое окружение и попадают в ситуацию гонки:

1. В начале CI/CD процесса, запускается Action, который через API запрашивает в Yandex Cloud список свободных директорий (директории созданные в рамках пула определяются по наличию метки `worker-directory`, а свободные по отсутствию `worker-used-by-*`)
 1.а. Если свободных директорий нет, то Action создаёт новую директорию, это занимает около 5 минут, после чего, помечает её меткой `worker-directory`
2. Выбирается случайная свободная директория и ей устанавливается метка `worker-used-by-<UUID>`, где `<UUID>` – это уникальный идентификатор сгенерированный в начале CI/CD процесса на основе репозитория, коммита и места откуда он был запущен.
3. Идёт процесс ожидания в 10 секунд (этого времени точно достаточно для установки тега), после чего, запрашивается директория с нужной меткой
 3.а. Если такой директории нет, значит другой процесс одновременно с нашим установил свою метку (ситуация гонки), и нужно повторить процесс с самого начала
4. Внутренний ID директории сохраняется в `output` переменной Action'a, которая будет использоваться в следующих шагах CI/CD процесса
5. После завершения CI/CD процесса, Action удаляет метку `worker-used-by-*` из директории, что позволяет использовать её в других процессах, есть возможность отключить эту опцию

и освобождать директорию не автоматически. Например, если директория запрашивается в момент создания PullRequest'a, то:

- на всё время пока PR открыт, за ним будет закреплена определённая директория
- все GitHub Actions, которые будут запускаться в рамках этого PR, будут использовать эту директорию
- на закрытие PR, нужно сделать отдельный процесс, который будет запускать Action с параметром освобождения директории

Благодаря использованию Serverless подхода, создание таких рабочих директорий в принципе не требует никаких финансовых затрат, так как весь функционал CI/CD будет генерировать минимальную нагрузку, которая не будет превышать бесплатные квоты.

```
1  on:  
2    pull_request:  
3      types: [opened]  
4  
5  jobs:  
6    create-pr-stand:  
7      runs-on: ubuntu-latest  
8      steps:  
9        - name: allocate folder  
10       id: folder  
11       uses: soprachevak/yandex-cloud-worker-folder@v1  
12       with:  
13         operation: get  
14         cloudId: ${{ secrets.YC_CLOUD_ID }}  
15         serviceAccountKeyJson: ${{ secrets.YC_SERVICE_ACCOUNT_KEY_FILE }}  
16  
17        - name: deploy  
18          run: echo "FOLDER_ID=${{ steps.folder.outputs.folderId }}"
```

Листинг 4. Пример запроса рабочей директории в момент открытия PR

```
1  on:  
2    pull_request:  
3      types: [closed]  
4  
5  jobs:  
6    delete-pr-stand:  
7      runs-on: ubuntu-latest  
8      steps:  
9        - name: free folder  
10       id: folder  
11       uses: soprachevak/yandex-cloud-worker-folder@v1  
12       with:  
13         operation: free  
14         cloudId: ${{ secrets.YC_CLOUD_ID }}
```

15

```
serviceAccountKeyJson: ${{ secrets.YC_SERVICE_ACCOUNT_KEY_FILE }}
```

Листинг 5. Пример освобождения рабочей директории в момент закрытия PR

4.1.5 Итоговый CI/CD Pipeline в GitHub Actions

В результате для работы каждого микросервиса было создано несколько CI/CD процессов, которые запускаются на различных этапах:

- На открытие и обновление PullRequest'a – запускает тесты, линтеры, проверяет код на соответствие стандартам, собирает и разворачивает временное тестовое окружение
- На закрытие PullRequest'a – удаляет временное тестовое окружение
- На push в `main` ветку – публикует новую версию в производственном окружении
- На push в `dev` ветку – публикует новую версию в тестовом окружении

При этом процессы на push в `main` и `dev` ветки являются одним и тем же процессом, который запускается с разными параметрами. Это позволяет гарантировать эквивалентность работы микросервиса в производственном и тестовом окружениях.

4.1.5.1 Процесс на открытие PullRequest'a

Запускается на открытие, переоткрытое и обновление PullRequest'a. Самый важный и комплексный процесс.

```
1  on:
2    pull_request:
3      types: [opened, synchronize, reopened]
4
5  jobs:
6    clear_plan_comments: ...
7    pr_plan: ...
8    unit: ...
9    clear_deploy_comments: ...
10   get_folder: ...
11   terraform: ...
12   comment: ...
13   e2e: ...
```

YAML

Листинг 6. Общая структура CI процесса для PR

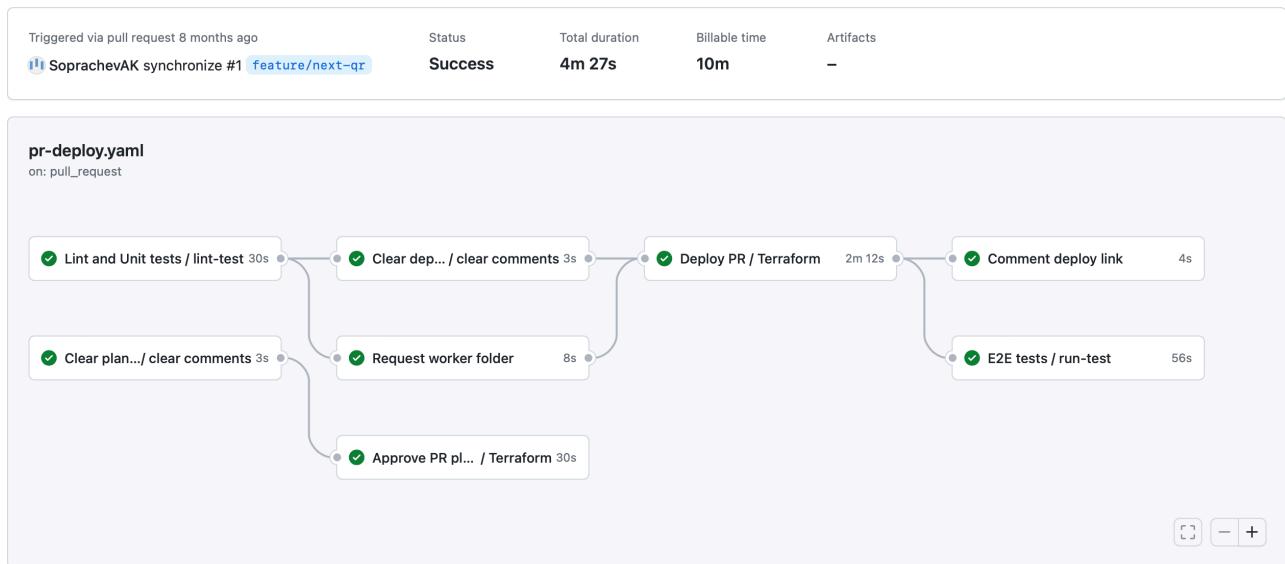


Рис. 26. Диаграмма выполнения CI процесса для PR

Как видно на диаграмме (Рис. 26), процесс состоит из двух независимых частей:

1. Процесс проверки и публикации:
 - 1.a. `Lint and Unit tests` – запускает все виды статических проверок, которые можно выполнить на исходном коде. В частности, запускаются линтер, который оценивает качество кода и проверяет его на соответствие стандартам, а так же запускаются модульные тесты
 - 1.b. На этом этапе параллельно запускается две задачи:
 - `Clear comments` – удаляет старый комментарий с отчётом из текущего PR
 - `Get folder` – запрашивает рабочую директорию в Yandex Cloud
 - 1.v. `Terraform` – применяет `Terraform` инфраструктуру в изолированную рабочую директорию из прошлого шага.
 - 1.g. На этом этапе параллельно запускается две задачи:
 - `Comment deploy link` – добавляет в PR комментарий с сгенерированным URL на API Gateway рабочей директории (это внутренний служебный поддомен Yandex Cloud, который выдаётся всем API Gateway)
 - `E2E tests` – запускает E2E тесты, которые запускаются на реальной инфраструктуре в Yandex Cloud. Проверяют корректность работы API, верность ответов, и время ответа. На вход получают реальный URL на API Gateway, который был сгенерирован в предыдущем шаге.
2. Комментарий в PR
 - 2.a. `Clear comments` – удаляет старый комментарий с отчётом из текущего PR
 - 2.b. `Terraform` – вычисляет `Terraform` план, который будет применён в случае принятия PullRequest'a и выводит его в комментарий к PR. Это позволяет проверяющему PR увидеть, какие изменения будут применены к инфраструктуре в случае принятия PR.

В результате выполнения процесса, получается отчёт о прохождении всех тестов (Рис. 27), в котором описаны проблемы в качестве кода (ESLint), ошибки в модульных тестах (JEST) и результаты E2E тестов (Cypress). А так же, в комментарии к PR добавляется план `Terraform`, который будет применён в случае принятия PR и ссылка на API Gateway рабочей директории, с помощью которой, можно вручную протестировать изменения (Рис. 28)

Lint and Unit tests / lint-test summary ...

ESLint

Error	Warning	Total
0	2	2

▼ Warnings

- <https://github.com/umap-space/qr-generator/blob/99fff48aff0d708191899ab3f355fc023a64ab4c/app/src/utils/gennext/generator.ts>
 - 108:17 'isDark' is assigned a value but never used. @typescript-eslint/no-unused-vars
- <https://github.com/umap-space/qr-generator/blob/99fff48aff0d708191899ab3f355fc023a64ab4c/app/tests/qrgenerator/qrgen.test.ts>
 - 1:10 'generate' is defined but never used. @typescript-eslint/no-unused-vars

JEST

Result	Passed	Failed	Duration
Passing	6	0	3.408 s

▼ Tests

test	passed
QR Code > classifyRect	
QR Code > determinateLineSegments	
QR Code > generatePath	
QR Code > image must be square	
QR Code > svg image must have height and width	
sum moduleA > empty	

Job summary generated at run-time

E2E tests / run-test summary ...

Cypress Results

Result	Passed	Failed	Pending	Skipped	Duration
Passing	28	0	0	0	13.432s

Job summary generated at run-time

Рис. 27. Отчёт о прохождении тестов

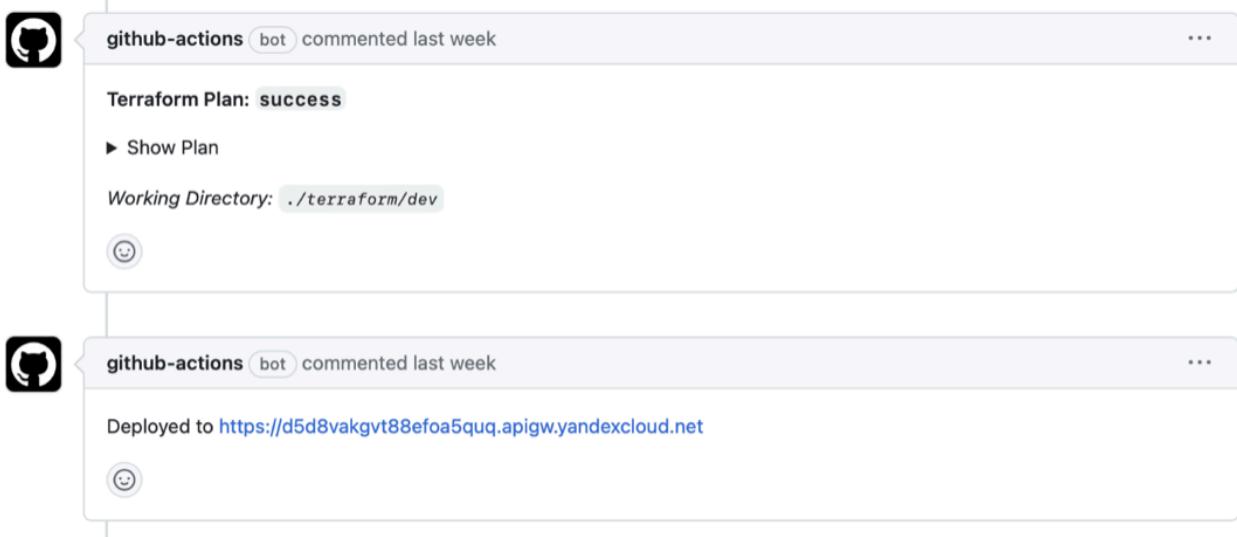


Рис. 28. Комментарий в PR после завершения CI/CD процесса

4.1.5.2 Процесс на применение изменений

Этот процесс занимается применением уже принятых PullRequest'ов в производственное или тестовое окружение. Он запускается на push в main или dev ветку. Процесс упрощён так как не требует резервирования директории. И состоит всего из трёх шагов:

1. Запуск оценки качества кода и Unit тестов. Несмотря на то, что эти процессы уже были пройдены переде принятием PullRequest'a, их повторный запуск позволяет точно гарантировать отсутствие человеческого фактора (например принять PR без учёта CI проверок). Сам процесс крайне быстрый и практически не влияет на общее время выполнения.
2. Применение Terraform плана к целевому окружению.
3. Запуск E2E тестов на целевом окружении, который позволит **гарантировать**, что уже применённые изменения не повлияли на работоспособность сервиса.

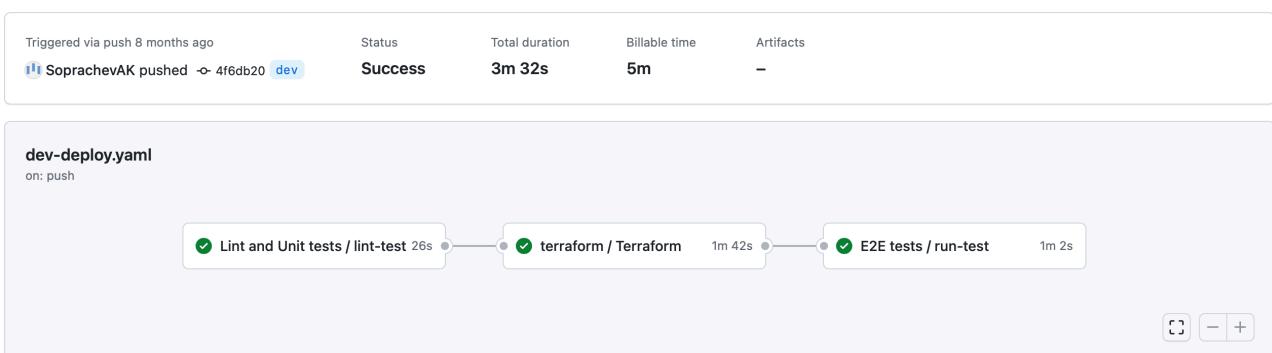


Рис. 29. Диаграмма выполнения CI процесса на применение изменений

4.2 Оценка качества кода

Оценка качества кода – это не менее важный этап разработки, чем тестирование, однако, многие разработчики пренебрегают им, из-за чего, в последствие код становится сложным в поддержке, усложняется его понимание, а так же, увеличивается количество ошибок, которые можно было бы избежать на этапе статического анализа.

В целом, оценка качества кода это комплексный процесс, и не все этапы могут быть автоматизированы, однако выделяют следующие объективные метрики:

- Читаемость (readability)

- ▶ Ясные и осмысленные имена переменных, функций, классов
- ▶ Чёткая структура и форматирование.
- ▶ Отсутствие дублирования кода.
- Поддерживаемость (maintainability)
 - ▶ Разделение ответственности
 - ▶ Модульность архитектуры
 - ▶ Чёткая и единообразная структура кода/стиля
- Тестируемость (testability)
 - ▶ Наличие модульных, интеграционный, e2e и UI тестов
 - ▶ Удобство тестирования (использование DI и интерфейсов)
- Производительность (performance)
 - ▶ Использования эффективных алгоритмов и структур данных
- Надёжность (reliability)
 - ▶ Обработка исключений
 - ▶ Проверка граничных условий
 - ▶ Проверка входных данных (от пользователей)
- Безопасность (security)
 - ▶ Защита от известных уязвимостей (SQL инъекции, XSS, CSRF)
 - ▶ Разделение прав доступа

4.2.1 Применение инструментов оценки качества кода в PolyMap

В контексте данной работы, сервис PolyMap исполняет практически все требования к качеству кода. Часть требований выполняется автоматически в следствие выбранной архитектуры, а в частности:

- в следствие использования микросервисов выполняется разделение ответственности, модульность архитектуры
- в следствие использования CloudNative подхода, выполняется разделение прав доступа (каждый микросервис имеет свой сервисный аккаунт, который имеет доступ только к тем ресурсам, которые необходимы для работы микросервиса)
- использования современных фреймворков и библиотек, реализует защиту от известных уязвимостей
 - ▶ параметризация в клиенте для базы данных, защита от SQL инъекций
 - ▶ Hono использует JWT токены для аутентификации и авторизации, что позволяет избежать CSRF атак
 - ▶ использование VueJS позволяет избежать XSS атак, так как он экранирует все пользовательские данные (настроен на запрет использования v-html директивы, которая позволяет вставлять HTML код в шаблон)
- использование TypeScript позволяет избежать ошибок в коде, связанных с неправильным использованием типов данных, в режиме strict проверяются все типы данных, и предупреждает об ошибках использования типов на этапе компиляции.

Для выполнения остальных требований, требуется использовать специализированные инструменты, в случае с PolyMap, это:

- ESLint – статический анализатор кода для JavaScript и TypeScript, позволяет проверять код на соответствие стандартам, на наличие ошибок и потенциальных проблем. В PolyMap во всех проектах используется ESLint с настройками:
 - ▶ @typescript-eslint/recommended – проверять код на соответствие стандартам TypeScript, а так же, на наличие ошибок (названия переменных, стилистика кода, неиспользуемые переменные, и т.д.).

- eslint-plugin-import – позволяет проверять корректность импортов в проекте.
- eslint-plugin-prettier – позволяет проверять код на соответствие стандартам форматирования кода.

ESLint запускается в CI/CD процессе на каждом этапе, и при наличии ошибок, процесс останавливается, и PullRequest не может быть принят. Это позволяет гарантировать отсутствие ошибок в коде. Кроме того, при правильной настройке IDE, ESLint работает в реальном времени и отображает ошибки и предупреждения прямо во время написания кода.

- Модульное тестирование – используется во всем микросервисах, запускается в CI/CD процессе на каждом этапе, и в случае ошибок, аналогично не позволяет принять PR. В качестве инструмента используется встроенный инструмент BunJS.
- Проверка входных данных в API – используется в каждом микросервисе как middleware для фреймворка Hono. Используется zod-validator, который позволяет проверять входные данные на соответствие заданной схеме и строго типизировать проверенные данные, что на уровне TypeScript гарантирует использование только корректных данных.

```

1 const schema = z.object({
2     foo: z.string().length(5),
3     bar: z.number()
4 })
5
6 api.post('/example',
7     zValidator('json', schema),
8     async (c) => {
9         const valid = c.req.valid('json')
10        console.log(valid.foo) // string with length 5
11        console.log(valid.bar) // number
12    })

```

ts

Листинг 7. Пример использования zod-validator

4.3 Модульное тестирование

Модульное тестирование важный этап разработки, он позволяет проверить отдельные части кода на корректность работы. В отличие от интеграционного и e2e тестирования, модульное тестирование проверяет только отдельные функции и классы, без учёта их взаимодействия с другими сервисами. Это позволяет проверить код на соответствие спецификации, однако не гарантирует его корректную работу в реальной среде. Важным преимуществом является скорость выполнения тестов, для них не требуется развёртывать инфраструктуру, и в контексте TypeScript, нет необходимости компилировать приложение. В PolyMap все микросервисы используют модульное тестирование с помощью встроенного в Bun инструмента bun:test. Это быстрый и мощный инструмент, который позволяет запускать TypeScript и JavaScript тесты, по синтаксису совместим с самым популярным тестовым инструментом jest, но работает быстрее и имеет дополнительные возможности. Bun Test поддерживает классические тесты, обладает Lifecycle hooks (beforeAll, afterAll, beforeEach, afterEach), поддерживает асинхронные тесты, Snapshot testing, и Mocking.

```

1 import { expect, test } from "bun:test";
2
3 test("2 + 2", () => {

```

ts

```
4   expect(2 + 2).toBe(4);
5 };
```

Листинг 8. Пример модульного теста Bun test

Для совместимости с тестами, код должен быть написан с учётом тестиования, а именно использовать Dependency Injection (DI) и интерфейсы, это позволит легко подменять зависимости в тестах используя Mock объекты вместо реальных зависимостей, благодаря чему, тесты лучше изолированы друг от друга.

```
1 import { test, expect, mock } from "bun:test";
2 const random = mock(() => Math.random());
3
4 test("random", async () => {
5   const val = random();
6   expect(val).toBeGreaterThan(0);
7   expect(random).toHaveBeenCalled();
8   expect(random).toHaveBeenCalledTimes(1);
9 });
```

ts

Листинг 9. Пример использования Mock в Bun test

Кроме того, Bun test поддерживает Mocking встроенных модулей и прм пакетов, путём подмены модулей в `preload` режиме, благодаря чему, обычный `import` модуля получит Mock версию, а не реальную.

```
1 import { test, expect, mock } from "bun:test";
2
3 mock.module("./module", () => {
4   return {
5     foo: "bar",
6   };
7 });
8
9 test("mock.module", async () => {
10   const esm = await import("./module");
11   expect(esm.foo).toBe("bar");
12
13   const cjs = require("./module");
14   expect(cjs.foo).toBe("bar");
15});
```

ts

Листинг 10. Пример использования Mock для подмены модуля Bun test

4.4 UI тестирование

Тестирование пользовательского интерфейса не так популярно как модульное или интеграционное, это связано с тем, что UI тесты писать достаточно сложно и выполняются они зачастую медленно, особенно, если технология UI не поддерживает такой вид тестов.

В тестировании UI выделяют три подхода:

1. E2E тестирование (End-to-End testing) – это тестирование всего приложения, которое проверяет его работоспособность с точки зрения пользователя. Инструмент запускает приложение, нажимает на кнопки, и проверяет, что на экране присутствуют нужные элементы и тексты. Однако, такой подход не учитывает вёрстку и визуальные составляющие (запускает тесты с отключённым css).
2. Тестирование на уровне компонентов (Component testing) – это тестирование отдельных компонентов пользовательского интерфейса, в случае, если используется компонентный UI фреймворк. В случае с PolyMap, это VueJS, который является компоненты и поддерживает компонентное тестирования.
3. Тестирование визуального интерфейса (Visual testing) – это тестирование наиболее приближенное к реальному пользовательскому опыту, оно проходит по сценарием использования, и на каждом шаге делает скриншот приложения, который сравнивается с эталонным. В случае, если скриншот отличается от эталонного, то тест считается проваленным. При хорошем покрытии, этот подход позволяет гарантировать, что внесённые изменения повлияли только на ожидаемую часть интерфейса. Например, частой проблемой является внесение изменений в компьютерную версию сайта, которые ломают мобильную версию, однако, программист вносящий изменения, мог проверять функционал только на компьютерной версии. Визуальное тестирование позволит избежать таких ошибок.

В случае с PolyMap может быть применено оба подхода, однако, было принято решение использовать только Visual testing, так как этот подход позволяет точно определить наличие нежелательного результата при применение изменений, однако существенно сокращает время на написание тестов. На начальных этапах разработки, компонентное тестирование требует значительное время на написание тестов, при этом, при каждом изменение тесты придётся переписывать, что замедлит разработку. В случае с визуальным тестированием, достаточно сделать несколько тестов основных пользовательских сценариев, и такие тесты представляют из себя обычный скриншот приложения.

4.4.1 Инструменты визуального тестирования

В текущий момент, на рынке существует три основных инструмента для UI тестирования:

- Selenium
- Cypress
- Playwright

Рассмотрим каждый из них подробнее с точки зрения пригодности для PolyMap, в частности нас интересует поддержка веб тестирования, компонентного для VueJS, возможность запуска тестов в Github CI/CD.

4.4.1.1 Selenium

Один из самых старых инструментов для браузерного тестирования, поддерживает множество языков программирования, включая Java, Python, C# и в том числе TypeScript, который нас интересует. Обладает обширной экосистемой, большим числом плагинов, хорошей документацией и самым большим сообществом. Хорошо интегрируется в CI/CD процессы. Позволяет запускать тесты во всех основных браузерах, и даже в устаревшем Internet Explorer. Однако, Selenium сам по себе не был предназначен для компонентного тестирования и не поддерживает его. Кроме того, из-за особенностей движка WebDriver, на котором он основан, тесты работают медленно.

```
1 import { Builder, By, until } from 'selenium-webdriver';
```

ts

```

2 import { strict as assert } from 'assert';
3
4 (async () => {
5     const driver = await new Builder().forBrowser('chrome').build();
6     try {
7         await driver.get(URL);
8         const incBtn = await driver.findElement(By.css('[data-
9         testid="increment"]'));
10        await incBtn.click();
11        const countEl = await driver.findElement(By.css('[data-testid="count"]'));
12        await driver.wait(until.elementTextIs(countEl, '1'), 2000);
13        assert.equal(await countEl.getText(), '1');
14    } finally {
15        await driver.quit();
16    }
17 })();

```

Листинг 11. Пример теста на Selenium

4.4.1.2 Cypress

Современный инструмент для браузерного тестирования, был разработан специально для компонентного тестирования React/Vue/Angular приложений, однако не ограничивается им. Кроме тестирования интерфейсов поддерживает и другие виды E2E тестирования, например тестирование REST API. В контексте Vue, Cypress позволяет монтировать компоненты в изолированном окружении без необходимости устанавливать сторонние плагины. Основными преимуществами Cypress выделяют простоту написания тестов и скорость их выполнения. Тесты выполняются в реальном браузере, том же, что и используется приложением, что позволяет значительно сократить время выполнения тестов. Cypress хорошо поддерживает интеграцию в CI/CD и позволяет записывать видео и делать скриншоты для провалившихся тестов, что позволяет оценить что именно пошло не так. Так же, Cypress поддерживает `time-travel` режим, который позволяет записывать DOM состояние на каждом шаге выполнения теста, и подробно изучать, что привело к ошибке, что крайне полезно для HTML приложений. Визуальное тестирование в Cypress достигается с помощью плагина `cypress-image-snapshot`, который позволяет находить разницу между двумя изображениями и определять изменившуюся область. Обладает обширной документацией и графической средой отладки тестов, что упрощает процесс написания тестов. Поддерживает современные браузеры. Инструмент с открытым исходным кодом.

```

1 // E2E
2 describe('Counter page', () => {
3     it('increments on click', () => {
4         cy.visit('/');
5         cy.get('[data-testid=count]').should('have.text', '0');
6         cy.get('[data-testid=increment]').click();
7         cy.get('[data-testid=count]').should('have.text', '1');
8     });

```

ts

```

9  });
10
11 // Component
12 import Counter from '@/components/Counter.vue';
13 describe('Counter component', () => {
14   it('increments on click', () => {
15     cy.mount(Counter); // mount из @cypress/vue
16     cy.get('[data-testid=count]').should('have.text', '0');
17     cy.get('[data-testid=increment]').click();
18     cy.get('[data-testid=count]').should('have.text', '1');
19   });
20 });

```

Листинг 12. Пример теста на Cypress

4.4.1.3 Playwright

Самый новый инструмент из представленных, разработан Microsoft, имеет полностью открытый исходный код. Поддерживает все современные браузеры. Инструмент является многоязычным и поддерживает тесты на Python, Java, C#, однако основной акцент делается на JavaScript/TypeScript. Playwright не зависит от фреймворка и хорошо поддерживает E2E тестирование интерфейсов, но компонентное тестирование, на данный момент, является экспериментальной функцией (@playwright/experimental-ct-vue), которая не гарантирует стабильность. Самый быстрый инструмент из представленных, поддерживает многопоточность. В отличие от Cypress, Playwright работает отдельно от браузера через механизмы DevTools, однако всё ещё остаётся быстрым за счёт многопоточности. Лучше всех альтернатив поддерживает визуальное тестирование, все необходимые механизмы для этого уже встроены в инструмент. Так же, Playwright предоставляет графическое приложение, для отслеживания выполнения тестов и записи трасс выполнения. Хорошо работает в CI/CD процессах. Основным недостатком является малое сообщество разработчиков, несмотря на хорошую документацию, далеко не все проблемы могут быть решены с её помощью, стоит учитывать, что инструмент новый и будет развиваться.

```

1 import { test, expect } from '@playwright/test'; ts
2
3 // E2E + Visual testing
4 test('increments counter', async ({ page }) => {
5   await page.goto('/');
6   await expect(page.locator('[data-testid=count]')).toHaveText('0');
7   await page.click('[data-testid=increment]');
8   await expect(page.locator('[data-testid=count]')).toHaveText('1');
9
10 // Visual testing
11 await expect(page).toHaveScreenshot();
12 });
13
14 // Component (Experimental)
15 import { test, expect } from '@playwright/experimental-ct-vue';

```

```

16 import Counter from '@/components/Counter.vue';
17 test('component increments', async ({ mount }) => {
18   const component = await mount(Counter);
19   await expect(component.locator('[data-testid=count]')).toHaveText('0');
20   await component.locator('[data-testid=increment]').click();
21   await expect(component.locator('[data-testid=count]')).toHaveText('1');
22 });

```

Листинг 13. Пример теста на Playwright

4.4.1.4 Выводы

Для проекта PolyMap, в качестве инструмента визуального тестирования был выбран Cypress за счёт компромисса между стабильностью и простотой написания тестов. В будущем, возможен переход на Playwright, который в техническом плане почти не уступает, однако на данный момент он ещё недостаточно стабилен, не поддерживает компонентное тестирование, и не обладает достаточным сообществом разработчиков.

Cypress был интегрирован в CI/CD процесс и запускается на каждый PR, отчёт о прохождение тестов выводится в результат Action'a, в случае проваленных тестов, прикладывается короткий видеоролик с выполнением неудачных тестов. Кроме того, Cypress используется и в некоторых микросервисах для тестирования визуального ответа, например в микросервисе qr-generator, который стилизует QR коды, Cypress позволяет проверить, что QR код отображается таким, каким он и ожидался (Visual testing). Пример отчёта о прохождении тестов (Рис. 27, раздел Cypress Results)

4.5 Интеграционное и End to End тестирование

Интеграционное тестирование является важным этапом разработки, оно позволяет проверить взаимодействие между различными компонентами системы, в свою очередь, E2E тестирование проверяет работу всей системы в целом на примере определённых сценариев использования. Несмотря на то, что типы тестирования проверяют похожие задачи, и проверка взаимодействия между компонентами может быть выполнена в рамках E2E тестов, эти два вида тестирования принято разделять, это связано с проблемами производительности. Для запуска E2E тестов требуется развёртывание всей инфраструктуры, что при классическом подходе требует затрат как времени так и ресурсов. Интеграционное тестирование же, может быть выполнено локально, что быстрее и дешевле, однако это накладывает на программиста ряд дополнительных обязанностей.

Для написании интеграционных тестов, приходится использовать Mock объекты инфраструктуры, то есть, например, вместе реально базы данных, использовать in-memory аналог, реализующий необходимый контракт. Написание и поддерживание таких Mock объектов требует значительных временных затрат разработчика.

Благодаря использованию Serverless подхода и IAC подхода к управлению инфраструктурой, в PolyMap удалось избежать написания интеграционных тестов. На каждый PR создаётся реальная инфраструктура в Yandex Cloud, которая в свою очередь, является полностью изолированной. На этой инфраструктуре можно запускать все необходимые тесты, используя не Mock объекты, а реальные компоненты Yandex Cloud. Благодаря Serverless такие запуски происходят быстро и не требует затрат ресурсов. Единственный вид интеграций, который всё ещё требует создания Mock объектов – это интеграции с другими микросервисами PolyMap, так как инфраструктура является изолированной, тестируемый микросервис не может взаимодействовать с ней.

ствовать с другими реальными микросервисами. Ещё одним преимуществом такого подхода, является возможность оценить не только факт работоспособности интеграции, а и затраченное время на выполнение, которое тоже проверяется в тестах (фактическое время выполнения не должно превышать ожидаемое), что было бы невозможно на локальной Mock инфраструктуре.

Таким образом, и Интеграционное и E2E тестирование в PolyMap выполняется с помощью одного инструмента – Cypress, который проводит как интеграционные и контрактные тесты, проверяя внешний REST API, так и полноценные E2E тесты по пользовательским сценариям. Входным параметром для Cypress передаётся служебный URL для API Gateway в инфраструктуре Yandex Cloud, таким образом, тесты запускаются не локально, а ходят от GitHub Runner'ов до Yandex Cloud, имитируя реальные действия пользователей. Отчёт о прохождении тестов (Рис. 27, раздел Cypress Results).

```
1 const MAX_RESPONSE_MS = 400; ts
2
3 describe('QR styling service', () => {
4   ('[png', 'svg']).forEach((type) => {
5     it(`Style for ${type}`, () => {
6       cy.request({
7         method: 'GET',
8         url: `/qr.${type}`,
9         qs: params
10      }).then((res) => {
11        expect(res.status).to.eq(200)
12        expect(res.headers['content-type']).to.include(mime(type))
13        expect(res.duration).to.be.lessThan(MAX_RESPONSE_MS)
14
15        if (type === 'png') expect(res.body.length).to.be.greaterThan(100)
16        else expect(res.body).to.match(/^<svg[^>]*>/)
17      })
18    })
19  })
20})
```

Листинг 14. Пример E2E теста на Cypress

Кроме этого, в организации GitHub присутствует отдельный репозиторий с E2E тестами всей системы PolyMap, который запускается вручную или по CRON таймеру раз в сутки и проверяет сценарии использования всей системы. Эти тесты запускаются на производственном окружении, и выполняет роль обычного пользователя, убеждаясь, что все основные сценарии использования работают корректно. В случае ошибок, создаётся issue в репозитории с описанием проблемы и ссылкой на видео с выполнением теста. Этот процесс не является частью какого либо CI/CD процесса и не привязан к обновлениям или изменениям в коде. Проблемы могут случаться в любое время даже без вмешательства разработчиков, например, в случае изменения API стороннего сервиса, или ошибки со стороны Yandex Cloud. Эти E2E тесты позволяют гарантировать, что в текущий момент, вся система работает так, как это и задумывалось.

4.6 Нагрузочное тестирование

Нагрузочное тестирование – это тестирование системы на предмет её производительности и устойчивости к нагрузке. В PolyMap используется бесконечно масштабируемый Serverless подход, и нагрузочное тестирование не имеет смысла, так как система будет масштабировать под любую нагрузку, и время ответа будет оставаться близким к константе. Однако, использование инструмента нагрузочного тестирования, позволяет убедиться в этом, а так же, проверить, как работает мониторинг и алerts в случае резкого увеличения нагрузки, подробнее об этом рассмотрено в разделе 3.4.

4.7 Вывод

В данной главе были рассмотрены механизмы тестирования и оценки качества кода в сервисе PolyMap. Были обоснованы выборы инструментов и походов к тестированию. Рассмотрены способы интеграции тестов и оценки качества кода в CI/CD процессы. В качестве статического анализатора кода был выбран ESLint, модульное тестирование выполняется с помощью встроенного инструмента Bun test, E2E и UI тестирование выполняется с помощью Cypress. Интеграционное тестирование выполняется в рамках E2E тестов, что стало возможным, благодаря использованию Serverless подхода и IAC подхода к управлению инфраструктурой. Нагрузочное тестирование мало применимо к бесконечно масштабируемому Serverless подходу, однако, в PolyMap с его помощью проверялись корректность работы метрик и алертов.

Заключение

Список литературы

ПРИЛОЖЕНИЕ А.

Отчёт СППР о выборе облачного провайдера

A.1. Исходные данные

	Вес	Yandex	Selectel	VK	Sber	AWS	Google	Azure
↑ Вычисления	2	9	6	5	7	10	9	9
↑ Интеграции	1	9	6	5	8	10	9	8
↑ БД, Очереди, Уведомления	0.8	8	7	6	8	10	9	8
↑ DevOps & DX	0.8	8	6	6	7	10	9	9
↑ Мониторинг, логи	0.9	8	6	6	8	9	9	8

Таблица 1. Исходные данные

A.1.1. Нормализованные веса

Для корректной работы алгоритма веса критериев были нормализованы.

- Вычисления - 0.36
- Интеграции - 0.18
- БД, Очереди, Уведомления - 0.15
- DevOps & DX - 0.15
- Мониторинг, логи - 0.16

A.2. Ход решения

Для определения оптимального варианта были построены матрицы бинарных отношений (БО), после чего к ним были применены следующие механизмы:

- Механизмы доминирования
- Механизмы блокирования
- Турнирный механизм
- Механизм K-такс

По каждому механизму был выбран лучший вариант, после чего была составлена сводная таблица с результатами

A.2.1. Механизм доминирования

Сколько раз варианты доминируют по всем БО. В матричном виде выбираются варианты, у которых в строках все значения равны 1.

- Yandex не блокирует ни в одной категории => 0
- Selectel не блокирует ни в одной категории => 0
- VK не блокирует ни в одной категории => 0
- Sber не блокирует ни в одной категории => 0
- AWS доминируют в категориях: Вычисления, Интеграции, БД, Очереди, Уведомления, DevOps & DX, Мониторинг, логи => $0.36 + 0.18 + 0.15 + 0.15 + 0.16 = 1$
- Google доминируют в категориях: Мониторинг, логи => 0.16
- Azure не блокирует ни в одной категории => 0

Вариант	Баллы	Место
Yandex	0	3
Selectel	0	3

Вариант	Баллы	Место
VK	0	3
Sber	0	3
AWS	1	1
Google	0.16	2
Azure	0	3

Таблица 2. Сводная таблица результатов механизма доминирования

A.2.2. Механизм блокирования

Сколько раз варианты блокируют по всем БО. В матричном виде выбираются варианты, у которых в столбцах все значения равны 1.

- **Yandex** не блокирует ни в одной категории => **0**
- **Selectel** не блокирует ни в одной категории => **0**
- **VK** не блокирует ни в одной категории => **0**
- **Sber** не блокирует ни в одной категории => **0**
- **AWS** блокируют в категориях: **Вычисления, Интеграции, БД, Очереди, Уведомления, DevOps & DX** => $0.36 + 0.18 + 0.15 + 0.15 = 0.84$
- **Google** не блокирует ни в одной категории => **0**
- **Azure** не блокирует ни в одной категории => **0**

Вариант	Баллы	Место
Yandex	0	2
Selectel	0	2
VK	0	2
Sber	0	2
AWS	0.84	1
Google	0	2
Azure	0	2

Таблица 3. Сводная таблица результатов механизма блокирования

A.2.3. Турнирный механизм

Сколько раз варианты предпочтительнее

- **Yandex** - в категории:
 - **Вычисления** – опережает **Selectel, VK, Sber** => $0.36 + 0.36 + 0.36 = 1.09$
Симметрично с **Google, Azure** => $0.36 / 3 = 0.12$
 - **Интеграции** – опережает **Selectel, VK, Sber, Azure** => $0.18 + 0.18 + 0.18 + 0.18 = 0.73$
Симметрично с **Google** => $0.18 / 2 = 0.09$
 - **БД, Очереди, Уведомления** – опережает **Selectel, VK** => $0.15 + 0.15 = 0.29$
Симметрично с **Sber, Azure** => $0.15 / 3 = 0.05$
 - **DevOps & DX** – опережает **Selectel, VK, Sber** => $0.15 + 0.15 + 0.15 = 0.44$
 - **Мониторинг, логи** – опережает **Selectel, VK** => $0.16 + 0.16 = 0.33$
Симметрично с **Sber, Azure** => $0.16 / 3 = 0.05$
- **Selectel** - в категории:
 - **Вычисления** – опережает **VK** => **0.36**
 - **Интеграции** – опережает **VK** => **0.18**

- ▶ **БД, Очереди, Уведомления** – опережает **VK** => **0.15**
- ▶ **DevOps & DX** – симметрично с **VK** => **0.15 / 2 = 0.07**
- ▶ **Мониторинг, логи** – симметрично с **VK** => **0.16 / 2 = 0.08**
- **VK** - в категории:
 - ▶ **DevOps & DX** – симметрично с **Selectel** => **0.15 / 2 = 0.07**
 - ▶ **Мониторинг, логи** – симметрично с **Selectel** => **0.16 / 2 = 0.08**
- **Sber** - в категории:
 - ▶ **Вычисления** – опережает **Selectel, VK** => **0.36 + 0.36 = 0.73**
 - ▶ **Интеграции** – опережает **Selectel, VK** => **0.18 + 0.18 = 0.36**
Симметрично с **Azure** => **0.18 / 2 = 0.09**
 - ▶ **БД, Очереди, Уведомления** – опережает **Selectel, VK** => **0.15 + 0.15 = 0.29**
Симметрично с **Yandex, Azure** => **0.15 / 3 = 0.05**
 - ▶ **DevOps & DX** – опережает **Selectel, VK** => **0.15 + 0.15 = 0.29**
 - ▶ **Мониторинг, логи** – опережает **Selectel, VK** => **0.16 + 0.16 = 0.33**
Симметрично с **Yandex, Azure** => **0.16 / 3 = 0.05**
- **AWS** - в категории:
 - ▶ **Вычисления** – опережает **Yandex, Selectel, VK, Sber, Google, Azure** => **0.36 + 0.36 + 0.36 + 0.36 + 0.36 = 2.18**
 - ▶ **Интеграции** – опережает **Yandex, Selectel, VK, Sber, Google, Azure** => **0.18 + 0.18 + 0.18 + 0.18 + 0.18 = 1.09**
 - ▶ **БД, Очереди, Уведомления** – опережает **Yandex, Selectel, VK, Sber, Google, Azure** => **0.15 + 0.15 + 0.15 + 0.15 + 0.15 = 0.87**
 - ▶ **DevOps & DX** – опережает **Yandex, Selectel, VK, Sber, Google, Azure** => **0.15 + 0.15 + 0.15 + 0.15 = 0.87**
 - ▶ **Мониторинг, логи** – опережает **Yandex, Selectel, VK, Sber, Azure** => **0.16 + 0.16 + 0.16 + 0.16 = 0.82**
Симметрично с **Google** => **0.16 / 2 = 0.08**
- **Google** - в категории:
 - ▶ **Вычисления** – опережает **Selectel, VK, Sber** => **0.36 + 0.36 + 0.36 = 1.09**
Симметрично с **Yandex, Azure** => **0.36 / 3 = 0.12**
 - ▶ **Интеграции** – опережает **Selectel, VK, Sber, Azure** => **0.18 + 0.18 + 0.18 + 0.18 = 0.73**
Симметрично с **Yandex** => **0.18 / 2 = 0.09**
 - ▶ **БД, Очереди, Уведомления** – опережает **Yandex, Selectel, VK, Sber, Azure** => **0.15 + 0.15 + 0.15 + 0.15 = 0.73**
 - ▶ **DevOps & DX** – опережает **Yandex, Selectel, VK, Sber** => **0.15 + 0.15 + 0.15 + 0.15 = 0.58**
Симметрично с **Azure** => **0.15 / 2 = 0.07**
 - ▶ **Мониторинг, логи** – опережает **Yandex, Selectel, VK, Sber, Azure** => **0.16 + 0.16 + 0.16 + 0.16 = 0.82**
Симметрично с **AWS** => **0.16 / 2 = 0.08**
- **Azure** - в категории:
 - ▶ **Вычисления** – опережает **Selectel, VK, Sber** => **0.36 + 0.36 + 0.36 = 1.09**
Симметрично с **Yandex, Google** => **0.36 / 3 = 0.12**
 - ▶ **Интеграции** – опережает **Selectel, VK** => **0.18 + 0.18 = 0.36**
Симметрично с **Sber** => **0.18 / 2 = 0.09**
 - ▶ **БД, Очереди, Уведомления** – опережает **Selectel, VK** => **0.15 + 0.15 = 0.29**
Симметрично с **Yandex, Sber** => **0.15 / 3 = 0.05**

- **DevOps & DX** – опережает **Yandex, Selectel, VK, Sber** => $0.15 + 0.15 + 0.15 + 0.15 = 0.58$
Симметрично с **Google** => $0.15 / 2 = 0.07$
- **Мониторинг, логи** – опережает **Selectel, VK** => $0.16 + 0.16 = 0.33$
Симметрично с **Yandex, Sber** => $0.16 / 3 = 0.05$

Вариант	Баллы	Место
Yandex	3.19	3
Selectel	0.85	6
VK	0.15	7
Sber	2.19	5
AWS	5.92	1
Google	4.31	2
Azure	3.04	4

Таблица 4. Сводная таблица результатов турнирного механизма

A.2.4. Механизм K-max

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Yandex	5	3	5	3	$16 * 0.36 = 5.82$	-
Selectel	1	1	1	1	$4 * 0.36 = 1.45$	-
VK	0	0	0	0	0	-
Sber	2	2	2	2	$8 * 0.36 = 2.91$	-
AWS	6	6	6	6	$24 * 0.36 = 8.73$	8.73 (Строго наиб.)
Google	5	3	5	3	$16 * 0.36 = 5.82$	-
Azure	5	3	5	3	$16 * 0.36 = 5.82$	-

Таблица 5. Таблица результатов механизма K-max для категории Вычисления

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Yandex	5	4	5	4	$18 * 0.18 = 3.27$	-
Selectel	1	1	1	1	$4 * 0.18 = 0.73$	-
VK	0	0	0	0	0	-
Sber	3	2	3	2	$10 * 0.18 = 1.82$	-
AWS	6	6	6	6	$24 * 0.18 = 4.36$	4.36 (Строго наиб.)
Google	5	4	5	4	$18 * 0.18 = 3.27$	-
Azure	3	2	3	2	$10 * 0.18 = 1.82$	-

Таблица 6. Таблица результатов механизма K-max для категории Интеграции

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Yandex	4	2	4	2	$12 * 0.15 = 1.75$	-
Selectel	1	1	1	1	$4 * 0.15 = 0.58$	-
VK	0	0	0	0	0	-
Sber	4	2	4	2	$12 * 0.15 = 1.75$	-
AWS	6	6	6	6	$24 * 0.15 = 3.49$	3.49 (Строго наиб.)
Google	5	5	5	5	$20 * 0.15 = 2.91$	-
Azure	4	2	4	2	$12 * 0.15 = 1.75$	-

Таблица 7. Таблица результатов механизма K-так для категории БД, Очереди, Уведомления

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Yandex	3	3	3	3	$12 * 0.15 = 1.75$	-
Selectel	1	0	1	0	$2 * 0.15 = 0.29$	-
VK	1	0	1	0	$2 * 0.15 = 0.29$	-
Sber	2	2	2	2	$8 * 0.15 = 1.16$	-
AWS	6	6	6	6	$24 * 0.15 = 3.49$	3.49 (Строго наиб.)
Google	5	4	5	4	$18 * 0.15 = 2.62$	-
Azure	5	4	5	4	$18 * 0.15 = 2.62$	-

Таблица 8. Таблица результатов механизма K-так для категории DevOps & DX

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Yandex	4	2	4	2	$12 * 0.16 = 1.96$	-
Selectel	1	0	1	0	$2 * 0.16 = 0.33$	-
VK	1	0	1	0	$2 * 0.16 = 0.33$	-
Sber	4	2	4	2	$12 * 0.16 = 1.96$	-
AWS	6	5	6	5	$22 * 0.16 = 3.6$	-
Google	6	5	6	5	$22 * 0.16 = 3.6$	-
Azure	4	2	4	2	$12 * 0.16 = 1.96$	-

Таблица 9. Таблица результатов механизма K-так для категории Мониторинг, логи

Вариант	Сумма sJp	Место sJp	Сумма sJm	Место sJm
Yandex	14.55	3	0	2
Selectel	3.38	6	0	2
VK	0.62	7	0	2
Sber	9.6	5	0	2
AWS	23.67	1	20.07	1

Вариант	Сумма sJp	Место sJp	Сумма sJm	Место sJm
Google	18.22	2	0	2
Azure	13.96	4	0	2

Таблица 10. Сводная таблица результатов механизма K-max

A.3. Результат

По результатам всех механизмов, в зависимости от полученного места были начислены баллы каждому варианту

Вариант	Дом	Блок	Тур	Sjp	Sjm	ИТОГО	Место
Yandex	5	6	5	5	6	27	3
Selectel	5	6	2	2	6	21	6
VK	5	6	1	1	6	19	7
Sber	5	6	3	3	6	23	5
AWS	7	7	7	7	7	35	1
Google	6	6	6	6	6	30	2
Azure	5	6	4	4	6	25	4

Таблица 11. Итоговая таблица результатов

A.3.1. Итоговый вариант

Максимальную сумму баллов набрал вариант AWS с суммой **35** баллов

ПРИЛОЖЕНИЕ Б.

Отчёт СППР о выборе фреймворка для веб-карты

Б.1. Исходные данные

	Вес	Vue3	React	Angular	Svelte
↑ Производительность	2	9	8	7	10
↑ Размер бандла	0.7	9	8	6	10
↑ TypeScript	2	9	9	10	5
↑ Встроенные возможности	1	10	5	8	6
↑ Реактивность (model)	2	10	7	6	7
↑ Минимальные зависимости	0.5	8	7	6	10
↑ Стабильность API	1.5	8	10	9	6
↑ Инструменты разработки	1	10	10	8	7
↑ Документация, сообщество	1	8	10	6	4

Таблица 1. Исходные данные

Б.1.1. Нормализованные веса

Для корректной работы алгоритма веса критериев были нормализованы.

- **Производительность** - 0.17
- **Размер бандла** - 0.06
- **TypeScript** - 0.17
- **Встроенные возможности** - 0.09
- **Реактивность (model)** - 0.17
- **Минимальные зависимости** - 0.04
- **Стабильность API** - 0.13
- **Инструменты разработки** - 0.09
- **Документация, сообщество** - 0.09

Б.2. Ход решения

Для определения оптимального варианта были построены матрицы бинарных отношений (БО), после чего к ним были применены следующие механизмы:

- Механизмы доминирования
- Механизмы блокирования
- Турнирный механизм
- Механизм K-max

По каждому механизму был выбран лучший вариант, после чего была составлена сводная таблица с результатами

Б.2.1. Механизм доминирования

Сколько раз варианты доминируют по всем БО. В матричном виде выбираются варианты, у которых в строках все значения равны 1.

- Vue3 доминируют в категориях: **Встроенные возможности, Реактивность (model), Инструменты разработки => $0.09 + 0.17 + 0.09 = 0.34$**
- React доминируют в категориях: **Стабильность API, Инструменты разработки, Документация, сообщество => $0.13 + 0.09 + 0.09 = 0.3$**

- **Angular** доминируют в категориях: **TypeScript => 0.17**
- **Svelte** доминируют в категориях: **Производительность, Размер бандла, Минимальные зависимости => 0.17 + 0.06 + 0.04 = 0.27**

Вариант	Баллы	Место
Vue3	0.34	1
React	0.3	2
Angular	0.17	4
Svelte	0.27	3

Таблица 2. Сводная таблица результатов механизма доминирования

Б.2.2. Механизм блокирования

Сколько раз варианты блокируют по всем БО. В матричном виде выбираются варианты, у которых в столбцах все значения равны 1.

- **Vue3** блокируют в категориях: **Встроенные возможности, Реактивность (model) => 0.09 + 0.17 = 0.26**
- **React** блокируют в категориях: **Стабильность API, Документация, сообщество => 0.13 + 0.09 = 0.21**
- **Angular** блокируют в категориях: **TypeScript => 0.17**
- **Svelte** блокируют в категориях: **Производительность, Размер бандла, Минимальные зависимости => 0.17 + 0.06 + 0.04 = 0.27**

Вариант	Баллы	Место
Vue3	0.26	2
React	0.21	3
Angular	0.17	4
Svelte	0.27	1

Таблица 3. Сводная таблица результатов механизма блокирования

Б.2.3. Турнирный механизм

Сколько раз варианты предпочтительнее

- **Vue3** - в категории:
 - ▶ **Производительность** – опережает **React, Angular => 0.17 + 0.17 = 0.34**
 - ▶ **Размер бандла** – опережает **React, Angular => 0.06 + 0.06 = 0.12**
 - ▶ **TypeScript** – опережает **Svelte => 0.17**
Симметрично с **React => 0.17 / 2 = 0.09**
 - ▶ **Встроенные возможности** – опережает **React, Angular, Svelte => 0.09 + 0.09 + 0.09 = 0.26**
 - ▶ **Реактивность (model)** – опережает **React, Angular, Svelte => 0.17 + 0.17 + 0.17 = 0.51**
 - ▶ **Минимальные зависимости** – опережает **React, Angular => 0.04 + 0.04 = 0.09**
 - ▶ **Стабильность API** – опережает **Svelte => 0.13**
 - ▶ **Инструменты разработки** – опережает **Angular, Svelte => 0.09 + 0.09 = 0.17**
Симметрично с **React => 0.09 / 2 = 0.04**
 - ▶ **Документация, сообщество** – опережает **Angular, Svelte => 0.09 + 0.09 = 0.17**
- **React** - в категории:
 - ▶ **Производительность** – опережает **Angular => 0.17**
 - ▶ **Размер бандла** – опережает **Angular => 0.06**

- ▶ **TypeScript** – опережает **Svelte => 0.17**
Симметрично с **Vue3 => 0.17 / 2 = 0.09**
- ▶ **Реактивность (model)** – опережает **Angular => 0.17**
Симметрично с **Svelte => 0.17 / 2 = 0.09**
- ▶ **Минимальные зависимости** – опережает **Angular => 0.04**
- ▶ **Стабильность API** – опережает **Vue3, Angular, Svelte => 0.13 + 0.13 + 0.13 = 0.38**
- ▶ **Инструменты разработки** – опережает **Angular, Svelte => 0.09 + 0.09 = 0.17**
Симметрично с **Vue3 => 0.09 / 2 = 0.04**
- ▶ **Документация, сообщество** – опережает **Vue3, Angular, Svelte => 0.09 + 0.09 + 0.09 = 0.26**
- **Angular** - в категории:
 - ▶ **TypeScript** – опережает **Vue3, React, Svelte => 0.17 + 0.17 + 0.17 = 0.51**
 - ▶ **Встроенные возможности** – опережает **React, Svelte => 0.09 + 0.09 = 0.17**
 - ▶ **Стабильность API** – опережает **Vue3, Svelte => 0.13 + 0.13 = 0.26**
 - ▶ **Инструменты разработки** – опережает **Svelte => 0.09**
 - ▶ **Документация, сообщество** – опережает **Svelte => 0.09**
- **Svelte** - в категории:
 - ▶ **Производительность** – опережает **Vue3, React, Angular => 0.17 + 0.17 + 0.17 = 0.51**
 - ▶ **Размер бандла** – опережает **Vue3, React, Angular => 0.06 + 0.06 + 0.06 = 0.18**
 - ▶ **Встроенные возможности** – опережает **React => 0.09**
 - ▶ **Реактивность (model)** – опережает **Angular => 0.17**
Симметрично с **React => 0.17 / 2 = 0.09**
 - ▶ **Минимальные зависимости** – опережает **Vue3, React, Angular => 0.04 + 0.04 + 0.04 = 0.13**

Вариант	Баллы	Место
Vue3	2.09	1
React	1.64	2
Angular	1.11	4
Svelte	1.16	3

Таблица 4. Сводная таблица результатов турнирного механизма

B.2.4. Механизм K-max

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Vue3	2	2	2	2	$8 * 0.17 = 1.37$	-
React	1	1	1	1	$4 * 0.17 = 0.68$	-
Angular	0	0	0	0	0	-
Svelte	3	3	3	3	$12 * 0.17 = 2.05$	2.05 (Строго наиб.)

Таблица 5. Таблица результатов механизма K-max для категории Производительность

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Vue3	2	2	2	2	$8 * 0.06 = 0.48$	-

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
React	1	1	1	1	$4 * 0.06 = 0.24$	-
Angular	0	0	0	0	0	-
Svelte	3	3	3	3	$12 * 0.06 = 0.72$	0.72 (Строго наиб.)

Таблица 6. Таблица результатов механизма K-max для категории Размер бандла

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Vue3	2	1	2	1	$6 * 0.17 = 1.03$	-
React	2	1	2	1	$6 * 0.17 = 1.03$	-
Angular	3	3	3	3	$12 * 0.17 = 2.05$	2.05 (Строго наиб.)
Svelte	0	0	0	0	0	-

Таблица 7. Таблица результатов механизма K-max для категории TypeScript

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Vue3	3	3	3	3	$12 * 0.09 = 1.03$	1.03 (Строго наиб.)
React	0	0	0	0	0	-
Angular	2	2	2	2	$8 * 0.09 = 0.68$	-
Svelte	1	1	1	1	$4 * 0.09 = 0.34$	-

Таблица 8. Таблица результатов механизма K-max для категории Встроенные возможности

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Vue3	3	3	3	3	$12 * 0.17 = 2.05$	2.05 (Строго наиб.)
React	2	1	2	1	$6 * 0.17 = 1.03$	-
Angular	0	0	0	0	0	-
Svelte	2	1	2	1	$6 * 0.17 = 1.03$	-

Таблица 9. Таблица результатов механизма K-max для категории Реактивность (model)

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Vue3	2	2	2	2	$8 * 0.04 = 0.34$	-
React	1	1	1	1	$4 * 0.04 = 0.17$	-
Angular	0	0	0	0	0	-
Svelte	3	3	3	3	$12 * 0.04 = 0.51$	0.51 (Строго наиб.)

Таблица 10. Таблица результатов механизма K-max для категории Минимальные зависимости

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Vue3	1	1	1	1	$4 * 0.13 = 0.51$	-
React	3	3	3	3	$12 * 0.13 = 1.54$	1.54 (Строго наиб.)
Angular	2	2	2	2	$8 * 0.13 = 1.03$	-
Svelte	0	0	0	0	0	-

Таблица 11. Таблица результатов механизма К-макс для категории Стабильность API

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Vue3	3	2	3	2	$10 * 0.09 = 0.85$	-
React	3	2	3	2	$10 * 0.09 = 0.85$	-
Angular	1	1	1	1	$4 * 0.09 = 0.34$	-
Svelte	0	0	0	0	0	-

Таблица 12. Таблица результатов механизма К-макс для категории Инструменты разработки

	HRo+ ER+ NR	HRo+ NR	HRo+ ER	HRo	Sjp	Sjm
Vue3	2	2	2	2	$8 * 0.09 = 0.68$	-
React	3	3	3	3	$12 * 0.09 = 1.03$	1.03 (Строго наиб.)
Angular	1	1	1	1	$4 * 0.09 = 0.34$	-
Svelte	0	0	0	0	0	-

Таблица 13. Таблица результатов механизма К-макс для категории Документация, сообщество

Вариант	Сумма sJp	Место sJp	Сумма sJm	Место sJm
Vue3	8.34	1	3.08	2
React	6.56	2	2.56	3
Angular	4.44	4	2.05	4
Svelte	4.65	3	3.28	1

Таблица 14. Сводная таблица результатов механизма К-макс

Б.3. Результат

По результатам всех механизмов, в зависимости от полученного места были начислены баллы каждому варианту

Вариант	Дом	Блок	Тур	Sjp	Sjm	ИТОГО	Место
Vue3	4	3	4	4	3	18	1
React	3	2	3	3	2	13	3
Angular	1	1	1	1	1	5	4
Svelte	2	4	2	2	4	14	2

Таблица 15. Итоговая таблица результатов

Б.3.1. Итоговый вариант

Максимальную сумму баллов набрал вариант **Vue3** с суммой **18** баллов

ПРИЛОЖЕНИЕ В.

Реализация функции пружинной интерполяции

```
1  function springEasing(ts
2    dampingRatio: number,
3    frequencyResponse: number,
4    duration: number = 1,
5    velocity: number = 0) {
6      const spring = new DampedHarmonicSpring(dampingRatio, frequencyResponse)
7      return (progress: number) => 1 - spring.position(progress * duration, 1,
8      velocity)
9    }
10
11
12  class DampedHarmonicSpring {
13    private dampingCoefficient: number
14    private mass: number
15    private stiffness: number
16
17    constructor(dampingRatio: number, frequencyResponse: number) {
18      this.stiffness = Math.pow(2 * Math.PI / frequencyResponse, 2)
19      this.dampingCoefficient = 4 * Math.PI * dampingRatio / frequencyResponse
20    }
21
22    private get dampingRatio(): number {
23      return this.dampingCoefficient / (2 * Math.sqrt(this.stiffness))
24    }
25
26    private get dampedNaturalFrequency(): number {
27      return Math.sqrt(this.stiffness) * Math.sqrt(Math.abs(1 -
28        Math.pow(this.dampingRatio, 2)))
29    }
30
31    position(time: number, initialPosition: number = 1, initialVelocity: number
32      = 0): number {
33      const ζ = this.dampingRatio
34      const λ = this.dampingCoefficient / 2
35      const ω_d = this.dampedNaturalFrequency
36      const s_0 = initialPosition
37      const v_0 = initialVelocity
38      const t = time
39
40      if (Math.abs(ζ - 1) < 1e-6) {
41        const c_1 = s_0
42        const c_2 = v_0 + λ * s_0
43
44        return Math.exp(-λ * t) * (c_1 + c_2 * t)
45      }
46    }
47  }
48}
```

```

41      }
42      else if ( $\zeta < 1$ ) {
43          const c_1 = s_0
44          const c_2 = (v_0 +  $\lambda * s_0$ ) /  $\omega_d$ 
45
46          return Math.exp(- $\lambda * t$ ) * (c_1 * Math.cos( $\omega_d * t$ ) + c_2 * Math.sin( $\omega_d * t$ ))
47      }
48      else {
49          const c_1 = (v_0 + s_0 * ( $\lambda + \omega_d$ )) / (2 *  $\omega_d$ )
50          const c_2 = s_0 - c_1
51
52          return Math.exp(- $\lambda * t$ ) * (c_1 * Math.exp( $\omega_d * t$ ) + c_2 * Math.exp(- $\omega_d * t$ ))
53      }
54  }
55 }
```

Листинг 1. Реализация функции пружинной интерполяции

ПРИЛОЖЕНИЕ Г.

Тестирование времени ответа в Serverless режиме

- Тесты проводились в Yandex Cloud
- Использовался Serverless Containers
- Запросы отправлялись напрямую в Serverless Containers по служебному URL
- Время ответа измерялось с помощью Yandex Cloud Logging – учитывается чистое время на запрос отображаемое в логах, без квантизации
- Проводилось по три теста:
 - Cold_n – время ответа при n холодном запуске. После прошлого запроса пауза 2 минуты
 - Hot_n – время ответа при n горячем запуске. После прошлого запроса пауза не более 10 секунд
- По трём запускам берётся среднее значение

Технология	Cold ₁	Cold ₂	Cold ₃	Hot ₁	Hot ₂	Hot ₃	Cold _{avg}	Hot _{avg}
Express	421	421	421	5	4	5	421.0	4.6
Express + validator	421	419	424	5	5	5	421.1	5.1
Express + validator + moment	721	710	713	5	5	5	714.6	5.2
Express + validator + moment + JSDOM	1392	1398	1399	15	14	15	1396.1	14.2
Express + validator + moment + linkedom	716	715	716	5	6	6	715.6	5.6
Express + linkedom	710	713	730	5	6	5	717.7	5.4
Fastify	704	711	704	4	4	4	706.3	3.9
Fastify + validator	705	705	702	4	4	4	703.7	4.0
Fastify + moment	704	706	702	4	4	4	704.2	4.1
Fastify + validator + moment	706	707	714	4	4	4	708.9	3.9
Fastify + validator + moment + linkedom	712	710	709	5	5	5	710.5	5.0
Express JS	269	415	265	4	4	4	316.2	4.1
elixir + cowboy + plug + phoenix	1280	1277	1279	4	3	4	1278.9	3.5
ESBuild -ext: express + moment + JSDOM	1406	1401	1407	20	20	21	1404.7	20.4
ESBuild: express + moment + -ext: JSDOM	1378	1382	1388	14	13	13	1382.7	13.3
ESBuild -ext: express + moment + linkedom	713	714	720	5	4	5	715.7	4.6
ESBuild: express + moment + linkedom	425	418	424	6	5	5	422.2	5.1

Технология	Cold ₁	Cold ₂	Cold ₃	Hot ₁	Hot ₂	Hot ₃	Cold _{avg}	Hot _{avg}
Parcel -ext: express + moment + linkedom	708	707	712	5	5	6	709.0	5.1
Parcel: express + moment + linkedom	275	270	274	5	7	3	273.0	4.9
Rollup: express + moment + linkedom	422	427	422	6	7	6	423.7	6.2
Rollup: express + moment + linkedom + sharp	429	448	430	15	15	14	435.7	14.5
Rollup: express + moment + linkedom + async sharp	524	516	520	15	14	15	520.0	14.7
Webpack +min - sideEffect: express + moment + linkedom	427	426	434	6	5	5	429.0	5.2
Webpack: express + moment + linkedom	432	435	440	6	6	6	435.7	6.1
Webpack +min + sideEffect: express + moment + linkedom	280	276	278	12	20	18	278.0	16.7
Webpack +min + sideEffect: express + moment + linkedom + sharp	434	431	432	15	15	15	432.3	15.0
Webpack +min + sideEffect: express + linkedom + async sharp	377	378	381	14	15	14	378.7	14.3
Webpack +min + sideEffect: express + async linkedom + async sharp	386	379	386	15	14	14	383.7	14.3
QR	1278	1300	1292	50	41	56	1290.0	49.0
QR/api	1253	1345	1246	54	52	75	1281.3	60.3
QR+WebPack	702	780	735	47	45	52	739.0	48.0
QR+WebPack + sharp - jimp	833	820	851	70	73	68	834.7	70.3
BUN node-slim, EXPRESS sharp + linkedom	934	912	958	18	20	15	934.7	17.7
BUN node-slim, Elysia sharp + linkedom	724	666	707	13	14	12	699.0	13.1
BUN distroless Elysia echo	256	249	260	1	4	3	255.0	2.7
BUN distroless hono echo	187	181	175	4	4	2	181.0	3.0
BUN bundle hono echo	139	189	182	4	4	4	170.0	4.0
BUN bundle executable	199	188	188	4	4	4	191.7	3.9

Технология	Cold ₁	Cold ₂	Cold ₃	Hot ₁	Hot ₂	Hot ₃	Cold _{avg}	Hot _{avg}
NODE Webpack hono echo	214	288	226	4	5	4	242.7	4.3
BUN bundle hono echo	168	179	165	7	6	6	170.7	6.2
BUN TS express echo [node18-slim]	387	391	409	5	5	5	395.7	5.0
BUN build —external express echo [node18-slim]	407	389	384	4.3	4	4	393.3	4.2
BUN build express echo [node18-slim]	384	397	371	5	5	4	384.0	4.9

Таблица 1. Сравнительная таблица времени ответа разных стеков в Serverless режиме