МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

СӘТБАЕВ УНИВЕРСИТЕТІ

Институт информационных и телекоммуникационных технологий

Аргымбек Адильжан Кайратович

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

На соискание академической степени магистра

Название диссертации	Создание микросервиса планирования маршрутов в системе оптимизации логистики
Направление подготовки	6М070400 – Вычислительная техника и программное обеспечение

Научный руководитель
канд. тех. наук, ассис.-проф
_____Р.Юнусов
«__» _____ 2020 г.

Оппонент
Доактор PhD, ассис.-проф
_____ А. Богданчиков
«__» _____ 2020 г.

Нормоконтроль
лектор
_____
Ж.М.Алибиева
«__» _____ 2020 г.

ДОПУЩЕН К ЗАЩИТЕ
Заведующий кафедрой ПИ
Доктор PhD,
_____Тұрдалыұлы М.
«__»_____ 2020 г.

Алматы 2020

Институт кибернетики и информационных технологий

Кафедра Программная инженерия

Специальность: 6М070400 – Вычислительная техника и программное обеспечение

УТВЕРЖДАЮ
Заведующий кафедрой ПИ
Доктор PhD,
_____Тұрдалыұлы М.
«___» _____ 2020 г.

ЗАДАНИЕ
на выполнение магистерской диссертации

магистранту Аргымбек Адильжану

Тема диссертации: «Создание микросервиса планирования маршрутов в системе оптимизации логистики»

Срок сдачи законченной диссертации «___» _____

Исходные данные к магистерской диссертации. Дана монолитная система оптимизации логистики, которая требует внедрение функции планирования маршрутов. Поставленные цели и задачи диссертационной работы направлены на анализ методов проектирования и поиске оптимального решения для построения микросервиса планирования маршрутов

Перечень подлежащих разработке в магистерской диссертации вопросов или краткое содержание магистерской диссертации: а) изучение особенностей монолитного и микросервисного подходов при проектировании масштабируемой веб архитектуры; б) исследование проблема маршрутизации транспортных средств; в) исследование процесса миграции монолитной архитектуры на микросервисы; г) сравнительный анализ технологий и инструментов для построение микросервисной архитектуры; д) построение микросервисной архитектуры на базе системы оптимизации логистики

ГРАФИК

подготовки магистерской диссертации

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю	Примечание
Раздел 1. Подход		
Раздел 2. Основные эксперименты		

Консультации по проекту с указанием относящихся к ним разделов проекта

Раздел	Консультант, (уч. степень, звание)	Сроки	Подпись
Нормоконтроль	Ж.М.Алибиева, Лектор кафедры программная инженерия		

Дата выдачи задания "____"_____ 2020 г.

Заведующий кафедрой _____ Тұрдалыұлы М.

Научный руководитель _____ Р. Юнусов

Задание принял к исполнению магистрант _____ А. Аргымбек

Дата "_____"_____ 2020 г.

АННОТАЦИЯ

Актуальность исследования. В настоящее время многие компании, такие как Netflix, Apple, Instagram и Pinterest перенесли свои приложения и системы на микросервисы, поскольку данная архитектура позволяет этим компаниям масштабировать свои вычислительные ресурсы в соответствии с их использованием. Данная необходимость появилась вследствие того, что современные веб-приложения имеют высокие требования для полноценной работы, такие как возможность предоставление программного интерфейса, обработка большого количества запросов, масштабируемость, обеспечение высокой скорости доступа к данным, обеспечение высокой надежности отказоустойчивости. Перечисленные гиганты столкнулись с различными проблемами, которые были решены благодаря переходу на микросервисы.

Netflix столкнулся со сложностью обработки информации огромного количества клиентов. Когда в компании обнаружили, что его рост опережает возможности традиционной монолитной архитектуры, микросервисы оказались верным решением для масштабируемости. В среднем по вечерам в будни на Netflix приходится почти треть всего интернет-трафика в Северной Америке, а к полуночи падает к минимуму. Микросервисная архитектуры стала спасением от падения сервиса при такой нестабильной нагрузке, позволив развертывать дополнительные сервера в пиковые часы

Будучи одним из самых быстро растущих сайтов в интернете, Pinterest использует микросервисы чтобы приспособиться к различным уровням трафика. При этом сохраняет невероятно малую команду. Их онлайн-сервис специально разработан для агрегации больших объёмов данных, что стало возможным благодаря использование микросервисов.

Instagram обратился к микросервисам для поддержания своего роста и удовлетворения большой потребности в масштабировании. Сервис впервые был запущен в 2010 году как простое монолитное приложение. Как результат, в течение нескольких часов сервер оказался перегружен, а Instagram пришлось развернуть микросервисы. Шесть месяцев спустя компания уже работала с тремя миллионами постоянных пользователей, не имея проблем с трафиком.

Apple обратилась к микросервисной архитектуре, чтобы оставаться на передовой позиции в сфере технологий. Непосредственно это произошло в ходе управления выпуском Siri — программы, имитирующей способность человека слушать и отвечать на вопросы пользователей. Микросервисы позволили Apple расширять и адаптировать Siri гораздо быстрее и без перерывов в обслуживания пользователей. Последние даже не заметили, что в течение долгого времени компания производила обновление версий и внедряла новый функционал.

Данные компании являются хорошим примером того, как гибкость микросервисов можно использовать для внедрения инноваций и оптимизации расходов. Тысячи других брендов по всему миру делают то же самое, открывая новые возможности микросервисной архитектуры.

Объектом исследования является понятие масштабируемой распределенной веб архитектуры, ее типы, принципы проектирования и инфраструктурные требования.

Предметом исследования является процесс миграции устаревшего ПО на микросервисную архитектуру при построении масштабируемого веб-приложения.

Цель и задачи исследования. Цель построить масштабируемое веб-приложения для оптимизации транспортной логистики на базе микросервисов с разделением основного клиентского приложения и системы планирования маршрутов

Поставленная задача предопределила **решение следующих задач**:

- Возможность проводить планирование несколько планирований маршрутов одновременно, без блокировки основного приложения
- Разделить процесса разработки и тестирования для системы планирования и основного приложения
- Способность легко масштабировать систему планирования при необходимости
- Возможность использования инструмента для решения задачи маршрутизации транспортных средств (VRP) на более производительном языке программирования, отличном от основного приложения.

Новизна исследования заключается в том, что сегодня нет какого-то общего решения для построения масштабируемой системы планирования маршрутов, которое полностью отвечало бы всем необходимым требованиям. С учетом большого опыт и экспертизы в данном сфере, было проведено исследования рынка ПО, на наличия такой системы, но явное решение найдено не было.

Научно-практическая значимость. Полученные автором результаты в построении микросервиса планирования маршрута в системе оптимизации транспортной логистики имеют научно-практическую ценность в процессе миграции от устаревшего приложения к масштабируемой распределенной веб архитектуре, которые могут быть использованы на производстве для решения различных транспортных задач.

Содержание диссертации. Диссертация состоит из введения, пяти глав, изложена на 77 страницах, содержит список используемой литературы из 30 наименований, включая 9 таблиц и 14 рисунков.

Содержание

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей диссертации применяют следующие обозначения и сокращения с соответствующими определениями:

Обозначение/сокращение	**Определение**
МА	Монолитная архитектура
MSA	Микросервисная архитектура
API	Application programming interface
REST	Representational state transfer
TLS	Transport layer security
SOA	Сервис-ориентированная архитектура
JS	Java Script
ПО	Программное обеспечение
HTTP	HyperText Transfer Protocol

ВВЕДЕНИЕ

Актуальность исследования. В настоящее время многие компании, такие как Netflix, Apple, Instagram и Pinterest перенесли свои приложения и системы на микросервисы, поскольку данная архитектура позволяет этим компаниям масштабировать свои вычислительные ресурсы в соответствии с их использованием. Данная необходимость появилась вследствие того, что современные веб-приложения имеют высокие требования для полноценной работы, такие как возможность предоставление программного интерфейса, обработка большого количества запросов, масштабируемость, обеспечение высокой скорости доступа к данным, обеспечение высокой надежности отказоустойчивости. Рассмотрим причины и цели данных компаний, которые привели к переходу на микросервисы:

- Netflix столкнулся со сложностью обработки информации огромного количества клиентов. Когда в компании обнаружили, что его рост опережает возможности традиционной монолитной архитектуры, микросервисы оказались верным решением для масштабируемости. В среднем по вечерам в будни на Netflix приходится почти треть всего интернет-трафика в Северной Америке, а к полуночи падает к минимуму. Микросервисная архитектуры стала спасением от падения сервиса при такой нестабильной нагрузке, позволив развертывать дополнительные сервера в пиковые часы.

- Будучи одним из самых быстро растущих сайтов в интернете, Pinterest использует микросервисы для проведения экспериментов, чтобы приспособиться к различным уровням трафика. При этом сохраняет невероятно малую команду. Их онлайн-сервис специально разработан для агрегации больших объёмов данных, что стало возможным благодаря использование микросервисов.

- Instagram обратился к микросервисам для поддержания своего роста и удовлетворения большой потребности в масштабировании. Сервис впервые был запущен в 2010 году как простое монолитное приложение. Как результат, в течение нескольких часов сервер оказался перегружен, а Instagram пришлось развернуть микросервисы. Шесть месяцев спустя компания уже работала с тремя миллионами постоянных пользователей, не имея проблем с трафиком.

- Apple обратилась к микросервисной архитектуре, чтобы оставаться на передовой позиции в сфере технологий. Непосредственно это произошло в ходе управления выпуском Siri — программы, имитирующей способность человека слушать и отвечать на вопросы пользователей. Микросервисы позволили Apple расширять

и адаптировать Siri гораздо быстрее и без перерывов в обслуживания пользователей. Последние даже не заметили, что в течение долгого времени компания производила обновление версий и внедряла новый функционал.

Данные компании являются хорошим примером того, как гибкость микросервисов можно использовать для внедрения инноваций и оптимизации расходов. Тысячи других брендов по всему миру делают то же самое, открывая новые возможности микросервисной архитектуры.

Объектом исследования является понятие масштабируемой распределенной веб архитектуры, ее типы, принципы проектирования и инфраструктурные требования.

Предметом исследования является процесс миграции устаревшего ПО на микросервисную архитектуру при построении масштабируемого веб-приложения.

Цель исследования: построить масштабируемое веб-приложения для оптимизации транспортной логистики на базе микросервисов с разделением основного клиентского приложения и системы планирования маршрутов

Задачи исследования:

- Возможность проводить планирование несколько планирований маршрутов одновременно, без блокировки основного приложения
- Разделить процесса разработки и тестирования для системы планирования и основного приложения
- Способность легко масштабировать систему планирования при необходимости
- Возможность использования инструмента для решения задачи маршрутизации транспортных средств (VRP) на более производительном языке программирования, отличном от основного приложения.

Новизна исследования заключается в том, что сегодня нет какого-то общего решения для построения масштабируемой системы планирования маршрутов, которое полностью отвечало бы всем необходимым требованиям. С учетом большого опыт и экспертизы в данном сфере, было проведено исследования рынка ПО, на наличия такой системы, но явное решение найдено не было.

Методы исследования: в ходе данной работы были применены теоретические и экспериментальные исследования.

Теоретические исследования:

- Анализ
- Сравнительный анализ

- Аналитическое исследования

Экспериментальные исследования:

- Производственные
- Искусственные (поисковые исследования)
- Анкетирование

Практическая значимость исследования заключается в построении микросервиса планирования маршрутов на базе системы оптимизации логистики, которая позволяет проводить планирования нескольких маршрутов одновременно без задержек и блокировки основного клиентского приложения, что дало возможность масштабировать приложение в условиях растущего трафика.

Структура исследования: работа состоит из введения, пяти глав, заключения и списка литературы.

1. Теоретическое изучение особенностей монолитного и микросервисного подходов при проектировании масштабируемой веб архитектуры

1.1. Определение понятия масштабируемой веб архитектуры и ее основные характеристики

Основная часть разработки веб-приложения - это его масштабируемость. Независимо от того, какой проект вы собираетесь запустить, вы должны быть готовы к притоку пользователей и ожидать, что система справится с этим. Необходимо принимать во внимание то, что могут быть случаи, когда ваша система недостаточно гибкая и не может выдерживать большую нагрузку. Чтобы предотвратить это, важно начать масштабирование приложений до того, как начнется этап разработки.

В данной главе речь пойдет о масштабируемом веб-приложении, которое может обрабатывать огромный поток данных без неожиданных сбоев. Начнем с определения масштабируемости веб-приложения и определим некоторые из ключевых принципов их построения.

Масштабируемость - это способность веб-приложения справляться с растущим числом пользователей, одновременно взаимодействующих с приложением. Следовательно, масштабируемое веб-приложение - это приложение, которое одинаково хорошо работает с одним или тысячей пользователей и выдерживает взлеты и падения трафика.

Если говорит о факторах, влияющих на масштабируемость веб-приложения, важно выделить следующее:

- Архитектура веб-приложений и правильно написанный код играют ключевую роль в создании масштабируемых веб-приложений.
- Фреймворк влияет на производительность приложения, необходимо выберите наиболее подходящий для решения бизнес требования с учетом различных факторов.
- Нагрузочное тестирование помогает найти и преодолеть слабые стороны приложения, чтобы гарантировать его бесперебойную работу.
- Сторонние сервисы также должны быть тщательно отобраны, иначе они могут привести к сбою в работе.

Однако, недостаточно просто написать программное обеспечение или запустить сервер. Оба они требуют регулярного обслуживания и управления. Чтобы обеспечить обслуживание как программного обеспечения, так и сервера, мы можем использовать платформы и инструменты, которые облегчают отслеживание приложения, позволяют применять обновления при необходимости и обеспечивают эффективную работу приложения.

В построении системы необходимо учитывать основные принципы масштабируемости. Многие из существующих типов перекрываются и даже иногда конфликтуют друг с другом, но их присутствие имеет решающее значение для процесса проектирования, и они всегда должны сосуществовать в балансе. Поскольку эти типы напрямую связаны с принципами проектирования, мы разберем их подробно ниже:

- Выбирайте горизонтальное масштабирование вместо вертикального. Вы можете обнаружить, что система иногда ограничена горизонтальным масштабированием, тем не менее, эти ограничения регулярно расширяют базы данных в одном направлении. Кроме того, в горизонтальном масштабирование вы можете просто добавить другой сервер, а не обновлять существующий. Это сэкономит вам немного ресурсов.
- Не нагружайте одно ядро. Со временем количество клиентов превысит количество доступных вам услуг. Таким образом, чтобы избежать каких-либо узких мест при масштабировании приложения, распределите как можно больше работы от ядра.
- В первую очередь API. Всегда рассматривайте веб-приложение как службу API. Приложения для смартфонов, веб-сайты с JS и настольные приложения сегодня очень популярны, потому что клиенты подключаются через API. Поскольку API не различает клиентов, он может продолжать обслуживать любого из них.

- Обязательно все кешируйте. Поскольку кэш хранит данные, которые позволяют быстрее обслуживать будущие запросы, они значительно улучшают масштабируемость и производительность.

- Автоматизируйте процессы поставки и обслуживания приложения. Если вы хотите, чтобы система работала эффективно каждый день, вы должны следить за ней и регулярно предоставлять обновления. Трата времени и усилий на поддержание приложения обеспечивает бесперебойную работу.

- Стремитесь к приложению, которое не будет зависеть от какого либо состояния (stateless). Такое подход актуален для распределенной архитектуры, которая поддерживает горизонтальную масштабируемость. Пока компоненты остаются без состояния, они могут легко перераспределяться в случае поломки и масштабироваться для адаптации к изменениям нагрузки.

При проектировании крупной веб-системы необходимо провести тщательное планирование, которое поможет в долгосрочной перспективе. Ниже представлены основные принципы, влияющие на разработку высоконагруженных масштабируемых веб-систем.

Доступность. Время безотказной работы сайта крайне важно для репутации и функциональности многих компаний. Для некоторых крупных сайтов недоступность даже в течение нескольких минут может привести к огромной потере доходов, поэтому проектирование их систем, обеспечивающих постоянную доступность и устойчивость к сбоям, является как фундаментальным бизнесом, так и технологическим требованием. Высокая доступность в распределенных системах требует тщательного рассмотрения избыточности для ключевых компонентов, быстрого восстановления в случае частичных сбоев системы и постепенного снижения производительности при возникновении проблем.

Производительность. Производительность веб-сайта стала важным показателем для большинства сайтов. Скорость веб-сайта влияет на работу и удовлетворенность пользователей, а также ранжирование поисковых систем, все это влияет на удержание аудитории и доход. Результатом хорошей производительности является создание системы, которая оптимизирована для быстрых ответов и низких задержек.

Надежность. Система должна быть надежной, чтобы запрос данных постоянно возвращал одни и те же данные. В случае, если данные изменяются или обновляются, этот же запрос должен вернуть новые данные. Пользователи должны знать, что если какие то данные записаны в систему, то они сохранятся и их можно будет использовать в дальнейшем.

Масштабируемость. Когда речь идет о любой большой распределенной системе, размер системы является это только одним фактором, который необходимо учитывать. Не менее важным является усилие, необходимое для

увеличения пропускной способности системы, необходимой для обработки больших объемов нагрузки, обычно называемой масштабируемостью системы. Масштабируемость может относиться ко многим различным параметрам системы: сколько дополнительного трафика она может обработать, насколько легко добавить больше места для хранения и обработки данных.

Управляемость. Разработка системы, которая проста в эксплуатации, является еще одним важным соображением в проектировании распределенной системы. Управляемость системы приравнивается к масштабируемости операций: обслуживание и обновления. Что касается управляемости, следует учитывать простоту диагностики проблем, которые возникают в ходе работы, простоту внесения изменений и обновлений, а также простоту эксплуатации системы.

При разработке любого веб-приложения важно учитывать перечисленные выше ключевые принципы. Некоторые их них могут влиять больше, другие меньше, но важно учесть каждый аспект, чтобы избежать дорогостоящих проблем в будущем.

Хотя инвестирование ресурсов в полное масштабирование системы, на начальном этапе, является неразумным, важно правильно выбрать архитектурный подход в проектировании системы, учитывая потенциал высокой масштабируемости в будущем. Все это поможет сэкономить ресурсы и время позже. Существуют основные факторы построения масштабируемой веб-архитектуры, облегчающие масштабируемость по мере необходимости, которые будут исследованы в следующей разделе.

1.2. Анализ основных типов архитектур для построения современного программного обеспечения

Понимание наилучшего способа разработки и развертывания приложений сегодня является важным фактором для любой организации, управляемой данными. Такие опции, как сервис-ориентированная архитектура (SOA) и микросервисы, предлагают ценную гибкость для создания и запуска приложений, которые не подходят для традиционных монолитных подходов. Однако стоит понимать различия между ними, чтобы определить, что лучше всего подходит для вашего приложения. В этом разделе мы рассмотрим различные типы архитектуры, а также их характеристики и то, как они используются в разработке программного обеспечения.

Монолитное приложение построено как единое целое, в котором интерфейс пользователя и доступа к данным объединены в одну программу на одной платформе. Корпоративные монолитные приложения состоят из трех частей: База данных, состоящая из множества таблиц, обычно в системе управления реляционными базами данных. Клиентский пользовательский интерфейс, состоящий из HTML и JavaScript, запускается в браузере. Серверные приложения, которые работают как посредник между

пользовательским интерфейсом и базой данных, будут обрабатывать HTTP-запросы, выполнять некоторую специфичную для домена логику, извлекать и обновлять данные из базы данных и заполнять HTML-представления для отправки в браузер.

Трудности при использовании монолитной архитектуры возникают при масштабировании приложения. Каждый раз, когда вы строите, тестируете и внедряете новый функционал, вы должны изменить весь монолит, потому что модули сильно зависят друг от друга. Монолитная архитектура наиболее эффективна в небольших проектах с четко определенной областью, где вы вряд ли будете поддерживать или развивать кодовую базу на постоянной основе.

SOA - это архитектурный подход для определения, связывания и интеграции бизнес-сервисов многократного использования, которые имеют четкие границы и независимы от своих собственных функций. Эти сервисы взаимодействуют друг с другом, чтобы обеспечить простую передачу данных, которая может включать два или более сервисов, координирующие деятельность. Сложность каждого сервиса в SOA обычно очень низкая, и они взаимодействуют друг с другом через набор API.

Обычно SOA используется в приложениях банковского сектора и страхования, где есть доступ к разным базам данных и необходимо часто получать бизнес-данные и технические детали для построения графического интерфейса. SOA дают вам большую гибкость при построении сложных архитектур, и если один сервис отвалится при развертывании, то это не приведет к падению всей системе. Один очевидный недостаток заключается в том, что, несмотря на простоту отдельных сервисов, архитектура может стать слишком сложной, чтобы соответствовать растущим требованиям бизнеса. Поэтому, если нет необходимости разделять отдельные функции вашей системы, то лучше воздержаться, поскольку это будет требовать больших ресурсов.

Микросервисная архитектура разделяет приложение на более мелкие, полностью независимые компоненты, что обеспечивает им большую гибкость и масштабируемость. Это логическая эволюция SOA, которая соответствует современным бизнес-процессам и требованиям.

Микросервисы решают проблемы устаревших монолитных систем. Этот тип архитектуры состоит из большого количества небольших сервисов, каждый из которых выполняет свои собственные функции и может быть независимо развернут, такой сервис легче понимать, разрабатывать и тестировать для обеспечения непрерывной поставки и улучшения. Архитектура микросервисов подходит для облачной платформы, поэтому, когда она будет готова, этот тип архитектуры может перейти на облачную платформу[9].

Проведем сравнительный анализ каждого типа архитектуры по основным критериям оценки: проектирование, масштабируемость, гибкость и разработка. Данные анализа представлены в таблице 1.

Таблица 1

Сравнительный анализ различных типов архитектур

	Микросервисы	SOA	Монолит
Проектирование	Микросервисы построены в виде небольших приложений, которые взаимодействуют друг с другом	Размер сервисов может варьироваться от небольших приложений до очень крупных корпоративных сервисов, которые включают больше бизнес-функций	Монолитные приложения разрастаются до огромных размеров, и в таких условиях сложно понять всю полноту приложения.
Масштабируемость	Микросервисы существуют как независимые единицы развертывания и могут масштабироватьс я независимо от других сервисов.	Зависимости между службами и повторно используемыми компонентами могут создавать проблемы масштабировани я.	Масштабировани е монолитных приложений часто может быть проблемой.
Гибкость	Маленькие независимые развертываемые модули упрощают управление сборкой и выпуском, тем самым обеспечивая высокую степень гибкости приложения.	Высокий уровень совместного использования компонентов, что увеличивает зависимости и ограничивает возможности управления.	Трудно добиться гибкости при повторном развертывании монолитных артефактов приложений.
Разработка	Разработка мискросервисов позволяет разработчикам	Многоразовые компоненты и стандартные методы помогают	Монолитные приложения реализуются с использованием

	использовать подходящую среду разработки для поставленной задачи.	разработчикам в реализации.	одного стека разработки, который может ограничивать доступность других технологий в разработке.

На основе этих данных можно сделать вывод что, при сопоставлении с SOA и монолитной архитектурой, кажется, что использование микросервисов является шагом вперед. Главное преимущество, в сравнении с другими архитектурами в том, что микросервисы разбита на несколько отдельных сервисов, каждый из которых может быть развернут, а затем повторно развернут независимо. Но это может быть слишком сложным для того, что нужно вашему бизнесу, так как более простая платформа может быть проще в реализации и экономичнее. Необходимо четко понимать ситуацию, в которой использование микросервисов принесет больше пользы чем проблем.

1.3. Обзор и анализ проблемы использования монолитной архитектуры в крупных веб системах. Детальное сравнение микросервисов и монолитной архитектуры

В настоящее время многие компании, такие как Netflix, Amazon и eBay, перенесли свои приложения и системы в облако и используют микроверсивсную архитектуру, поскольку модель облачных вычислений позволяет этим компаниям масштабировать свои вычислительные ресурсы в соответствии с их использованием [1].

При выборе монолита или микросервисов как архитектуры вашей системы или приложения, необходимо правильно понимать контекст. Некоторые уверенно начинали с микросервисов, в то время как другие вначале придерживались монолита и в конечном итоге переходили к микросервисам по мере роста их стартапов. Рассмотрите собственный контекст и следующие сценарии, чтобы определить, какая архитектура соответствует вашей ситуации.

В данном разделе мы сравним микросервисы и монолитную архитектуру, обсудим, какие команды и проекты должны использовать какую архитектуру, а также рассмотрим их преимущества и недостатки.

Монолитная архитектура считается традиционным способом построения приложений. Монолитное приложение строится как единое и неделимое целое. Обычно такое решение содержит клиентский

пользовательский интерфейс, серверное приложение и базу данных. Он унифицирован, и все функции управляются и обслуживаются в одном месте.

Обычно монолитные приложения имеют одну большую кодовую базу и не имеют модульности. Если разработчики хотят что-то обновить или изменить, они получают доступ к той же базе кода. Таким образом, они вносят изменения во весь стек одновременно.

Сильные стороны монолитной архитектуры:

- Меньше сквозных проблем. Сквозные проблемы - это проблемы, которые влияют на все приложение, такие как ведение логирования, кэширование и мониторинг производительности. В монолитном приложении эта область функциональности касается только одного приложения, поэтому с ним проще работать.
- Более простая отладка и тестирование. В отличие от архитектуры микросервисов, монолитные приложения гораздо проще отлаживать и тестировать. Поскольку монолитное приложение представляет собой единый неделимый блок, вы можете выполнить сквозное тестирование намного быстрее.
- Прост в развертывании. Еще одним преимуществом, связанным с простотой монолитных приложений, является более простое развертывание. Когда дело доходит до монолитных приложений, вам не нужно обрабатывать много развертываний - только один файл или каталог.
- Прост в разработке. Пока монолитный подход является стандартным способом построения приложений, любая инженерная группа обладает необходимыми знаниями и возможностями для разработки монолитного приложения.

Слабые стороны монолитной архитектуры:

- Понимание кода и структуры. Когда монолитное приложение расширяется, оно становится слишком сложным для понимания. Кроме того, сложная система кода в одном приложении сложна в управлении.
- Внесение изменений. Сложнее реализовать изменения в большом и сложном приложении с очень сильной зависимостью. Любое изменение кода влияет на всю систему, поэтому оно должно быть тщательно скоординировано. Это делает общий процесс разработки намного дольше.

- Масштабируемость. Вы не можете масштабировать компоненты независимо, только все приложение.

- Барьер для использования новые технологий. Крайне проблематично применить новую технологию в монолитном приложении, потому что тогда все приложение должно быть переписано.

В то время как монолитное приложение представляет собой единое единое целое, архитектура микросервисов разделяет его на совокупность более мелких независимых единиц. Эти подразделения выполняют каждый процесс подачи заявки как отдельную услугу. Таким образом, все сервисы имеют свою собственную логику и базу данных, а также выполняют определенные функции.

Мартин Фаулер определил архитектуру микросервисов как подход к разработке набора небольших сервисов, работающих как единое приложение. Сервисы обмениваются данными через легковесные механизмы, такие как API ресурсов HTTP, и каждый сервис работает независимо в своем собственном процессе [2].

Другими словами, в архитектуре микросервисов вся функциональность разделена на независимо развертываемые модули, которые взаимодействуют друг с другом через определенные методы, называемые API (интерфейсы прикладного программирования). Каждый сервис охватывает свою собственную область и может обновляться, развертываться и масштабироваться независимо.

Сильные стороны микросервисной архитектуры:

- Независимые компоненты. Во-первых, все службы могут быть развернуты и обновлены независимо, что дает большую гибкость. Во-вторых, ошибка в одном микросервисе влияет только на конкретный сервис и не влияет на все приложение. Кроме того, гораздо проще добавлять новые функции в приложение микросервиса, чем монолитное.

- Простота в управлении и понимания. Микросервисное приложение, разбитое на более мелкие и простые компоненты, легче для понимания и управления. Вы просто концентрируетесь на конкретной услуге, которая связана с вашей бизнес-целью.

- Лучшая масштабируемость. Другое преимущество подхода на основе микросервисов состоит в том, что каждый элемент может масштабироваться независимо. Таким образом, весь процесс является менее затратным и экономичным по сравнению с монолитами, когда необходимо масштабировать все приложение, даже если в этом нет необходимости. Кроме того, у каждого монолита есть ограничения с точки зрения масштабируемости,

поэтому чем больше пользователей вы приобретаете, тем больше у вас проблем с монолитом. Поэтому многие компании в конечном итоге восстанавливают свои монолитные архитектуры.

- Гибкость в выборе технологии. Инженерные команды не ограничены технологией, выбранной с самого начала. Они могут свободно применять различные технологии и структуры для каждого микросервиса.
- Высокий уровень отказоустойчивости. Любая ошибка в приложении микросервисов влияет только на конкретную услугу, а не на все решение. Таким образом, все изменения и эксперименты реализованы с меньшими рисками и меньшим количеством ошибок.

Слабые стороны микросервисной архитектуры:

- Дополнительная сложность. Поскольку архитектура микросервисов является распределенной системой, вы должны выбрать и настроить соединения между всеми модулями и базами данных. Кроме того, поскольку такое приложение включает в себя независимые службы, все они должны развертываться независимо.
- Распределенная система. Архитектура микросервисов представляет собой сложную систему, состоящую из нескольких модулей и баз данных, поэтому все соединения должны обрабатываться осторожно.
- При создании приложения для микросервисов вам придется столкнуться с рядом сквозных проблем, которые влияют на все приложение, такие как ведение логирования, кэширование и мониторинг производительности.
- Тестирование. Множество независимо развертываемых компонентов значительно усложняет тестирование решения на основе микросервисов.

Сравним монолитную архитектуру и микросервисы на основных критериях проектирования системы в таблице 2.

Таблица 2

Сравнительный анализ монолитной архитектуры и микросервисов на основных критериях проектирования системы

Критерий	Монолит	Микросервисы

Развертывание (Deployment)	Позволяют вам настроить развертывание один раз, а затем просто настроить его на основе текущих изменений. Однако, существует также только одна точка отказа во время развертывания, если что то пойдет не так, вы можете сломать весь ваш проект.	Требуют гораздо больше работы, вам нужно будет развернуть каждый сервис независимо, позаботиться об оркестрации и настроить CI/CD. Но есть и большой плюс, если что-то пойдет не так, вы сломаете только один небольшой микросервис, что менее проблематично, чем весь проект. Откатить один маленький микросервис также намного проще, чем и все приложение.
Обслуживание (Maintenance)	Достаточно прост в обслуживание так как представлен в виде одного приложения, которое необходимо поддерживать.	нужны знания DevOps и умение работать с контейнерами (Docker) и оркестрацией (Kubernetes, Docker Swarm), которые помогут управлять инфраструктурой с помощью множества движущихся частей. Также важно отслеживать и поддерживать рабочее состояние конфигурации CI для каждого микросервиса и всей инфраструктуры.
Надежность (Reliability)	Поломка любой, даже маленькой, функциональности повлечет за собой сбой всей системы, которая не будет отвечать пользователям.	Поломка одного микросервиса не повлечет за собой сбой в работе всего приложения.

Масштабируемость (Scalability)	Трудно масштабировать, Вы не можете масштабировать систему по горизонтали, оставляя только вертикальное масштабирование для вашего монолитного.	Хорошо масштабируется, ресурсы могут использоваться более точечно и позволяют масштабировать только те части, которые требуют больше ресурсов.
Стоимость (Cost)	С небольшим приложением вы можете запустить на хосте за небольшие деньги. Однако, в более крупном монолитном приложении вы можете разместить очень дорогой экземпляр, потому что вы не можете разместить его раздельно на нескольких небольшими дешевыми хостами.	Исходя из лучшей масштабируемостью микросервисов вы можете настроить автоматическое масштабирование и оплачивать его только тогда, когда объем пользователей действительно требует больше ресурсов. В то же время, чтобы поддерживать эту инфраструктуру в рабочем состоянии, вам нужны устройства, которые должны быть оплачены.
Разработка (Development)	Быстрый запуск приложения, но через какое то время кодовая база разрастается и дальнейшая разработка проекта становится сложным и трудоемким процессом.	Сложно в начале, требует интеграцию и разработку дополнительных вещей, таких как контейнеризация и оркестрация. Однако со временем процесс разработки становится понятным и гибким.
Релизы (Releasing)	Имеет много внутренних зависимостей, которые невозможно разбить. Существует риск того,	Позволяют быстрее выпускать новые функции, сокращая время QA, время сборки и время

	что вы будете сильно зависеть от изменений в вашей команде, которые потенциально могут отложить релизы.	выполнения тестов. Все это дает возможность релизиться часто и качественно.

Также было проведено другое исследование, в ходе которого были получены несколько сценариев, которые указывают на то, какой вид архитектуры необходимо использовать в зависимости от того, на какой стадии находится ваша система и что она требует. Данные с полученными сценариями представлены в таблице 3.

Таблица 3

Сравнительный анализ монолитной архитектуры и микросервисов на базе основных сценариев состояния продукта и команды

Фактор	Использование монолита	Использование микросервисов
Команда	Ваша команда находится на начальной стадии. Команда состоит из 2-5 человек и не может работать с более широкой архитектурой микросервисов с высоким уровнем накладных расходов.	В команде более 10 разработчиков широкого спектра, которые имеют глубокие знание в различных направлениях разработки
Стадия проекта	Вы создаете непроверенный продукт и проверяете его жизнеспособность. Если это новая идея, она может разворачиваться и развиваться с течением времени. Ваша цель сделать продукт как можно быстрее, даже если вы в конечном итоге его полностью переделаете.	Вы занимаетесь проектом в течение продолжительного времени и имеете глубокую экспертизу в области применения. Ваш продукт имеет широкую пользовательскую аудиторию, бизнес модель успешна на рынке.

Поставка услуг	Вы можете обновлять весь продукт единоразово в определенные удобные вам периоды.	Вам нужна быстрая независимая поставка услуг для отдельных функциональностей. Часть продукта может обновляться раз в месяц, а другая через день.
Производительность	Вам не нужна высокая производительность, так как вы имеете относительно небольшой трафик.	Некоторые части системы нуждаются в крайне высокой производительности для того чтобы справиться с растущим трафиков и нагрузками
2	У вас нет опыта работы с микросервисами и вы не готовы тратить время и ресурсы на ее изучение.	У вас есть опыт работы с микросервисами и вы четко понимаете механизм их работы.

2. Исследование проблема маршрутизации транспортных средств (VRP)

2.1. Определение и классификация основных видов задачи маршрутизации транспортных средств (VRP)

Задача маршрутизации транспортных средств (VRP) может быть определена как задача поиска оптимальных маршрутов для доставки или забора от одного или нескольких складов до ряда городов или клиентов, при этом удовлетворяя некоторым ограничениям. Сбор бытовых отходов, автоцистерны для доставки бензина, дистрибуция товаров, вывоз снега и доставка почты являются наиболее часто используемыми приложениями VRP. VRP играет жизненно важную роль в дистрибуции и логистике. Огромные исследовательские усилия были посвящены изучению VRP с 1959 года, когда Данциг и Рамзер описали эту проблему как обобщенную проблему задачи коммивояжера (TPS) [3]. Были проведены исследования по нескольким вариантам VRP, таким как задача маршрутизации транспортного средства с временными окнами (VRPTW), задача маршрутизации транспортного средства с учетом забора и доставки (VRPPD) и задача маршрутизации с учетом загруженности транспортного средства (CVRP).

Целью VRP является оптимальная доставка для группы клиентов с известными требованиями к маршрутам транспортных средств с минимальной стоимостью, которые начинаются и заканчиваются на складе.

В классическом варианте VRP клиенты известны заранее. Более того, время в пути между покупателями и время обслуживания каждого покупателя были известны. Классический VRP можно определить следующим образом: Пусть G = (V, A) - граф, где V = {1. , , n} - это множество вершин, представляющих города с депо, расположенным в вершине 1, а A - это множество дуг. С каждой дугой (i, j) i ≠ j связана неотрицательная матрица расстояний C = (cij). В некоторых случаях cij может интерпретироваться как стоимость поездки или время в пути. Когда C симметричен, часто удобно заменить A на множество E неориентированных ребер. Кроме того, предположим, что в депо имеется m доступных транспортных средств, где mL <m <mU. Когда mL = mU, m считается фиксированным. Когда mL = 1 и mU = n - 1, m называется свободным. Когда m не является фиксированным, часто имеет смысл связывать фиксированную стоимость f с использованием транспортного средства. VRP состоит из разработки набора маршрутов транспортных средств с наименьшей стоимостью таким образом, чтобы:

- Каждый город в V \ {1} посещался ровно один раз ровно одним транспортным средством.
- Все автомобильные маршруты начинаются и заканчиваются в депо.
- Некоторые поставленные ограничения выполнены

Задача маршрутизации с учетом вместимости транспортного средством (CVRP) можно описать следующим образом: Пусть G = (V ’, E) приведен неориентированный граф, где V’ = {0, 1,. , , , n} - это множество из n + 1 вершин, а E - это множество ребер. Вершина 0 представляет депо, а набор вершин V = V ’\ {0} соответствует n клиентам. Неотрицательная стоимость dij связана с каждым ребром {i, j} ∈ E. единицы qi поставляются из депо 0 (мы предполагаем, что q0 = 0). Набор из m идентичных транспортных средств грузоподъемностью Q размещен в депо 0 и должен использоваться для снабжения клиентов. Маршрут определяется как простой цикл графа G с наименьшей стоимостью, проходящий через депо 0 и такой, что общая потребность в посещенных вершинах не превышает вместимость транспортного средства. Практическая важность CVRP обеспечивает мотивацию усилий, связанных с разработкой эвристических алгоритмов [7] .

Задача маршрутизации транспортного средства с временными окнами (VRPTW) является обобщением известного VRP. Это можно рассматривать как объединенную проблему маршрутизации и планирования транспортных средств, которая часто возникает во многих реальных приложениях. Он заключается в том, чтобы оптимизировать использование парка транспортных

средств, которые должны сделать несколько остановок для обслуживания группы клиентов, и указать, какие клиенты должны обслуживаться каждым транспортным средством, и в каком порядке минимизировать стоимость с учетом вместимости транспортного средства и ограничения по времени обслуживания. Проблема заключается в назначении транспортных средств для поездок таким образом, чтобы стоимость назначения и соответствующая стоимость маршрутизации были минимальными. VRPTW может быть определено следующим образом: Пусть G = (V, E) - связный орграф, состоящий из набора из n + 1 узлов, каждый из которых может быть достигнут только в течение определенного интервала времени или временного окна, и набора E дуг с неотрицательными весами, представляющими расстояния прохождения и связанное время прохождения. Пусть один из узлов будет обозначен как депо. Каждый узел i, кроме депо, запрашивает услугу размера qi [23].

Проблемы, которые необходимо решить в реальных ситуациях, обычно намного сложнее, чем классический VRP. Одна сложность, которая возникает на практике, заключается в том, что товары должны быть не только доставлены из склада покупателям, но также должны быть забраны у ряда покупателей и возвращены в хранилище. Эта проблема хорошо известна как VRP с использованием забора и доставки (VRPPD). В литературе VRPPD также называют VRP с транзитными соединениями (VRPB). Проблема может быть разделена на два независимых CVRP; один для клиентов с доставкой и один для клиентов с забором, так что некоторые транспортные средства будут предназначены для забора, а другие - для доставки.

Формулировка VRP может быть представлена следующим образом: Пусть xij - целочисленная переменная, которая может принимать значение {0, 1}, ∀ {i, j} ∈ E \ {{0, j}: j ∈ V } и значение {0, 1, 2}, ∀ {0, j} ∈ E, j∈V. Обратите внимание, что x0j = 2, когда в решении выбран маршрут, включающий одного клиента j.

VRP можно сформулировать в виде следующей целочисленной программы:

Минимизировать ∑ ≠ ji ijij xd (1)

При условии:

∑ = jijx1, ∈i ∈ V (2),

∑ = iijx1, ∀j ∈ V (3),

v (S) | S | xiij −≥∑, {S: S ⊆ V \ {1}, | S | ≥ 2} (4),

xij ∈ {0, 1}, ∀ {i, j} ∈ E; i (j) (5)

В этой формулировке (1), (2), (3) и (5) определяют модифицированную проблему присвоения (то есть присвоения на главной диагонали запрещены). Ограничения (4) являются ограничениями на устранение под-туров: v (S) - это

подходящая нижняя граница для числа транспортных средств, необходимых для посещения всех вершин S в оптимальном решении.

VRP был представлен как NP-Hard, что подтолкнуло исследователей к использованию эвристики. Тем не менее, точные алгоритмы были также применены для VRP.

Были опубликованы труды о нескольких применениях генетических алгоритмов (ГА) в VRP, поскольку в последнее время ГА широко используются в современной метаэвристике. Применение ГА также сообщалось для варианта VRP, для задачи маршрутизации в нескольких депо и задачи маршрутизации школьного автобуса [24].

С другой стороны был использован гибридный подход к VRP с использованием нейронных сетей (НС) и ГА. Исследования показали, что ГА пока что не оказали большого влияния на базовый VRP. Оптимизация с использованием муравьиного алгоритма - это еще один недавний подход к сложным комбинаторным проблемам с несколькими успешными приложениями, включая VRP[4]. С учетом 2-оптимальной эвристики, включенной для улучшения отдельных маршрутов, создаваемых искусственными муравьями, этот подход также дал результаты.

2.2. Определение эвристических алгоритмов и анализ работы принципа “разрушения и воссоздания” в задачах VRP

Поскольку VRP является NP-трудной задачей, наиболее успешными алгоритмами поиска оптимального маршрута являются эвристические алгоритмы. Задача эвристики - найти решение в разумные сроки, достаточно подходящее для решения стоящей проблемы. Это решение может быть не лучшим из всех решений этой проблемы, или оно может просто приближаться к точному решению. Но это все еще ценно, потому что обнаружение этого не требует чрезмерно долгого времени.

Термин эвристика используется для алгоритмов, которые находят решения среди всех возможных, но они не гарантируют, что будет найдено лучшее, поэтому их можно рассматривать как приблизительные, а не точные алгоритмы. Эти алгоритмы обычно находят решение, близкое к лучшему, и находят его быстро и легко. Иногда эти алгоритмы могут быть точными, то есть на самом деле они находят лучшее решение, но алгоритм по-прежнему называется эвристическим, пока это лучшее решение не окажется наилучшим[5]. Метод, используемый в эвристическом алгоритме, является одним из известных методов, таких как жадность, но для простоты и скорости алгоритм игнорирует или даже подавляет некоторые требования задачи.

В математическом программировании эвристический метод, или сокращенно эвристика, - это процедура, которая определяет хорошие или почти оптимальные решения задачи оптимизации. Эвристика может давать результаты самостоятельно, или они могут использоваться в сочетании с

алгоритмами оптимизации для повышения их эффективности, например, они могут использоваться для получения хороших начальных значений. Эвристические алгоритмы получили широкое использование в решении сложных вычислительных задач. Сегодня эвристика лежит в основе всей области искусственного интеллекта и компьютерного моделирования мышления, поскольку они могут использоваться в ситуациях, когда нет известных алгоритмов. [6]

Результаты использование данного алгоритма в NP-трудных задачах делают эвристику единственным приемлемым вариантом для решения множества сложных задач оптимизации, которые необходимо регулярно решать в реальных приложениях.

Одним из наиболее распространенных подходов к VRP и его модификациям является принцип «разрушить и воссоздать», когда выбранный эвристический алгоритм строит решение, затем некоторые узлы отбираются из решения, а затем другой заново восстанавливает новое решение путем добавление исключенных узлов.

Перейдем к оптимизации маршрутизации транспортных средств. Мы рассматриваем случай, когда выезд машин начинается с центрального склада. Все транспортные средства имеют заданную максимальную вместимость. Они обслуживают множество клиентов (точек) с известными требованиями. Решение VRP состоит из набора маршрутов транспортных средств с минимальной стоимостью, удовлетворяющих следующим условиям: Каждое транспортное средство начинает и заканчивает свой маршрут в центральном депо. Каждый клиент обслуживается ровно один раз. Сумма требований всех клиентов, обслуживаемых одним транспортным средством, не превышает его вместимости. Стоимость набора маршрутов для VRP является суммой длины всех маршрутов. Каждый клиент может добавить временное окно или ограничение временного интервала к проблеме, то есть для каждого клиента есть самое раннее и самое позднее время, когда транспортное средство может обслуживать клиента. Если такие ограничения введены, мы говорим о проблеме маршрутизации транспортных средств с временными окнами (VRPTW).

Вкратце, задача состоят из N пунктов обслуживания клиентов и одного склада. И расстояния, и время в пути между клиентами определяются соответствующими евклидовыми расстояниями. Следовательно, эта библиотека включает в себя многие отличительные особенности маршрутизации транспортных средств с временными окнами: размер парка, вместимость транспортного средства, пространственное и временное распределение клиентов, плотность временных окон, ширина временных окон и время обслуживания клиентов. Задача этой проблемы - обслуживать всех клиентов, минимизируя, в первую очередь, количество транспортных средств, а затем минимизируя расстояние.

Используя принцип «разрушения и воссоздания» мы строим решение задачи следующим образом: начиная с исходного решения, он распадается на части решения, что приводит к (i) набору заданий, которые больше не

обслуживаются транспортным средством, и (ii) частичному решению, содержащему все остальные задания. Таким образом, этот шаг называется шагом разрушения. Основываясь на частичном решении (ii), все работы из (i) снова реинтегрируются, что называется воссозданием, дающей новое решение. Если новое решение имеет определенное качество, оно принимается в качестве нового лучшего решения, после чего начинается новая итерация «разрушай и воссоздай». Эти шаги повторяются снова и снова до тех пор, пока не будет достигнут определенный критерий завершения (например, время вычислений, количество повторений и т. д.).

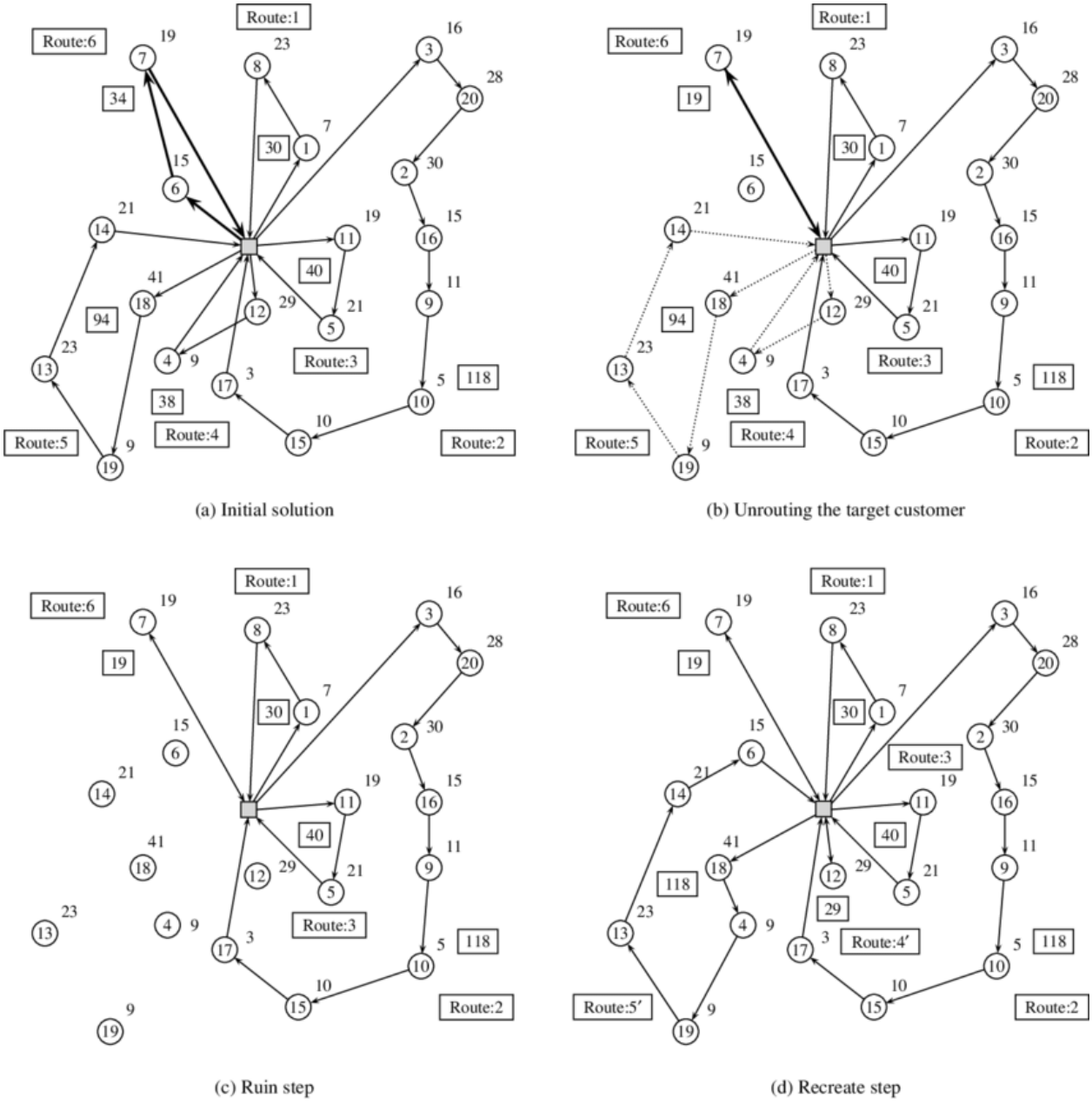

Рисунок 1. Принцип работы адгоритма «разрушения и воссоздания»

Подводя итоги, на основании полученных данных, мы можем выделить основные плюсы в использовании принципа «разрушения и воссоздания» при решение задачи VRP:

- Лучше всего подходит для сложных задач, которые имеют много ограничений и большое количество возможных решений.
- Является универсальной эвристикой, которая может быть использована для решения ряда классических типов VRP.
- Может быть вычислен одновременно интуитивно понятным способом.
- Стратегии поиска могут быть легко изменены на маленькие и большие шаги в зависимости от сложности проблемы.
- Может генерировать совершенно новые соседские структуры.
- Привлекателен по своей структуре и сравнительно прост для понимания.
- Существует четкое различие между разрушением и воссозданием, что значительно облегчает проверку ограничений.

2.3. Анализ подходящих инструментов для планирования маршрутов на базе VRP.

Существует несколько основных инструментов для решения задачи маршрутизации транспортных средств (VRP):

- Jsprit - это набор инструментов с открытым исходным кодом на основе Java для решения сложных задач коммивояжера (TSP) и проблем маршрутизации транспортных средств (VRP). Это легкий, гибкий и простой в использовании, основанный на единой универсальной метаэвристике, решаемой в настоящее время
- OR-tools - это программное обеспечение с открытым исходным кодом для комбинаторной оптимизации, которое стремится найти наилучшее решение проблемы из очень большого набора возможных решений. Вот несколько примеров проблем, которые решает OR-Tools: сложная задача коммивояжера (TSP) и проблем маршрутизации транспортных средств (VRP)
- OptaPlanner - решение на основе ИИ. Он оптимизирует задачи планирования и планирования, такие как проблема маршрутизации транспортных средств, составление списка сотрудников, планирование технического обслуживания, распределение задач,

школьное расписание, оптимизация облачных вычислений, планирование конференций, планирование рабочих мест, упаковка в складские помещения и многое другое.

В таблице 4 приведен сравнительный анализ каждого инструмента по основным критериям оценки ПО.

Таблица 4

Сравнительный анализ инструментов OR-tools, Jsprit и OptaPlanner по основным критериям оценки ПО.

	OR-tools	Jsprit	OptaPlanner
Язык и среда	C++, Python, Java, Google AI	Java, GraphHopper	Java, Red Hat
Производительность	Высокая	Высокая	Средняя
Порог вхождения	Средний	Низкий	Средний
Поддержка	Хорошая документация, средняя сообщество	Хорошая документация, большое сообщество	Средняя документация, маленькое сообщество

Для выбора подходящего инструмента необходимо определить какое решение будет оптимальным для решения задачи VRP. В таблице 5 приведен сравнительный анализ каждого инструмента на основании возможностей в решение задачи VRP.

Таблица 5

Сравнительный анализ инструментов OR-tools, Jsprit и OptaPlanner на основании возможностей в решение задачи VRP

	OR-tools	Jsprit	OptaPlanner
Учет временных окон	✔	✔	✔
Учет веса и объема	✔	✔	✔
Учет веса, объема и временных окон одновременно	✘	✔	✘

Использование нескольких точек забора (Multiple depot)	✔	✔	✔
Забор и доставка	✔	✔	✘
Использование своей матрицы дистанций (Distance matrix)	✘	✔	✘
Гибкая система ограничений (constraints)	✔	✔	✘

3. Исследование процесса миграции монолитной архитектуры на микросервисы

3.1. Причины и предпосылки перехода от монолитной архитектуры к микросервисам

Крупные корпоративные приложения обычно начинаются как монолиты. Эта практика кажется разумной, так как поддержка менее требовательна, и система работает хорошо из ограниченных областей. Микросервисы имеют преимущество, однако, только в тех случаях, когда система становится слишком большой и сложной. Практика показывает, что наступает момент, когда появляется необходимость разбивать монолит на маленькие автономные части, каждая из которых разворачивается и поддерживается гибкой командой, оставляя позади проблемы совместной работы команды.

Когда приложение и команда разработчиков становятся больше и достигают определенного размера, компании сталкиваются с критическим узким местом в управлении и менеджменте. Более того, если программный продукт основан на массивной монолитной архитектуре, возникают и технологические проблемы. В таких случаях предприятиям требуется решение для исправления рабочего процесса и улучшения совместной работы над проектом. Микросервисная архитектура является ответом на проблемы, связанные с традиционным внутренним монолитом, когда его сложность требует большей масштабируемости.

Предприятия обычно принимают такое решение только тогда, когда возникает практическая необходимость. В большинстве случаев команда разработчиков, состоящая из более чем 20-25 человек, начинает испытывать

трудности с совместной работой при обслуживании монолитных приложений. Таким образом, инициатива по внедрению микросервисов часто исходит от команды разработчиков или поставщика программного обеспечения.

Микросервисы были изобретены, чтобы отвечать требованиям современного рынка. Компании должны анализировать свои данные, вводить новшества и запускать новые продукты и услуги лучше и быстрее, чем их конкуренты. Они должны быть гибкими, чтобы удовлетворить меняющиеся потребности своих клиентов. Переход на микросервисную архитектуру позволяет им делать это более эффективно.

Рассмотрим основными причины для перехода монолитной системы в микросервисы являются:

- Рост бизнеса. Поскольку приложение обслуживает больше клиентов и обрабатывает больше транзакций, оно нуждается в большей емкости и ресурсах.
- Увеличение трафика. В идеале, система должна иметь возможность масштабироваться автоматически или, по крайней мере, динамически, таким образом, чтобы инфраструктура не развивалась до максимальной пропускной способности для поддержки пикового трафика. Масштабирование монолитных приложений часто может быть проблемой.
- Ускорение выхода обновлений. Для бизнеса очень важно, чтобы добавление или изменение функции занимало дни или недели, а не месяцы и не требовало чрезмерного (и часто дорогостоящего) регрессионного тестирования.
- Сложная поддержка приложения. Монолитные приложения могут превратиться в «большой шарик грязи», ситуация, когда ни один разработчик или группа разработчиков не понимают всего приложения.
- Повторное использование. В монолитных приложениях сильно ограниченное повторное использование существующих функций.
- Операционная гибкость. Трудно добиться операционной гибкости при повторном развертывании монолитных приложений.
- Набор инструментов. По определению монолитные приложения реализуются с использованием одного стека разработки, который может ограничивать доступность подходящего инструмента для работы.
- Гибкость. Проблема в сложности добавления новых функций поверх существующих в монолитном приложении.

Архитектуры микросервисов вписываются в среду гибкой разработки, поскольку разделение монолита на микросервисы часто согласуется с разбиением ориентированной на бункер структуры команды на самоорганизующиеся и автономные группы. Ситуации в реальном мире всегда отличаются от теории, и случай с микросервисами не является исключением. Предприятие должно всегда учитывать свои собственные бизнес-потребности, отраслевые угрозы и возможности, прежде чем переходить на микросервисы.

Подходы, описанные ранее, являются просто ориентирами, которые помогут вам пройти процесс миграции. Каждая бизнес-ситуация уникальна и требует оригинального решения. Миграция на микросервисы принесла пользу многим компаниям, демонстрирующих, что микросервисы имеют преобразующий потенциал для всех видов традиционных и современных предприятий. Изменение неизбежно, и каждый должен быть готов принять его.

3.2. Инфраструктурные требования для перехода к микросервисной архитектуре

Среда, которая поддерживает микросервисы, в основном нуждается в наборе базовых требований для обеспечения определенного уровня работоспособности. Если организация собирается работать, она должна быть готова взять на себя ответственность за их запуск и поддержку. Накладные расходы не будут незначительными. Чтобы развернуть микросервисы, потребуются время и деньги

В каждой организации должна быть внутренняя группа, отвечающая за инфраструктуру, которая будет предоставляться группам разработки и эксплуатации для использования микросервисов. Эта группа должна состоять из лучших разработчиков организации или даже консультантов, а не из организации.

Не существует единого правила настройки инфраструктуры для микросервисов. Но есть стандартный набор направлений, которые описывают функциональные возможности, которые должны быть реализованы в инфраструктуре:

- Непрерывная интеграция и непрерывное развертывания (CI/CD)
- Виртуальные машины или контейнеры
- Мониторинг
- Логирование

Организация должна решить, как создавать, тестировать и развертывать микросервисы. Эти операции должны выполняться автоматически [13]. В настоящее время существует множество систем сборки, обеспечивающих функциональность конвейера [14]. Группа, ответственная за микросервисную

инфраструктуру, должна решить, как это сделать, и выбрать инструменты для этого:

- Контроль исходного кода - где и как должен храниться и поддерживаться исходный код.
- Инструмент сборки - как должен быть построен микросервис.
- Инструмент тестов - как проводить тесты.
- Инструмент развертывания - как следует развертывать микросервис.

Другое важное решение - какую технологию использовать для среды исполнения. Многие предприятия с существующими приложениями, работающими в стабильной инфраструктуре виртуальных машин, выбирают подход «вода в ногу». Развертывая контейнеры на виртуальных машинах, они получают преимущества зрелого мониторинга и изоляции с более быстрыми процессами DevOps. По сравнению с контейнерами, работающими на голом металле, они снижают производительность, масштабируемость и стоимость. Но это, безусловно, правильный способ перехода [15].

Микросервисные архитектуры естественным образом подходят для облачных приложений. Собственное облачное приложение просто определяется как приложение, созданное с нуля для архитектур облачных вычислений. Это просто означает, что наше приложение является нативным облаком, если мы спроектируем его так, как будто оно будет развернуто в распределенной и масштабируемой инфраструктуре [16].

Мониторинг является важной частью инфраструктуры microservices. Используя ниже перечисленные пять принципов, которые позволяют организации установить более эффективный мониторинг. Эти принципы позволят организации решать как технологические изменения, связанные с микросервисами, в дополнение к организационным изменениям, связанным с ними.

- Контролируйте контейнеры и что внутри них.
- Оповещение о производительности сервиса, но не о контейнере представление.
- Мониторинг услуг, которые эластичны и имеют многократное размещение.
- Мониторинг API.
- Сопоставьте свой мониторинг с вашей организационной структурой.

Крайне важно регистрировать информацию о микросервисах в производстве. Чтобы сделать это эффективно, служба регистрации должна быть централизованной и иметь сильный визуализатор. В настоящее время существует множество инструментов, обеспечивающих такую функциональность. Ниже перечислены лучшие практики для регистрации микросервисов [17]:

- Сопоставьте запросы с уникальным идентификатором.
- Включите уникальный идентификатор в ответ.
- Структурируйте ваши данные журнала.
- Добавить контекст к каждому запросу.
- Запись логов в локальное хранилище.

3.3. Миграция устаревшего программного обеспечения на микросервисную архитектуру

Миграция устаревшего монолитного программного обеспечения на микросервисы не может быть проведена быстро. Важно знать, что существует высокая общая стоимость, связанная с разложением существующей системы на микросервисы, и это может занять много итераций [8]. Поскольку унаследованное корпоративное приложение является очень широким термином, нельзя сказать, что это только один хороший способ перехода от устаревшего монолита к архитектуре микросервисов [10]. Поскольку микросервисы являются относительно новым архитектурным стилем и не существует общепринятого способа миграции, разные организации используют разные модели и методы миграции [11].

Ключевой проблемой в этом контексте является извлечение микросервисы из существующих унаследованных монолитных кодовых баз [12]. В этой главе дается обзор различной техники, используемой для выполнить миграцию. В общем, можно выделить две основные стратегии:

- Перестройка приложения с нуля
- Модернизация (рефакторинг) действующей кодовой базы

Не все монолитные приложения могут быть легко перенесены в микросервисная архитектура. Иногда это более экономично выгодно перестраивать приложение с нуля, а не рефакторинг его. У следующих типов устаревших приложений не рекомендуется проводить рефакторинг:

- Очень старые приложения, созданные с использованием старых языков программирования и баз данных.

- Плохо спроектированное приложения, которое будет тяжело изменять.
- Приложение, которое тесно связано с базой данных.

Другой подход к модернизации унаследованных приложений состоит в том, чтобы реорганизовать все с целью разделения унаследованных приложений на микросервисы и объединения этих микросервисов в одну платформу. У каждого из них есть свои преимущества и недостатки. Некоторые из них носят общий характер и могут использоваться с любым типом приложений, тогда как другие являются более конкретными и будут работать только с некоторыми предположениями. Рассмотрим и сравним данные методы, опубликованные в научных статьях.

Метод идентификации микросервисов в монолитных системах, которую описали Алессандра Левковиц, Рикардо Терра и Марко Тулио Валенте[18], состоит из следующих этапов:

- Декомпозиция базы данных - отображение таблиц базы данных в подсистемы.
- График зависимости - создание графика зависимости между фасадами, бизнес-функциями и таблицами базы данных.
- Карта подсистем и определение пар фасадов и таблиц базы данных. На основе графика зависимости определяются пары фасадов и таблицы базы данных.
- Определение кандидатов, которые будут преобразованы на микросервисах. Для каждой отдельной пары, полученной в предыдущем шаге, проверяется код фасада и бизнес-функций, которые находятся на пути от фасада к таблице базы данных в графе зависимостей.
- Создание шлюзов API, чтобы сделать миграцию на микросервисы прозрачной для клиентов. API-шлюз состоит из промежуточного уровня между клиентской стороной и серверным приложением.

Метод декомпозиции на основе потока данных разработан Руи Чен, Шаньшань Ли и Чжэн Ли при помощи анализа сверху вниз[19].. Метод состоит из трех этапов:

- Инженеры вместе с пользователями проводят анализ бизнес-требований и создают детализированную, в то же время детальную диаграмму потока данных бизнес-логики.
- Алгоритм объединяет одни и те же операции с одними и теми же типами выходных данных в виртуальный абстрактный поток данных.

- Алгоритм извлекает отдельные модули «операции и ее выходные данные» из виртуального абстрактного потока данных для представления идентифицированных кандидатов на микросервис.

Метод извлечения микросервисов, позволяющий алгоритмически рекомендовать кандидатов на микросервисы в сценарии рефакторинга и миграции. Авторы Генк Мазлами, Юрген Сито и Филипп Лейтнер представляют инструмент, который поддерживает декомпозицию структурированных сервисов посредством вырезания графов. Внутреннее представление системы, подлежащей разложению, основано на каталоге из 16 различных критериев связи, которые были извлечены из литературы и отраслевых ноу-хау. Артефакты разработки программного обеспечения и документы, такие как доменные модели и варианты использования, служат входом для генерации значений связи, которые формируют представление графа[20].

Метод миграции для разложения приложения на микросервисы предложили Хольгер Кноше и Вильгельм Хасслбринг [21]. Метод модернизации состоит из пяти этапов:

- Определение внешнего фасада сервиса - определение внешнего фасада сервиса, который фиксирует функциональность, требуемую клиентскими системами, в форме четко определенных операций сервиса.
- Адаптация сервисного фасада - реализовать функциональность внешнего сервисного фасада.
- Перенос клиентов на фасад службы - начните использовать вновь созданные интерфейсы внешнего фасада службы.
- Создание внутренних сервисных фасадов - реструктуризация приложений внутри компании.
- Замена внедрений сервиса на микросервисы - приложение заменено на микросервисы.

Метод перехода от монолитной к микросервисной архитектуре предложил Жамак Дехгани [282828]. Предлагаемый поток состоит из следующих принципов:

- Свести к минимуму зависимость от монолита.
- Отнести разделение возможностей на начальный этап
- Провести вертикальное разделение и отнести выпуск данных на начальный этап.
- Отделить то, что важно для бизнеса и часто меняется.

- Сначала определяются макро процессы, затем микро.
- Миграция в атомных эволюционных процессах.

Метод процесс миграции, основанный на SDLC, включая все методы и инструменты, необходимые при проектировании, разработке и реализации предложили Чэнь-Юань Фан и Шанг-Пин предложили процесс миграции, основанный на SDLC, включая все методы и инструменты, необходимые при проектировании, разработке и реализации [22].

В таблице 6 рассмотрены основные методы миграции от монолита к микросервисам с учетом преимуществ и недостатков.

Таблица 6

Методы миграции от монолита к микросервисам с учетом преимуществ и недостатков

Метод	Преимущества	Недостатки
Методика идентификации микросервисов в монолитных системах	Предложенный метод позволяет идентифицировать и классифицировать все подсистемы, а также создавать и анализировать граф зависимостей при оценке и классификации только таблиц базы данных в бизнес-подсистемы, что требует доступа только к исходному коду и режиму базы данных.	В некоторых сценариях могут потребоваться дополнительные усилия для переноса подсистемы на набор микросервисов: подсистем, которые совместно используют одну и ту же таблицу базы данных. Микросервис, представляющий операцию, которая всегда находится в середине другой операции. Бизнес-операции, включающие более одной бизнес-подсистемы в области транзакции.
Метод извлечения микросервисов, позволяющий алгоритмически рекомендовать кандидатов на микросервисы в	Механизм, управляемый потоком данных, гарантирует наиболее тонкодисперсные кандидатуры микросервисов с точки	Выявление одних и тех же операций с данными в определенной степени требует опыта. Кандидатные микросервисы, полученные из

сценарии рефакторинга и миграции.	зрения обработки данных в рамках бизнес-логики. Извлечение может принимать различные текстовые источники, помимо содержимого веб-страницы, и мало заботится о том, куда пойдут выходные данные.	предложенного механизма разложения, могут все еще нуждаться в экспертной оценке, прежде чем они будут разработаны на практике. Предложенный механизм декомпозиции не получил широкого применения в крупных проектах.
Метод извлечения микросервисов, позволяющий алгоритмически рекомендовать кандидатов на микросервисы в сценарии рефакторинга и миграции.	Оценка производительности показывает, что, по большей части, предлагаемый подход масштабируется с учетом размера истории изменений (логическая связь и связь между участниками). Оценка качества показывает, что предлагаемый подход может уменьшить размер микросервисной команды до четверти размера команды монолита или даже меньше.	Одним из ограничений является тот факт, что модель извлечения основана на классах в качестве атомарной единицы вычисления в стратегиях и графике. Использование методов, процедур или функций в качестве атомарных единиц извлечения может потенциально улучшить детализацию и точность перестановки и реорганизации кода.
Метод миграции для разложения приложения на микросервисы	Создание четко определенных, независимых от платформы интерфейсов, основанных на ограниченных контекстах основного домена. Улучшение эволюционируемости приложения.	Некоторые части приложения не могут быть модернизированы с использованием представленного подхода. В частности, некоторые пользовательские интерфейсы, которые основаны на запатентованных

	Сокращение количества точек входа и предотвращение доступа к внутренним компонентам, перенос функциональности не-клиентов в отдельные компоненты и устранение избыточных и устаревших частей приложения.	технологиях, не имеют необходимых средств для абстрагирования сервиса.
Метод перехода от монолитной к микросервисной архитектуре	Процесс миграции с помощью этого подхода можно разделить на небольшие этапы. Можно безопасно остановиться и восстановить.	Очень долгий процесс миграции. Это очень формальный способ без каких-либо измерений.
Метод процесс миграции, основанный на SDLC, включая все методы и инструменты, необходимые при проектировании, разработке и реализации	Специализированный и простой: микросервисы предназначены для решения проблем в одном домене. Отказоустойчивость: ошибка одного микросервиса не может сломать все приложение. Автоматизированный: средства автоматизации, используемые для построения, развертывания, мониторинга.	Сложные настройки среды: конфигурация не так проста, как в системе с монолитной архитектурой, и многие инструменты автоматизации должны быть тщательно настроены для достижения желаемых результатов. Использование большего количества ресурсов. Микросервисы используют несколько инструментов для достижения гибкости архитектуры, таких как Service Discovery и API Gateway.

Каждое устаревшее монолитное приложение уникально. Бизнес-объект, сложность, технологический стек, размер команды, командные навыки и прочее - это вещи, которые могут сильно отличаться в каждом конкретном случае. Нет лучшего способа перехода с монолита на микросервисы, только истории успеха и передовой опыт. Микросервисы - это относительно новый архитектурный стиль, в результате которого отсутствуют общие рекомендации по переносу монолитов на микросервисы.

Выбрать методы и методы миграции, которые лучше всего подходят для организации, очень сложная задача. Первый вопрос, на который всегда нужно отвечать: рефакторинг или перестройка? В большинстве случаев не так много ресурсов и времени, чтобы полностью перестроить решение.

Архитектура микросервисов предоставляет множество различных преимуществ, но также увеличивает сложность решения. Для борьбы с усложнением организации необходимо подготовить специальную инфраструктуру для микросервисов. У каждой организации есть свои причины и цели, почему был выбран переход от монолита к микросервисам. План миграции к микросервисной архитектуре должен ориентироваться на эти цели.

4. Сравнительный анализ технологий и инструментов для построение микросервисной архитектуры

4.1. Микросервисные коммуникации с использование очереди сообщений. Сравнение очереди сообщений (MQ) и REST API

При работе с микросервисами довольно важный момент проектирования то, как сервисы должны общаться друг с другом. Многие люди обычно выбирают RESTful HTTP API, предоставляемый каждой службой, и затем другие службы вызывают его с помощью обычного HTTP-клиента.

Это имеет некоторые преимущества, в первую очередь облегчая обнаружение служб с помощью разрешения DNS и шлюзов API, но также имеет много недостатков. Например, что если вызываемый сервис вышел из строя и не может ответить? Ваша клиентская служба должна реализовать какую-то логику переподключения или восстановления после отказа, иначе вы рискуете потерять запросы и фрагменты информации. Облачная архитектура должна быть устойчивой и корректно восстанавливаться после сбоев. Кроме того, HTTP-запрос является блокирующим API, что делает его асинхронным, что подразумевает некоторые хитрые махинации на стороне клиента.

Использование очередей сообщений на самом деле является довольно старым и качественным решением при работе с несколькими службами связи. Архитектура очереди сообщений требует дополнительной службы, называемой брокером сообщений, которая занимается сбором, маршрутизацией и распространением ваших сообщений от отправителей к нужным получателям.

Архитектуру очереди сообщений можно представить в виде почтовой службе. Вы (producer) отправляете письмо (message) кому-то (consumer), и вы делаете это, указывая адрес (логику маршрутизации для сообщения, например, тему, по которой оно опубликовано) и давая письмо в местное почтовое отделение (брокер сообщений). После того, как вы оставите письмо, вам больше не нужно думать дойдет ли письмо, об этом позаботится почтовая служба. В микросервисной среде это действительно очень надежное решение, поскольку оно добавляет уровень абстракции (сам брокер сообщений) между слабо связанными сервисами и обеспечивает полностью асинхронную связь.

Передача репрезентативного состояния (REST) была определена Роем Филдингом в его докторской диссертации 2000 года, озаглавленной «Архитектурные стили и проектирование сетевых программных архитектур». Доктор Филдинг работал в команде, которая работала с протоколом HTTP. Работая над в этой сфере он превратил свою модель в основной набор принципов, свойств и ограничений, которая теперь называется REST [25].

Взаимодействия RESTful стали жизненно важными для корпоративных вычислений, поскольку сегодня они поддерживают множество API в сети. С учетом сказанного давайте определим, какие проблемы лучше всего решает REST:

- Синхронный запрос / ответ - HTTP (сетевой протокол, по которому транспортируется REST) сам по себе является протоколом запроса / ответа, поэтому REST отлично подходит для взаимодействия запрос / ответ.

- Общедоступные API-интерфейсы. Поскольку HTTP является де-факто транспортным стандартом благодаря работе IETF, транспортный уровень API-интерфейсов, созданных с использованием REST, совместим с любым языком программирования. Кроме того, полезная нагрузка сообщения может быть легко задокументирована с помощью таких инструментов, как Swagger (спецификация OpenAPI). И из-за широкого спектра угроз безопасности, присутствующих в Интернете, экосистема безопасности для REST является надежной, от брандмауэров до OAUTH (аутентификация / авторизация).

Из-за популярности сервисов RESTful сегодня многие компании попадают в ловушку использования REST как инструмента «все в одном». Конечно, некоторая часть этой популярности обусловлена мощью, которую REST предоставляет на основе своих собственных достоинств. Разработчики также привыкли проектировать приложения с синхронным запросом / ответом, поскольку API и базы данных научили разработчиков вызывать метод и ожидать немедленного ответа [26].

Эта чрезмерная зависимость от использования REST и синхронных шаблонов имеет негативные последствия, которые в первую очередь относятся к обмену данными между микросервисами в рамках предприятия и в некоторых случаях противоречат принципам правильной архитектуры микросервисов:

- Тесная связь - всегда будет некоторая связь сервисов вокруг интерфейса (особенно вокруг данных), но при вызове сервиса RESTful разработчик предполагает, что сообщение когда-либо нужно будет доставить только в одно место. Появится проблема, когда в будущем появится услуга или компонент, и им понадобятся. Конечно, вы можете обновить код, чтобы добавить новую конечную точку, но появится лишнее соединение. Вскоре ваш простой микросервис станет оркестратором, который игнорирует главный атрибут микросервиса - «единый в назначении».

- Блокировка - при вызове REST сервиса, ваш сервис блокируется в ожидании ответа. Это снижает производительность приложения, поскольку этот поток может обрабатывать другие запросы. Подумайте об этом так: что, если бармен принял заказ на напиток, сделал коктейль и терпеливо ждал, пока клиент закончит пить свой напиток, прежде чем перейти к следующему клиенту? У этого клиента будет отличный опыт, но все остальные клиенты будут испытывать жажду и быть совершенно несчастными. Бар может добавить дополнительных барменов, но потребуется один для каждого клиента. Очевидно, что это будет дорого для бара, и его невозможно будет увеличивать и уменьшать, поскольку посетители приходят и уходят. Во многом такие же проблемы возникают, когда потоки блокируются в приложениях, ожидающих ответа служб RESTful.

- Обработка ошибок - HTTP был создан для работы в Интернете и часто встречается ситуация, когда браузер застревает, пытаясь получить доступ к веб-странице. Обычно мы нажимаем кнопку обновления, и страница отображается. Но что, если он снова потерпит неудачу? Попробовать обновить еще раз? Можно ли начать реализовывать экспоненциальную человеческую форму, выпив чашку кофе и повторив попытку через несколько минут? Мы не знаем, что делать, поскольку каждая веб-страница отличается и имеет уникальное поведение. Такая же проблема возникает при прямом вызове службы RESTful. Должна ли эта сложная логика повторения находиться в коде сервиса? Если это произойдет, сервис будет еще более тесно связан с другими сервисами, что опять-таки нарушает ключевой принцип сохранения архитектуры микросервисов единым назначением и небольшим по размеру.

Решением многих недостатков, связанных с RESTful / синхронными взаимодействиями, является объединение принципов управляемой событиями архитектуры с микросервисами. Управляемые событиями микросервисы по своей сути асинхронны и уведомляются, когда пора выполнять работу. Во многих случаях асинхронное общение - это количество ежедневных взаимодействий. Рассмотрим пример работы Facebook, было бы невероятно неэффективно переходить к каждому другу и проверять, есть ли у него обновление статуса. Вместо этого мы получаем уведомление, когда друг обновил свой статус или добавил новую информацию, что дает большое удобство в пользовании. Рассмотрим преимущества обмена сообщениями для управляемых событиями микросервисов многочисленны и разнообразны:

- Слабая связь - при использовании обмена сообщениями, в частности, функции публикации / подписки, службы не имеют сведений о других службах. Они уведомляются о новых событиях, обрабатывают эту информацию и производят / публикуют новую информацию. Эта новая информация может быть использована любым количеством служб благодаря публикации / подписке. Слабая связь позволяет микросервисам быть готовыми к бесконечным изменениям, которые происходят на предприятиях.

- Неблокирующая - микросервисы должны работать максимально эффективно, без пустой траты ресурсов, когда многие потоки заблокированы и ожидают ответа. С помощью асинхронного обмена сообщениями приложения могут отправлять запрос и обрабатывать другой запрос вместо ожидания ответа. Это становится понятным при повторном рассмотрении аналогии бармена. Бармены - сложные люди, которые могут обслуживать нескольких клиентов и чередовать выполнение нескольких задач одновременно. Они переходят от клиента к клиенту и обрабатывают несколько заказов, не блокируя и не ожидая ни одного клиента, поэтому каждый клиент остается довольным.

- Простота масштабирования. По мере роста приложений и предприятий способность динамически масштабироваться становится одним из важнейших преимуществ микросервисной архитектуры. Поскольку каждая служба небольшая и выполняет только одну задачу, каждая служба должна иметь возможность увеличиваться или уменьшаться по мере необходимости. Управление событиями и обмен сообщениями упрощают масштабирование микросервисов, поскольку они не связаны и не блокируются. Это также позволяет легко определить, какой сервис является узким местом, и добавлять дополнительные экземпляры только этого сервиса, вместо того, чтобы вслепую масштабировать все сервисы, что может быть в случае, когда микросервисы связаны между собой синхронной связью. Возможность масштабирования с

использованием обмена сообщениями была доказана такими компаниями, как Linkedin и Netflix.

Сравним очереди сообщений (MQ) и REST API по основным критериям оценки ПО в таблице 7.

Таблица 7

Сравнение очереди сообщений и REST API по основным критериям оценки ПО

Критерий	Message Queue	REST API
Использование (Usage)	Простой, но мощный, доступен почти на любом языке программирования и поддерживается всеми платформами	Может взаимодействовать с любым языком программирования и платформой
Производительность (Performance)	Работает с помощью асинхронного обмена сообщениями, приложения могут отправлять запрос и обрабатывать другой запрос вместо ожидания ответа, это дает вашему приложению высокую производительность	Работает синхронно, то есть ваш REST сервис блокируется в ожидании ответа. Это снижает производительность приложения, поскольку этот поток может обрабатывать другие запросы
Маштабируемость (Scalability)	Упрощает масштабирование микросервисов, так как может работать с неограниченным количеством экземпляров без каких либо дополнительных доработок. Очередь может работать как с одним получателем, так и с множеством.	Процесс масштабирования требует дополнительного внедрения балансировщика нагрузки, для раскидывания тасков среди действующих экземпляров системы
Отказоустойчивость (Reliability)	Предоставляет высокую отказоустойчивость. Если одна часть системы недоступна, другая может продолжать взаимодействовать с	Имеет меньшую отказоустойчивость, так как требует правильной конфигурации балансировщика нагрузки, который будет

	очередью. Очереди делают ваши данные постоянными и уменьшают количество ошибок, возникающих при отключении различных частей вашей системы.	отслеживать работоспособность каждого экземпляра. Если балансировщик выйдет из строя, система может полностью отвалиться.
Связи (Coupling)	Имеет слабую связь, которая позволяет микросервисам быть готовыми к бесконечным изменениям. Очередь уведомляет о новых сообщениях, ждет ответ и возвращает его. Она ничего не знает о содержимом и ей не важно от кого и кому было отправлено сообщение.	Имеет тесную связь, так как всегда существует некоторая зависимость сервисов от интерфейса. При вызове REST сервис имеет понятие кто его вызывает и кому нужно отправить данные, для того чтобы отправить все данные верно. Любые кардинальные изменения в сторонних сервисах, обычно, требуют доработки на REST сервисе.
Безопасность (Security)	Использует протокол защиты сообщений AMS, без каких либо дополнительных изменений на стороне приложения	Может быть использован протокол TLS, но его интеграция требует внесение изменений в приложениях получателя и отправителя, что может вызвать дополнительные трудности.

На основе данных, рассмотренных в таблице 4, можно сделать вывод, что разработчики и архитекторы должны чаще рассматривать системы очередей сообщений при построении микросервисной архитектуры, поскольку они обеспечивают основу для создания потрясающих систем и приложений.

4.2. Обзор фреймворков передачи данных Apache Kafka и RabbitMQ и их сравнение. Описание принципа работы брокера сообщений RabbitMQ

Apache Kafka и RabbitMQ - это две системы обмена сообщениями на основе подписок с открытым исходным кодом, легко внедряемые в проекты. RabbitMQ - более старый инструмент, выпущенный в 2007 году, который был основным компонентом в системах обмена сообщениями и SOA. Сегодня он также используется для потоковых сценариев использования. Kafka - это более новый инструмент, выпущенный в 2011 году, который с самого начала создавался для потоковых сценариев.

RabbitMQ - это брокер сообщений общего назначения, поддерживающий такие протоколы, как MQTT, AMQP и STOMP. Он может работать с высокопроизводительными сценариями использования, такими как обработка онлайн-платежей. Он может обрабатывать фоновые задания или выступать в роли посредника сообщений между микросервисами.

Kafka - это шина сообщений, разработанная для воспроизведения входных данных и потоков. Kafka - это надежный брокер сообщений, который позволяет приложениям обрабатывать, сохранять и повторно обрабатывать потоковые данные. Кафка имеет простой подход к маршрутизации, который использует ключ маршрутизации для отправки сообщений в тему.

В таблице 8 представлен сравнительный анализ брокер сообщений Kafka и RabbitMQ с учетом основных критериях проектирования системы

Таблица 8

Сравнительный анализ брокер сообщений Kafka и RabbitMQ с учетом основных критериях проектирования системы

	Kafka	RabbitMQ
Высокая доступность (High Availability)	Предоставляет с помощью Zookeeper для управления состоянием кластера.	Предоставляет кластеризацию и высокодоступные очереди
Распределение и параллелизм (Distribution and parallelism)	Способ распределения потребителей осуществляется по тематическим разделам, и каждый получатель из группы выделяется для одного раздела. Вы можете использовать механизм разделов для отправки каждому разделу различного набора сообщений по бизнес-ключу	Можно масштабировать количество получателей, это означает, что для каждого экземпляра очереди у вас будет много получателей, которые конкурируют за потребление сообщения. В этой форме работа по обработке сообщений распространяются всеми активными получателями, но все же сообщение

		может быть получено только один раз.
Производительность (Performance)	Предлагает гораздо более высокую производительность. Использует последовательный дисковый ввод-вывод для повышения производительности. Это может обеспечить высокую пропускную способность при ограниченных ресурсах, что необходимо для случаев использования больших данных.	Также имеет высокую производительность, которая позволяет обрабатывать миллион сообщений в секунду, но требует больше ресурсов
Резервное копирование (Replication)	По своему замыслу копирует брокер, и если мастер-брокер не работает, автоматически вся работа передается другому, у которого есть полная копия умершего, сообщение не теряется.	Очереди не реплицируются автоматически, это необходимо настроить. Если в вашем кластере есть как минимум три узла, все, что вам нужно сделать, это объявить свою очередь как очередь кворума, который позаботится о резервном копировании
Множественные подписки	На сообщения могут быть подписаны несколько получателей, то есть сообщение может быть обработана одновременно несколькими получателями	Сообщения могут быть направлены в многочисленные очереди. В каждой очереди только один получатель из этой очереди может обработать сообщение, но если сообщение попадает в несколько очередей, оно может быть обработано несколькими потребителями.
Порядок сообщений	Есть разделы, вы можете	Нет явного порядка

(Message ordering)	получать порядок сообщений в этом модуле. Сообщения направляются в темы по ключу сообщения, поэтому при выборе правильного ключа вы получите один раздел для каждого ключа с упорядоченными сообщениями.	сообщений. Можно реализовать с помощью определения множества очередей и отправляя каждое сообщение в другую очередь. Но, в масштабе, это решение может быть затруднительным.
Протоколы сообщений (Message protocols)	Поддерживает примитивы (int8, int16, int32, int64, string, arrays) и двоичные сообщения.	Поддерживает любые стандартные протоколы очереди, такие как AMQP, STOMP (на основе текста), MQTT (облегченный обмен сообщениями публикации / подписки) и HTTP
Время жизни сообщения (Message lifetime)	Представлен в виде журнала, сообщения всегда находятся там, вы можете контролировать это, определяя политику хранения сообщений.	Представлен в виде очереди, сообщения удаляются после использования и приходит подтверждение. В RabbitMQ вы можете настроить постоянные сообщения, пометить очередь как постоянную и сообщения как постоянные.
Гибкая маршрутизация (Flexible routing)	Сообщение отправляется в топик по ключу	Больше опций, например, с помощью регулярных выражений и подстановочных знаков, документы проверяются для получения дополнительной информации.
Приоритет сообщения (Message priority)	Отсутствует. Может быть реализовано с помощью клавиш сообщений, но в	Можно определить приоритеты сообщений, а также использованные сообщения с высоким

	больших масштабах это может быть трудно	приоритетом.
Мониторинг (Monitoring)	Можно использовать сторонние инструменты (Confluent, Landoop, Kafka Tool)	Есть встроенный интерфейс управления, доступный на выбранном порту. Также есть встроенная интеграции с популярными системами мониторинга Grafana и Prometeus
Подтверждение сообщения (Message acknowledgment)	В обеих платформах отправители получают подтверждение того, что сообщение поступает в очередь / топик, а также получатели отправляет подтверждение, когда сообщение успешно используется, так что можно быть уверенным, что сообщения не будут потеряны по пути.	

На основе этих данных мы можем определить сценарии, для которых лучше подходит Kafka и RabbitMQ.

Используйте Kafka в случаях если необходима:

- Много получателей на одно и то же сообщение;
- Высокая пропускная способность (миллионы сообщений в секунду);
- Потоковая обработка;
- Резервное копирование очередей;
- Высокая доступность;
- Важен порядок сообщений.

Используйте RabbitMQ в случаях если необходима:

- Один получатель на одно и тоже сообщение;
- Гибкая маршрутизация;
- Очередь приоритетов;
- Использование сообщений стандартного протокола

В заключении можно сказать, что RabbitMQ достаточно хорош для простых случаев использования очереди сообщений, с небольшим трафиком данных. Есть определенные преимущества, такие как приоритетная очередь и гибкие параметры маршрутизации. Но для массивных данных и высокой пропускной способности используйте Кафку без споров. Также, если вам нужен журнал логов или несколько получателей для одних и тех же сообщений, перейдите в Kafka, потому что RabbitMQ не сможет помочь с этим.

RabbitMQ - это бесплатное и расширяемое решение для организации очередей сообщений. Это брокер сообщений, который понимает AMQP (расширенный протокол очереди сообщений), но также может использоваться с другими популярными решениями обмена сообщениями, такими как MQTT. Он высокодоступен, отказоустойчив и масштабируем. RabbitMQ реализован в Erlang OTP, технологии, разработанной для построения стабильных, надежных, отказоустойчивых и хорошо масштабируемых систем, которые обладают встроенными возможностями обработки очень большого числа одновременных операций [27].

При проектировании веб-системы, мы можем представить RabbitMQ как уровень промежуточного программного обеспечения, который позволяет различным службам в вашем приложении взаимодействовать друг с другом, не беспокоясь о потере сообщений, обеспечивая при этом различные требования к качеству обслуживания. Он также обеспечивает тонкую и эффективную маршрутизацию сообщений, что обеспечивает широкое разделение приложений. RabbitMQ будет особенно полезен, когда нужна архитектура, которая является более гибкой и адаптируется к постоянно меняющимся потребностям многих микросервисов.

Суть работы RabbitMQ заключается а том, что он принимает и пересылает сообщения. Вы можете думать об этом как о почтовом отделении: когда вы помещаете почту, которую вы хотите опубликовать, в почтовый ящик, вы можете быть уверены, что почтальон в итоге доставят почту вашему получателю. В этой аналогии RabbitMQ - это почтовый ящик, почтовое отделение и почтальон.

Рассмотрим основные компоненты используемые при работе:

- *Producer* - программа которая отправляет сообщение
- *Queue* - это имя почтового ящика, который находится внутри RabbitMQ. Хотя сообщения проходят через RabbitMQ и ваши приложения, они могут храниться только в очереди. Очередь ограничена только пределами памяти хоста и диском, по сути, это большой буфер сообщений. Многие *Producers* могут отправлять сообщения, которые идут в одну очередь, и многие *Consumers* могут пытаться получать данные из одной очереди
- Consumer - программа которая ждет для получения сообщения из очереди

Основное различие между RabbitMQ и почтовым отделением состоит в том, что он не работает с бумагой, а принимает, хранит и пересылает двоичные двоичные объекты данных - сообщений.

4.3. Определение контейнеризация и преимущества ее использования. Микросервисы и контейнеризация. Контейнеры Docker

Процесс разделения части программы от монолита и добавление его в отдельный микросервис называется контейнеризацией. Это легковесная виртуализация и изоляция ресурсов на уровне операционной системы, которая позволяет запускать приложение и необходимый ему минимум системных библиотек в полностью стандартизованном контейнере.

Контейнеризация стала основной тенденцией в разработке программного обеспечения в качестве альтернативы или дополнения к виртуализации. Он включает в себя инкапсуляцию или упаковку программного кода и всех его зависимостей, чтобы он мог работать равномерно и согласованно в любой инфраструктуре. Технология быстро развивается, что дает ощутимые преимущества для разработчиков и рабочих групп, а также для всей инфраструктуры программного обеспечения.

Контейнеризация позволяет разработчикам создавать и развертывать приложения быстрее и безопаснее. С помощью традиционных методов код разрабатывается в конкретной вычислительной среде, которая при переносе на новое место часто приводит к ошибкам и ошибкам. Например, когда разработчик переносит код с настольного компьютера на виртуальную машину или из Linux в операционную систему Windows. Контейнеры устраняют эту проблему, объединяя код приложения вместе со связанными файлами конфигурации, библиотеками и зависимостями, необходимыми для его запуска. Этот единственный пакет программного обеспечения или «контейнер» абстрагируется от операционной системы хоста, и, следовательно, он сам по себе и становится переносимым - способным работать на любой платформе или в облаке без проблем.

Контейнеры часто называют «легковесными», то есть они совместно используют ядро операционной системы компьютера и не требуют дополнительных затрат на ассоциирование операционной системы в каждом приложении. Емкости по своей сути меньше, чем у виртуальной машины, и требуют меньше времени запуска, что позволяет гораздо большему количеству контейнеров работать на той же вычислительной мощности, что и у одной виртуальной машины. Это повышает эффективность работы сервера и, в свою очередь, снижает затраты на сервер и лицензирование.

Проще говоря, контейнеризация позволяет писать приложения один раз и запускать где угодно. Эта мобильность важна с точки зрения процесса разработки и совместимости с поставщиком. Он также предлагает другие заметные преимущества, такие как изоляция отказов, простота управления и безопасность, и многие другие [29].

Контейнеры инкапсулируют приложение как единый исполняемый пакет программного обеспечения, который связывает код приложения вместе со всеми соответствующими файлами конфигурации, библиотеками и зависимостями, необходимыми для его запуска. Контейнерные приложения «изолированы» в том смысле, что они не объединяются в копию операционной системы. Вместо этого в операционную систему хоста устанавливается движок с открытым исходным кодом (такой как движок Docker), который становится каналом для контейнеров, чтобы совместно использовать операционную систему с другими контейнерами в той же вычислительной системе.

Контейнеризация позволяет разработчикам создавать и развертывать приложения быстрее и безопаснее, независимо от того, является ли приложение традиционным монолитом (одноуровневое программное приложение) или модульным микросервисом (набором слабо связанных служб). Новые облачные приложения могут создаваться с нуля как контейнерные микросервисы, разбивая сложное приложение на ряд небольших специализированных и управляемых сервисов. Существующие приложения могут быть переупакованы в контейнеры и контейнерные микросервисы, которые используют вычислительные ресурсы более эффективно [30].

Контейнеризация предлагает значительные преимущества для разработчиков и групп разработчиков. Рассмотрим основные преимущества:

- Переносимость. Контейнер создает исполняемый пакет программного обеспечения, который абстрагируется, не привязан к операционной системе хоста или не зависит от нее. Поэтому контейнер является переносимым и способен работать равномерно и согласованно на любой платформе или в облаке.

- Гибкость. Docker Engine для запуска контейнеров начал промышленный стандарт для контейнеров с простыми инструментами разработчика и универсальным подходом к пакетированию, который работает как в Linux, так и в Windows. Контейнерная экосистема перешла на механизмы, управляемые Инициативой открытых контейнеров (OCI). Разработчики программного обеспечения могут продолжать использовать гибкие инструменты или процессы DevOps для быстрой разработки и улучшения приложений.

- Скорость. Контейнеры часто называют «облегченными», то есть они совместно используют ядро операционной системы (ОС) компьютера и не перегружены этими дополнительными издержками. Это не только повышает эффективность работы серверов, но и снижает затраты на сервер и лицензирование, а также ускоряет запуск, поскольку операционная система не загружается.

- Изолирование ошибок. Каждое контейнерное приложение изолированно и работает независимо от других. Отказ одного контейнера не влияет на дальнейшую работу любых других контейнеров. Группы разработчиков могут выявлять и исправлять любые технические проблемы в одном контейнере без каких-либо простоев в других контейнерах. Кроме того, механизм контейнеров может использовать любые методы изоляции безопасности ОС, такие как управление доступом SELinux, для изоляции сбоев в контейнерах.

- Эффективность. Программное обеспечение, работающее в контейнерах, совместно использует ядро ОС компьютера, а уровни приложений в контейнере могут совместно использоваться контейнерами. Таким образом, контейнеры по своей природе имеют меньшую емкость, чем виртуальная машина, и требуют меньше времени запуска, что позволяет гораздо большему количеству контейнеров работать на той же вычислительной мощности, что и одна виртуальная машина. Это повышает эффективность работы сервера, снижает затраты на сервер и лицензирование.

- Простота управления. Платформа оркестрации контейнеров автоматизирует установку, масштабирование и управление контейнерными рабочими нагрузками и сервисами. Платформы оркестровки контейнеров могут упростить задачи управления, такие как масштабирование контейнерных приложений, развертывание новых версий приложений, а также обеспечение мониторинга, ведения журнала и отладки, среди прочих функций. Kubernetes - самая популярная доступная система контейнерной оркестровки. Это технология с открытым исходным кодом (изначально с открытым исходным кодом от Google, основанная на их внутреннем проекте под названием Borg), которая изначально автоматизирует функции контейнеров Linux. Kubernetes работает со многими механизмами контейнеров, такими как Docker, но он также работает с любой системой контейнеров, которая соответствует стандартам Open Container Initiative (OCI) для форматов изображений контейнеров и сред выполнения.

- Безопасность. Изоляция приложений как контейнеров по своей сути предотвращает вторжение вредоносного кода в другие контейнеры или хост-систему. Кроме того, могут быть определены разрешения безопасности для автоматической блокировки нежелательных компонентов от входа в контейнеры или ограничения связи с ненужными ресурсами.

Разработчики программного обеспечения рассматривают микросервисы как превосходный подход к разработке приложений и управлению ими по сравнению с более ранней монолитной моделью, в которой программное приложение сочеталось с соответствующим пользовательским интерфейсом и базовой базой данных в одном устройстве на одной серверной платформе. С помощью микросервисов сложное приложение разбивается на ряд более мелких, более специализированных сервисов, каждый со своей базой данных и собственной бизнес-логикой. Микросервисы затем взаимодействуют друг с другом через общие интерфейсы (например, API) и интерфейсы REST (например, HTTP). Используя микросервисы, группы разработчиков могут сосредоточиться на обновлении определенных областей приложения, не затрагивая его в целом, что приводит к ускорению разработки, тестирования и развертывания.

Концепции, лежащие в основе микросервисов и контейнеризации, схожи, поскольку обе они представляют собой методы разработки программного обеспечения, которые по существу превращают приложения в наборы небольших сервисов или компонентов, которые являются переносимыми, масштабируемыми, эффективными и простыми в управлении.

Более того, микросервисы и контейнеризация хорошо работают вместе. Контейнеры обеспечивают легкую инкапсуляцию любого приложения, будь то традиционный монолит или модульный микросервис. Микросервис, разработанный в контейнере, затем получает все присущие контейнеру преимущества - мобильность с точки зрения процесса разработки и совместимости с поставщиком (без привязки к поставщику), а также гибкость разработчика, изоляция сбоев, эффективность работы серверов, автоматизация установка, масштабирование и управление, а также уровни безопасности, среди прочего.

Контейнеры, микросервисы и облачные вычисления работают вместе, чтобы вывести разработку и доставку приложений на новый уровень, что невозможно при традиционных методологиях и средах. Эти подходы следующего поколения добавляют гибкости, эффективности, надёжности и безопасности к жизненному циклу разработки программного обеспечения - все это приводит к более быстрой доставке приложений и усовершенствований для конечных пользователей и рынка.

Концепция контейнеризации и изоляции процессов давно существует, но появление в 2013 году Docker Engine с открытым исходным кодом, отраслевого стандарта для контейнеров с простыми инструментами разработчика и универсальным подходом к упаковке, ускорило внедрение этой технологии. Последние исследования прогнозирует, что более 50% компаний будут использовать контейнерные технологии к 2020 году [28].

Когда мы говорим о контейнерах, прежде всего мы подразумеваем Docker - open-source-технологию, применяемое для разработки, тестирования, доставки и запуска веб-приложений в средах с поддержкой контейнеризации. Он нужен для более эффективного использование системы и ресурсов,

быстрого развертывания готовых программных продуктов, а также для их масштабирования и переноса в другие среды с гарантированным сохранением стабильной работы. Благодаря своей популярности данная технология стала синонимом слова «контейнер».

Запускаясь Docker делает всего несколько системных вызовов и ядро создает для нового процесса отдельное пространство, отдельную виртуальную сеть, отдельный набор ограничений по ресурсам. Процесс "запущенный в docker", на самом деле находится не в какой-то виртуальной машине (нет никакого эмулятора настоящей машины, никакой виртуальной сущности), он запущен на той же машине, тем же ядром.

Определим основные функции и свойства Docker контейнера:

- Имеет в себе содержимое - что-то находится внутри контейнера или вне контейнера.
- Портативный. Контейнер Docker можно использовать на локальном компьютере, компьютере вашего коллеги или на серверах облачного провайдера. Вроде как та коробочка детских безделушек, которую ты продолжаешь перевозить с собой из дома в дом.
- Имеет понятные интерфейсы для доступа. Контейнер Docker имеет несколько механизмов взаимодействия с внешним миром. Он имеет порты, которые могут быть открыты для взаимодействия через браузер. Вы можете настроить его для взаимодействия с данными через командную строку.
- Может быть получен из удаленного доступа. Контейнер Docker хранит образ (image), похожее на форму вашего контейнера. Затем, когда вам нужен контейнер, вы можете сделать его из образа. Образ содержит Dockerfile с инструкциями для нового образа, образ операционной системы, библиотеки и кодовую базу, необходимые для запуска вашего приложения.

Использование Docker контейнеров позволяет осуществлять быструю и последовательную доставку нужных приложений. Docker оптимизирует жизненный цикл разработки, позволяя разработчикам работать в стандартизированных средах, используя локальные контейнеры, которые предоставляют ваши приложения и услуги. Контейнеры отлично подходят для непрерывной интеграции и рабочих процессов с непрерывной доставкой (CI / CD).

Разработчики могут писать код локально и делиться своей работой со своими коллегами, используя контейнеры Docker. Они используют Docker, чтобы внедрить свои приложения в тестовую среду и выполнить автоматические и ручные тесты. Когда разработчики находят ошибки, они могут исправлять их в среде разработки и повторно развертывать их в тестовой среде для тестирования и проверки. После завершения тестирования получить

исправление для клиента так же просто, как отправить обновленный образ в производственную среду.

Контейнерная платформа Docker обеспечивает высокую переносимость рабочих нагрузок. Контейнеры Docker могут работать на локальном ноутбуке разработчика, на физических или виртуальных машинах в центре обработки данных, в облачных провайдерах или в различных средах. Портативность и легковесность Docker также упрощают динамическое управление рабочими нагрузками, масштабирование или разрушение приложений и служб в соответствии с потребностями бизнеса практически в реальном времени.

Docker использует архитектуру клиент-сервер. Клиент Docker общается с демоном Docker, который выполняет тяжелую работу по сборке, запуску и распространению ваших контейнеров Docker. Docker-клиент и демон могут работать в одной системе или вы можете подключить Docker-клиент к удаленному Docker-демону. Клиент Docker и демон взаимодействуют с помощью REST API, через сокеты UNIX или сетевой интерфейс.

Docker-демон (Docker daemon) прослушивает запросы Docker API и управляет объектами Docker, такими как изображения, контейнеры, сети и тома. Демон также может взаимодействовать с другими демонами для управления сервисами Docker.

Docker-клиент (Docker client) - это основной способ взаимодействия многих пользователей Docker с Docker. Когда вы используете такие команды, как docker run, клиент отправляет эти команды в Docker daemon, который выполняет их. Команда docker использует Docker API. Клиент Docker может общаться с несколькими демонами.

В реестре Docker (Docker registry) хранятся изображения Docker. Docker Hub - это общедоступный реестр, который может использовать каждый, и Docker по умолчанию настроен на поиск изображений в Docker Hub. Вы даже можете запустить свой собственный личный реестр. Если вы используете Docker Datacenter (DDC), он включает Docker Trusted Registry (DTR). Когда вы используете команды Docker Pull или Docker Run, необходимые изображения извлекаются из вашего сконфигурированного реестра. Когда вы используете команду docker push, ваш образ помещается в ваш настроенный реестр.

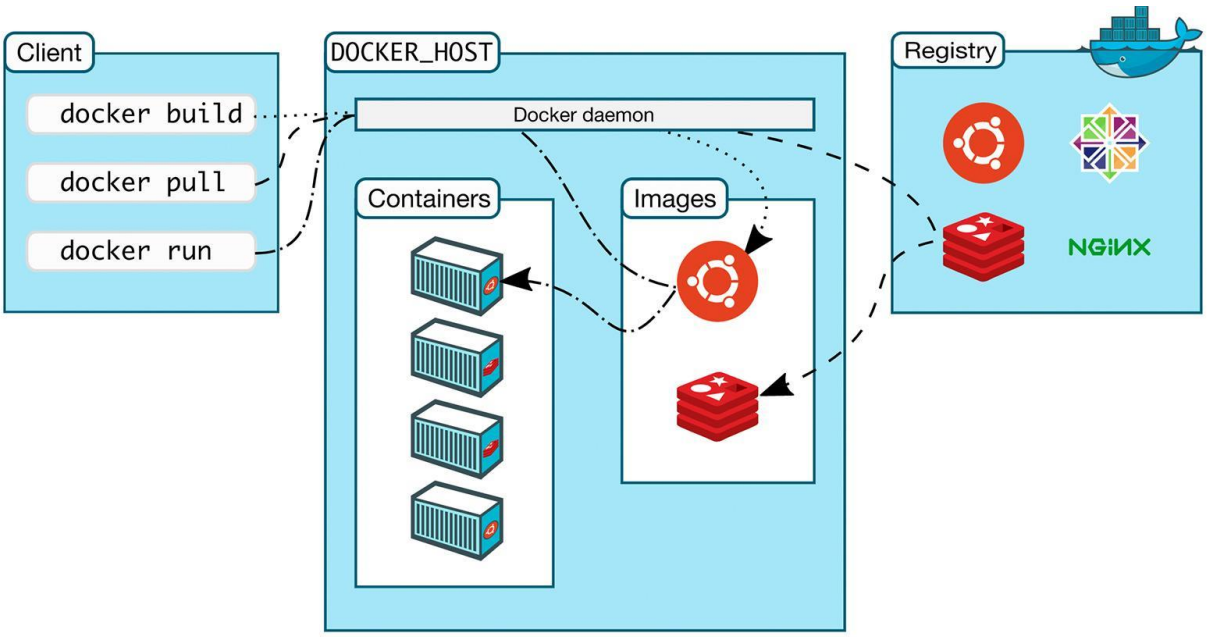

Рисунок 2. Схема принципа работы архитектуры Docker

Docker легкий и быстрый. Он обеспечивает жизнеспособную, экономически выгодную альтернативу виртуальным машинам, поэтому вы можете использовать больше вычислительных мощностей для достижения своих бизнес-целей. Docker идеально подходит для сред с высокой плотностью и для малых и средних развертываний, где вам нужно делать больше с меньшими ресурсами.

4.4. Определение оркестрация контейнеров, преимущества и принцип работы. Сравнение и обзор Kubernetes, Docker Swarm и Apache Mesos

Эволюция технологий изменила способ построения архитектуры приложений. Docker, облачные сервисы и сервисы оркестрации контейнеров предоставили нам возможность разрабатывать распределенные, более масштабируемые и надежные решения.

Оркестрация контейнеров автоматизирует развертывание, управление, масштабирование и создание сетей контейнеров. Предприятия, которым необходимо развернуть и управлять сотнями или тысячами контейнеров и хостов Linux, могут воспользоваться преимуществами оркестровки контейнеров.

Оркестрация контейнеров может использоваться в любой среде, где вы используете контейнеры. Это может помочь вам развернуть одно и то же приложение в разных средах без необходимости его перепроектирования. А микросервисы в контейнерах упрощают организацию сервисов, в том числе хранилищ, сетей и безопасности.

Рассмотрим задачи, которые могут быть решены с помощью оркестрации контейнеров:

- Поставка и развертывание
- Конфигурация и управление приложением
- Распределение ресурсов
- Масштабирование или удаление контейнеров на основе балансировки рабочих нагрузок в инфраструктуре приложения
- Балансировка нагрузки и маршрутизация трафика
- Мониторинг состояния контейнера
- Настройка приложений на основе контейнера, в котором они будут работать
- Поддержание взаимодействия между контейнерами

Когда вы используете инструмент оркестрации контейнера, такой как Kubernetes или Docker Swarm, вы обычно описываете конфигурацию своего приложения в файле YAML или JSON, в зависимости от инструмента оркестровки. В этих файлах конфигурации (например, docker-compose.yml) вы указываете инструменту оркестровки, где собирать образы контейнеров (например, из Docker Hub), как устанавливать сеть между контейнерами, как монтировать тома хранения и где хранить журналы для этого контейнера. Как правило, группы разветвляют и управляют версиями этих файлов конфигурации, чтобы они могли развертывать одни и те же приложения в разных средах разработки и тестирования, прежде чем развертывать их в рабочих кластерах.

Контейнеры развертываются на хостах, обычно в реплицируемых группах. Когда пришло время развертывать новый контейнер в кластере, инструмент управления контейнером планирует развертывание и ищет наиболее подходящий хост для размещения контейнера на основе предопределенных ограничений (например, доступность ЦП или памяти). Вы даже можете размещать контейнеры в соответствии с метками или метаданными или в соответствии с их близостью по отношению к другим хостам - можно использовать все виды ограничений.

Когда контейнер запущен на хосте, инструмент оркестровки управляет своим жизненным циклом в соответствии со спецификациями, изложенными в файле определения контейнера (например, его Dockerfile). Основное удобство инструментов оркестровки контейнеров в том, что вы можете использовать их в любой среде, в которой вы можете запускать контейнеры [30]. В настоящее время контейнеры поддерживаются практически в любой среде, от традиционных локальных серверов до экземпляров общедоступного облака

Существуют различные платформы для оркестрации контейнеров. Они позволяют реализовать удобные и эффективные средства развертывания контейнерных систем, построения единой централизованной консоли для управления. Сегодня таких платформ очень много, они все больше набирают интерес и развитие в среде разработки высоконагруженных систем. Наиболее известны следующие системы: Kubernetes, Docker Swarm и Apache Mesos.

Kubernetes — OpenSource-система для управления контейнерными кластерами. Благодаря тому, что Google открыла код и сделала свою систему оркестрации открытой, она стала самой популярной. Сегодня многие рассматривают ее как лучшее и универсальное решение для работы с приложениями в облачной среде.

Первоначально разработанный Google как ответвление от его проекта Borg, Kubernetes зарекомендовал себя как фактический стандарт для оркестровки контейнеров. Это флагманский проект Cloud Native Computing Foundation, который поддерживается такими ключевыми игроками, как Google, Amazon Web Services (AWS), Microsoft, IBM, Intel, Cisco и RedHat.

Рассмотрим основные компоненты архитектуры Kubernetes:

- *Cluster*. Кластер - это набор узлов с по крайней мере одним главным узлом и несколькими рабочими узлами (иногда называемыми миньонами), которые могут быть виртуальными или физическими машинами.

- *Kubernetes master*. Мастер управляет планированием и развертыванием экземпляров приложений на узлах, а полный набор сервисов, которые запускает мастер-узел, называется плоскостью управления. Мастер связывается с узлами через сервер API Kubernetes. Планировщик назначает узлы модулям (одному или нескольким контейнерам) в зависимости от ограничений ресурса и политики, которые вы определили.

- *Kubelet*. Каждый узел Kubernetes запускает процесс агента, называемый kubelet, который отвечает за управление состоянием узла: запуск, остановка и обслуживание контейнеров приложений на основе инструкций из плоскости управления. Kublet получает всю свою информацию от сервера API Kubernetes.

- *Pods*. Базовый блок планирования, который состоит из одного или нескольких контейнеров, которые гарантированно размещаются на хост-машине и могут совместно использовать ресурсы. Каждому модулю назначается уникальный IP-адрес в кластере, что позволяет приложению использовать порты без конфликтов. Вы описываете желаемое состояние контейнеров в модуле через объект YAML или JSON, который называется PodSpec. Эти объекты передаются в кубеле через сервер API

- *Развертывания, реплики и наборы реплик*. Развертывание - это объект YAML, который определяет модули и количество экземпляров контейнера, называемых репликами, для каждого модуля. Вы определяете количество реплик, которые вы хотите запустить в кластере, с помощью ReplicaSet, который является частью объекта развертывания. Так, например, если узел, на котором запущен модуль, умирает, набор реплик будет гарантировать, что другой модуль запланирован на другом

Docker Swarm — вторая по популярности система оркестрации. Инструмент контейнерной кластеризации Docker Swarm появился немного позже и стал частью платформы Docker. Он позволяет объединять Docker-хосты в общий виртуальный хост. Docker Swarm интересна, прежде всего, малым и средним предприятиям.

Несмотря на то, что Docker полностью принял Kubernetes в качестве предпочтительного механизма управления контейнером, компания по-прежнему предлагает Swarm, свой собственный полностью интегрированный инструмент управления контейнером. Немного менее расширяемый и сложный, чем Kubernetes, это хороший выбор для энтузиастов Docker, которые хотят более простой и быстрый путь к развертыванию контейнеров. Фактически, Docker объединяет Swarm и Kubernetes в своей корпоративной версии в надежде сделать их дополнительными инструментами.

Рассмотрим основные компоненты архитектуры Docker Swarm:

- *Swarm*. Как и кластер в Kubernetes, swarm - это набор узлов, по крайней мере, с одним главным узлом и несколькими рабочими узлами, которые могут быть виртуальными или физическими машинами.

- *Service* - это задачи, которые узлы менеджера или агента должны выполнять в рое, как это определено администратором роя. Служба определяет, какие образы контейнеров должен использовать рой и какие команды рой будет запускать в каждом контейнере. Служба в этом контексте аналогична микросервису; например, именно здесь вы определяете параметры конфигурации для веб-сервера nginx, работающего в вашем рое. Вы также определяете параметры для реплик в определении сервиса.

- *Диспетчер узла*. При развертывании приложения в рое узел менеджера предоставляет несколько функций: он доставляет работу (в форме задач) на рабочие узлы, а также управляет состоянием роя, к которому он принадлежит. Узел менеджера может запускать те же рабочие узлы служб, но вы также можете настроить их для запуска только служб, связанных с узлом менеджера.

- *Рабочие узлы*. Эти узлы запускают задачи, распределенные узлом менеджера в рое. На каждом рабочем узле запускается агент, который сообщает главному узлу о состоянии назначенных ему задач, поэтому узел менеджера может отслеживать службы и задачи, выполняемые в swarm.

- *Task* - это контейнеры Docker, которые выполняют команды, определенные вами в службе. Узлы диспетчера назначают задачи рабочим узлам, и после этого назначения задача не может быть перемещена другому рабочему. Если задача не выполняется в наборе реплик, менеджер назначит новую версию этой задачи другому доступному узлу в рое.

На третьем месте по популярности система Apache Mesos. Она объединяет существующие объекты в единый виртуальный ресурс, формируя крупные кластеры и эффективную систему управления серверной инфраструктурой, в которой каждому кластеру выделяется свой индивидуальный пул ресурсов.

Apache Mesos, немного старше, чем Kubernetes, - проект программного обеспечения с открытым исходным кодом, первоначально разработанный в Калифорнийском университете в Беркли, но в настоящее время широко используемый в таких организациях, как Twitter, Uber и Paypal. Облегченный интерфейс Mesos позволяет легко масштабировать до 10 000 узлов (или более) и позволяет независимо развивающимся средам, работающим на его основе, развиваться.

Рассмотрим основные компоненты архитектуры Apache Mesos:

- *Master daemon*. Часть главного узла, которая управляет демонами агентов. С помощью Apache Zookeeper вы можете создать главный кворум Mesos, состоящий как минимум из трех основных узлов, для целей высокой доступности.

- *Agent daemon*. Другая часть главного узла, которая выполняет задачи, отправленные платформой.

- *Framework*. Mesos не запускает рабочие нагрузки для оркестровки приложений; вместо этого Marathon получает ресурсы от мастера Mesos (в форме предложений), а Marathon отправляет задачи на основе предложений ресурсов исполнителям, которые запускают задачи на агентах.

- *Offer*. Мастер Mesos собирает информацию о ЦП и доступности памяти узлов агента и отправляет эту информацию в Marathon, чтобы Marathon знал, какие ресурсы доступны.

- *Task*. Это основные единицы работы, которые Marathon планирует на основе предложений ресурсов от мастера Mesos. Задачи выполняются исполнителями на агентских узлах.

В заключении можно сказать, что оркестрация контейнеров является очень важным и полезным инструментом, который позволяет создавать информационные системы из множества контейнеров, каждый из которых отвечает только за одну определенную задачу, а общение осуществляется через сетевые порты и общие каталоги. При необходимости каждый такой контейнер можно заменить другим, что позволяет, например, быстро перейти на другую версию базы данных при необходимости.

Глава 5. Построение микросервисной архитектуры на базе системы оптимизации логистики

Результатом данной работы является проектирование масштабируемой архитектуры для системы оптимизации транспортной логистики.

В ходе работы были достигнуты следующие результаты:

- Поднят микросервис планирования с использованием передовых технологий (RabbitMQ, Docker, Kubernetes), который работает независимо от остальной части системы, что делает его более гибким и доступным
- Полностью разделен процесс развертывания, тестирования и разработки
- Реализована возможность управлять и масштабировать микросервис независимо от других частей системы
- Для планирования маршрутов внедрен инструмент Jsprit, который сделал процесс планирования быстрее и качественнее.

Основные компоненты микросервсиной архитектуры системы оптимизации транспортной логистики Relog описаны в таблице 9.

Таблица 9

Основные компоненты микросервсиной архитектуры системы оптимизации транспортной логистики

Компонент	Описание	Технология
App Relog - основное приложение системы (монолит)	Панель управления диспетчера, которая позволяет управлять заказами и курьерами,	NodeJS приложение, с использованием Meteor, React и Redux

	смотреть всю историю передвижения курьеров, отслеживать в режиме реального времени изменения в статусах заказов и многое другое.	
Relog Algo - микро - сервис построения оптимального маршрута	Микросервис ищет оптимальный маршрут с помощью алгоритма, который решает задачу маршрутизации транспортных средств. За определенное количество времени приложение производит необходимое количество итераций и возвращает наилучший ответ.	Java приложение, с использованием библиотеки Jsprit
Очередь алгоритма	Очередь содержит в себе сообщения с данными планирования (водители. заявки, временные окна и т.д), в виде сообщения в формате json	RabbitMQ
Relog Router	Микросервис строит матрицу дистанций, необходимую для расчета дистанций между точками и взаимодействует с микросервисом Relog Algo, посредством передачи данных с помощью REST API	OSRM Router API
Контейнеры	Контейнеры содержат в себе поднятые экземпляры микросервисов	Docker
Система оркестрации	Автоматизирует развертывание, управление, масштабирование и создание контейнеров, а также их мониторинг и логирование	Kubernetes

В итоге получилась полноценная микрсервисная архитектура с использованием передовых технологий (RabbitMQ, Docker, Kubernetes), каждая часть которой работает независимо друг от друга, что делает его более гибким и доступным.

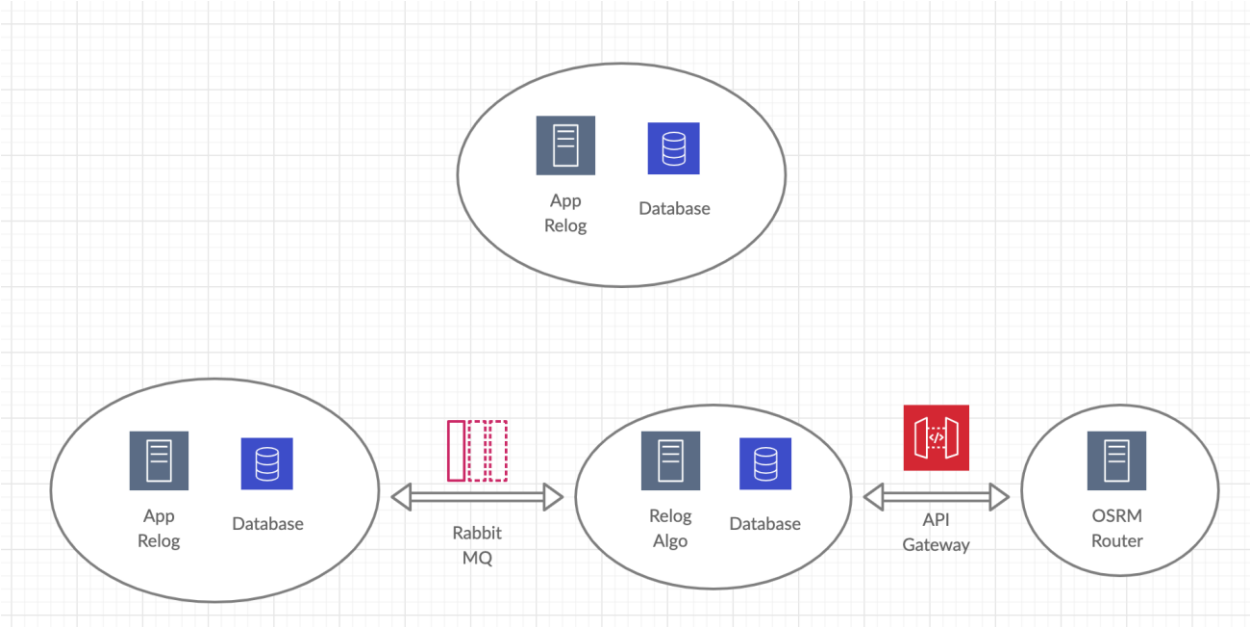

Рисунок 3. Схемы архитектуры системы планирования маршрутов

Рассмотрим основной принцип работы системы на примере реального планирования маршрута, проведенного в системе App Relog.

Пользователь выбирает необходимые заявки из списка всех заявкой, которые есть в системе.

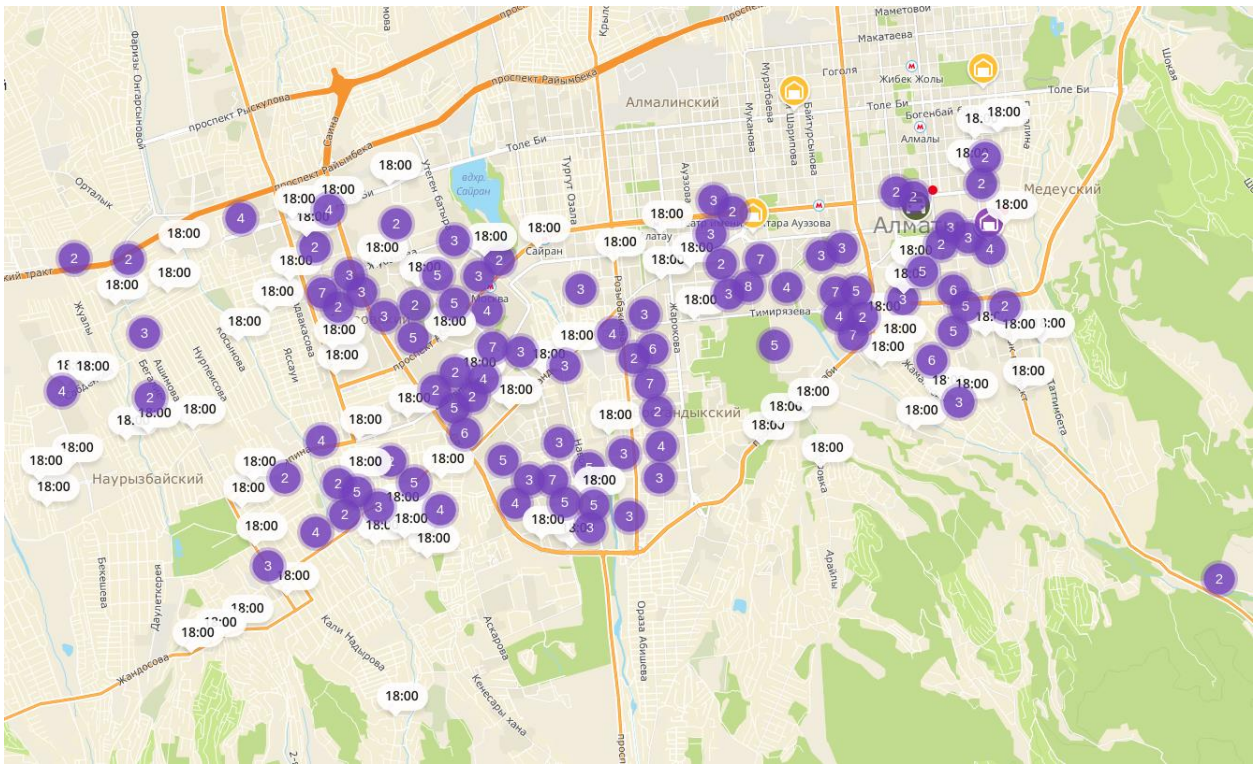

Рисунок 4. Страницы выбора заявок в приложении App Relog

Далее пользователь указывает параметры маршрутизации и нажимает кнопку применить.

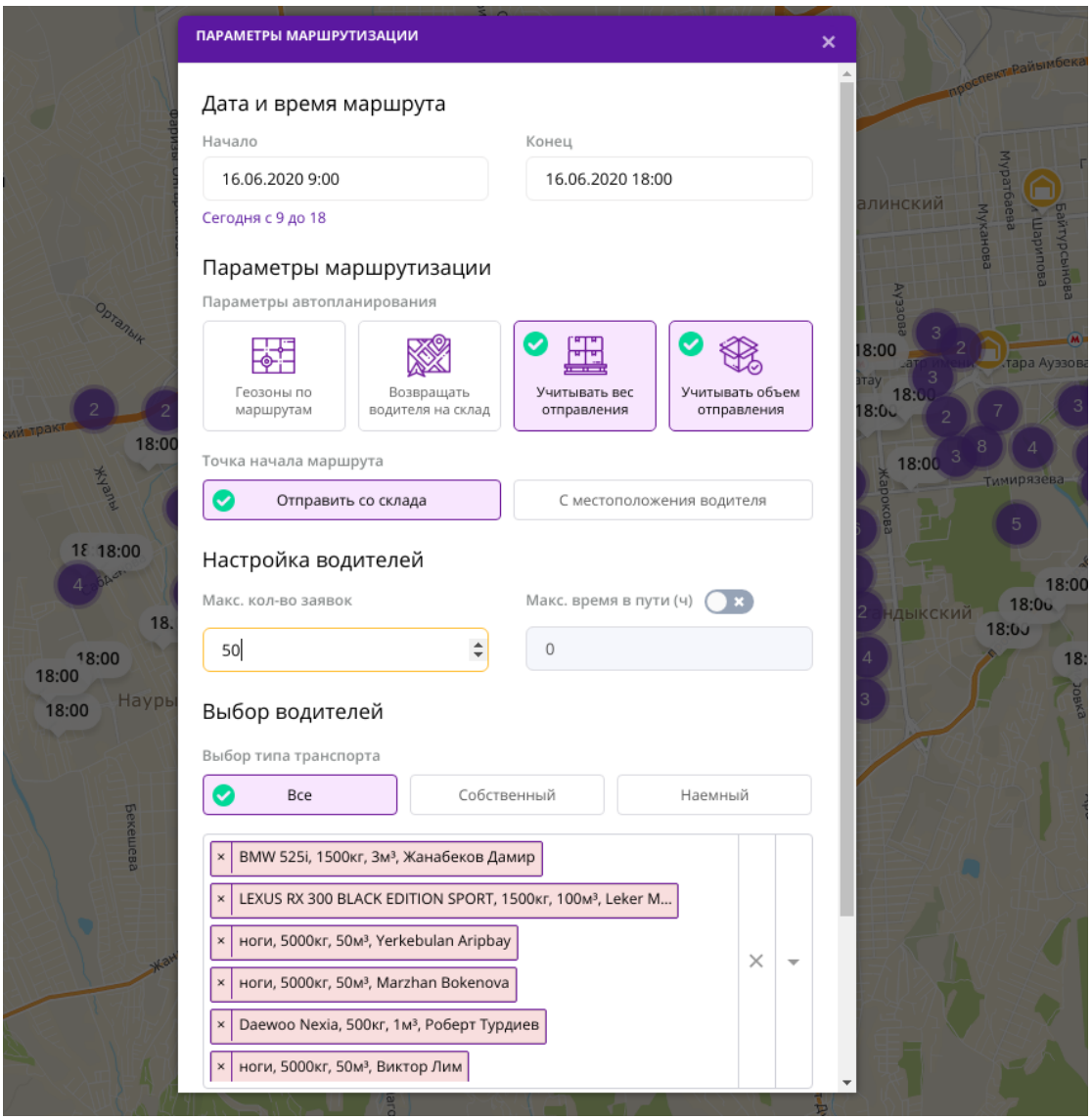

Рисунок 5. Окно параметров маршрута в приложении App Relog

После этого данные планирования отправляется в очередь rabbitmq в виде сообщения в формате json

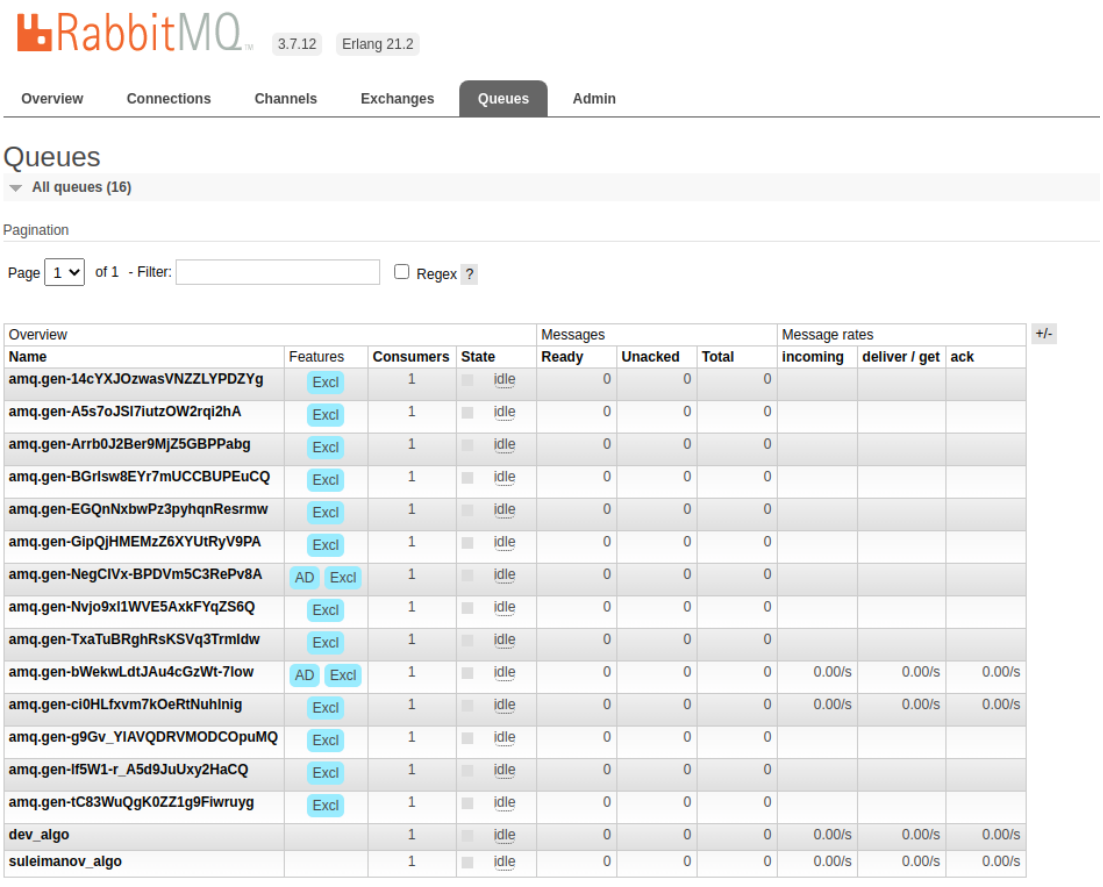

Рисунок 6. Список очереди сообщений в RabbitMQ

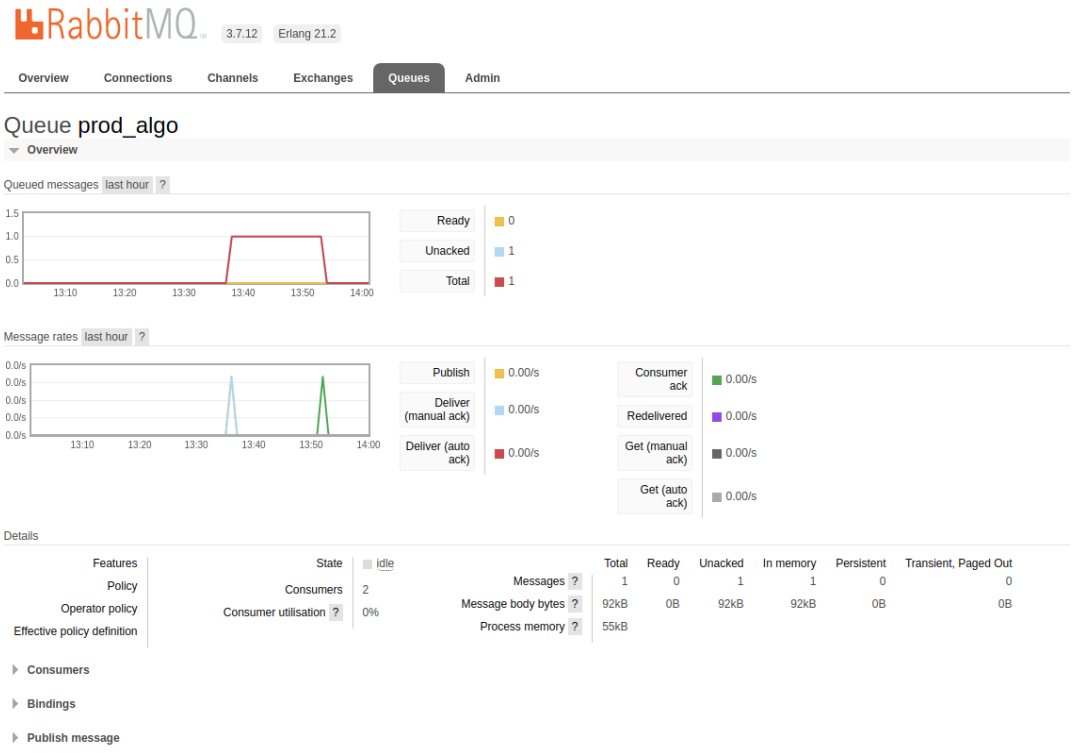

Рисунок 7. Информация о статусе сообщения в RabbitMQ

Запущенный алгоритм которые случшает очередь на новые сообщения, имеет ограничение, по которому 1 экземпляр алгоритма может обработать только 1 сообщения, остальные находятся в очереди и ждут.

Если экземпляр алгоритма свободен он берет сообщение из очереди в работу, в RabbitMQ данное сообщение получает статус в процессе и передает данные интерфейсу о статусе через канал, который реализован для получения прогресса планирования

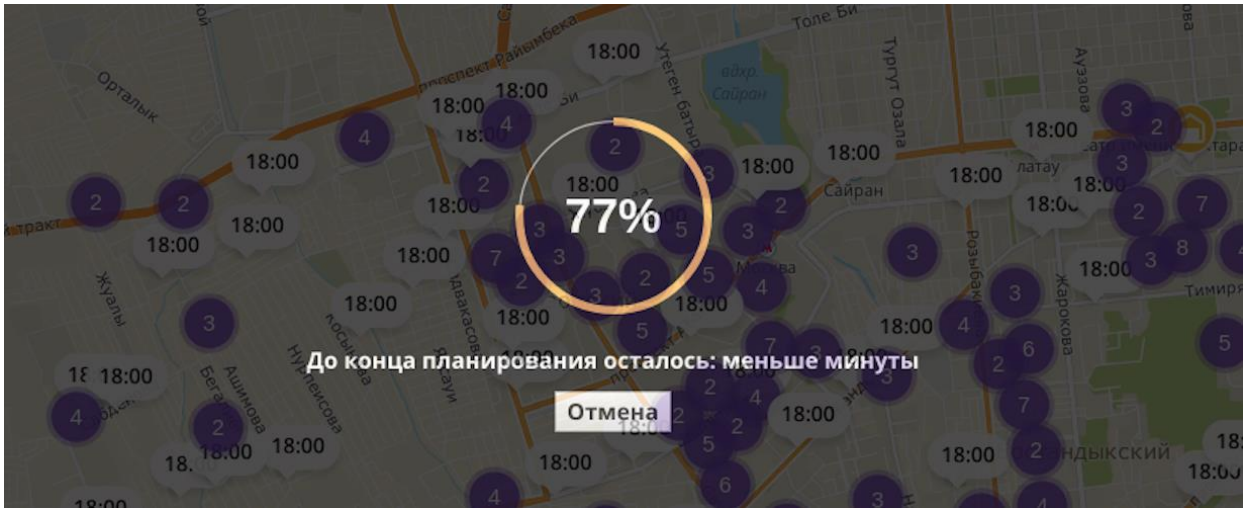

Рисунок 8. Статус работы планирования в приложении App Relog

Как только данные из очереди попадают в микросервис планирования Relog Algo, система производит следующие действия:

- Микросервис планирования преобразует json сообщение в полноразмерные объекты, необходимые для расчетов
- Собирает в одну кучу координаты точек и машин и отправляет в отдельный микросервис Router (OSRM) через API, который строит матрицу дистанций, необходимую для расчета дистанций между точек
- Запускается библиотека Jsprit, который начинает производить итерация по поиску оптимального маршрута с ограничением по времени (30 минут на поиск маршрута) и кол-во итераций (максимум 50 итераций)
- Лучший вариант маршрута, полученный после работы библиотеки отправляем обратным ответом в клиентский сервис в виде сообщения в формате json через Rabbit
- Полученные данные записываются в базу и выводятся на интерфейс;

Как только результат планирования попадает в клиентское приложение App Relog, пользователь системы может проверить получившийся маршрут и подтвердив, начинает его использовать.

```
[pool-1-thread-3] INFO com.relog.RPC.MyConsumer - ------------------------------------------------ START ------------------------------------------------
[pool-1-thread-3] INFO com.relog.VRP.DefaultSolver - Start solving problem with key: qF2rxsqjQGe9RZqwy
[pool-1-thread-3] INFO com.relog.VRP.DefaultSolver - Start building vehicle routing problem...
[pool-1-thread-3] INFO com.graphhopper.jsprit.core.problem.VehicleRoutingProblem - setup problem: [fleetSize=FINITE][#jobs=421][#vehicles=9][#vehicleTypes=9]
[transportCost=com.graphhopper.jsprit.core.util.VehicleRoutingTransportCostsMatrix@44145acf][activityCosts=com.graphhopper.jsprit.core.problem.cost.WaitingTimeCosts@3671d6a]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.io.algorithm.AlgorithmConfigXmlReader - read algorithm-config from file algorithmConfig_greedyWithRegret.xml
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.io.algorithm.AlgorithmConfigXmlReader - read algorithm: jar:file:/src/app/target/PlanningWorker-SNAPSHOT-jar-with-
dependencies.jar!/algorithmConfig_greedyWithRegret.xml
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.io.algorithm.VehicleRoutingAlgorithms - setup executor-service with 5 threads
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.problem.vehicle.VehicleFleetManagerImpl - initialise [name=finiteVehicles]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.recreate.ShipmentInsertionCalculator - initialise [name=calculatesShipmentInsertion]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.recreate.ServiceInsertionCalculator - initialise [name=calculatesServiceInsertion]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.recreate.BreakInsertionCalculator - initialise [name=calculatesServiceInsertion]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.recreate.VehicleTypeDependentJobInsertionCalculator - initialise [name=vehicleTypeDependentServiceInsertion]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.recreate.VehicleTypeDependentJobInsertionCalculator - set vehicleSwitchAllowed to true
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.recreate.RegretInsertionConcurrentFast - initialise [name=regretInsertion][additionalScorer=[name=defaultScorer]
[twParam=-0.5][depotDistanceParam=0.1]]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.SearchStrategy - initialise searchStrategy [#modules=0][selector=[name=selectBest]][acceptor=
[name=GreedyAcceptance]]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.ruin.RuinRandom - initialise [name=randomRuin][noJobsToBeRemoved=211]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.SearchStrategy - module added [module=com.graphhopper.jsprit.core.algorithm.module.RuinAndRecreateModule@4d417dd2]
[#modules=1]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.SearchStrategy - initialise searchStrategy [#modules=0][selector=[name=selectBest]][acceptor=
[name=GreedyAcceptance]]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.ruin.JobNeighborhoodsImpl - initialize [name=neighborhoodWithCapRestriction][capacity=127]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.ruin.JobNeighborhoodsImpl - calculates distances from EACH job to EACH job --&gt; n^2=177241.0 calculations, but
`only` 53467 are cached.
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.ruin.JobNeighborhoodsImpl - preprocess distances between locations ...
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.ruin.JobNeighborhoodsImpl - preprocessing comp-time: 0.127 sec; nuOfDistances stored: 53467; estimated memory:
4945908 bytes
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.ruin.RuinRadial - initialise [name=radialRuin][noJobsToBeRemoved=127]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.SearchStrategy - module added [module=com.graphhopper.jsprit.core.algorithm.module.RuinAndRecreateModule@62e1269e]
[#modules=1]
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.VehicleRoutingAlgorithm - set maxIterations to 2000
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.io.algorithm.VehicleRoutingAlgorithms - set default prematureBreak, i.e. no premature break at all.
[pool-1-thread-3] DEBUG com.graphhopper.jsprit.core.algorithm.VehicleRoutingAlgorithm - set maxIterations to 50
```

Рисунок 9. Логи работы библиотеки Jsprit в микросервисе Relog Algo

```
+-----------------------------------------------------------------------------------------------------------------------+
| detailed solution                                                                                                     |
+---------+---------------------+----------------------+----------------+----------------+----------------+-----------+
| route   | vehicle             | activity             | job            | arrTime        | endTime        | costs     |
+---------+---------------------+----------------------+----------------+----------------+----------------+-----------+
| 1       | PxnvMaf4n98NjZdTR   | start                | -              | undef          | 1592276427     | 0         |
| 1       | PxnvMaf4n98NjZdTR   | delivery             | sC6SurYdYvRy6jCZE | 1592277415     | 1592278015     | 6667      |
| 1       | PxnvMaf4n98NjZdTR   | end                  | -              | 1592278015     | undef          | 6667      |
+---------+---------------------+----------------------+----------------+----------------+----------------+-----------+
| 2       | tdbK7Hqd96JydJukH   | start                | -              | undef          | 1592276427     | 0         |
| 2       | tdbK7Hqd96JydJukH   | pickup               | Lx9WRYAjvD64sjFDw | 1592276922     | 1592277522     | 2695      |
| 2       | tdbK7Hqd96JydJukH   | pickup               | d3QA4RDqF5sh23QJG | 1592278228     | 1592278828     | 7395      |
| 2       | tdbK7Hqd96JydJukH   | pickup               | dybDR95sqnv422ztz | 1592278861     | 1592279461     | 7539      |
| 2       | tdbK7Hqd96JydJukH   | pickup               | AByHf3ZJkBxeJzbLB | 1592279946     | 1592280546     | 10799     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | tpixzeSgZ5oG7AnX8 | 1592280751     | 1592281351     | 11950     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | E7NNQhX7BMuGF2DkH | 1592281683     | 1592282283     | 13772     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | ThYJjNSqgaP68ncaH | 1592282559     | 1592283159     | 15410     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | e85uq4yDr85edduZ6 | 1592283197     | 1592283797     | 15678     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | 2kPyiagdy9DnarH9d | 1592283819     | 1592284419     | 15830     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | CGoegD8nv7yNZHkue | 1592284424     | 1592285024     | 15863     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | o8HhRLRvLZnERJ928 | 1592285089     | 1592285689     | 16208     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | pEFHCB2euoFGJoxfv | 1592285794     | 1592286394     | 16541     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | NJjdQ2HzsGDLKwBjP | 1592286587     | 1592287187     | 17155     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | 4rAcKGNLwpGpxLRqG | 1592287507     | 1592288107     | 18486     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | PRuNMjzC62W9Kismf | 1592288121     | 1592288721     | 18522     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | Ygzixesug6nk36JAG | 1592288739     | 1592289339     | 18552     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | gbLaBq8EYXProjyPK | 1592289424     | 1592290024     | 18936     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | eRMauGQnjPdhedbXm | 1592290211     | 1592290811     | 20139     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | AwAbkE25P4wwYXsKq | 1592290933     | 1592291533     | 20949     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | YKe2Wr4LMgHDGRTrd | 1592291769     | 1592292369     | 22853     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | SdHSLaArb3g96B5ac | 1592292535     | 1592293135     | 23994     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | KjRQZdTHxrqRYnLdi | 1592293357     | 1592293957     | 25270     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | 6WgpWhGdoGwMBhPYB | 1592294003     | 1592294603     | 25481     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | mcnjX5bbuLq3gDpPs | 1592294759     | 1592295359     | 26614     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | YJNJk4AoCKFxBxEZf | 1592295367     | 1592295967     | 26636     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | hqcfoCApSjGJD5egh | 1592296193     | 1592296793     | 27722     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | dDj8jqTLz9nRw2267 | 1592296953     | 1592297553     | 29554     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | djwPK9BLgfTFXSgmp | 1592297810     | 1592298410     | 31022     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | MQcraPKgSvCEE42LE | 1592298515     | 1592299115     | 32391     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | zvu77cLwTAQHeYxH8 | 1592299210     | 1592299810     | 33250     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | 7nSoEddAyXZWr82a4 | 1592299907     | 1592300507     | 33924     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | vvgNGtZSdkEqmAvxm | 1592300602     | 1592301202     | 34565     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | fSbEHQa7GiwyDyzpL | 1592301239     | 1592301839     | 34811     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | keSBySEBXyzGJvzqv | 1592302015     | 1592302615     | 35986     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | KuvZNyrqTbJziFfz5 | 1592302741     | 1592303341     | 36860     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | hsW2hwejhFFueMx3k | 1592303400     | 1592304000     | 37254     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | 8HF4xNb6dzfho9KgC | 1592304075     | 1592304675     | 37815     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | uJZrNY8BdWJ8hKeta | 1592304800     | 1592305400     | 38377     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | YWjfL6vERzZjvqr7u | 1592305517     | 1592306117     | 38749     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | vZPC6bRWWAQMtrSa6 | 1592306147     | 1592306747     | 38816     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | fwSbY4N4hxBA8Frjw | 1592306961     | 1592307561     | 39943     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | mPagmWzY6yhuWZmiF | 1592307746     | 1592308346     | 41207     |
| 2       | tdbK7Hqd96JydJukH   | pickup               | Jyyp84oxSY4A5Pk9T | 1592308411     | 1592309011     | 41696     |
```

Рисунок 10. Логи результата планирования маршрута в Relog Algo

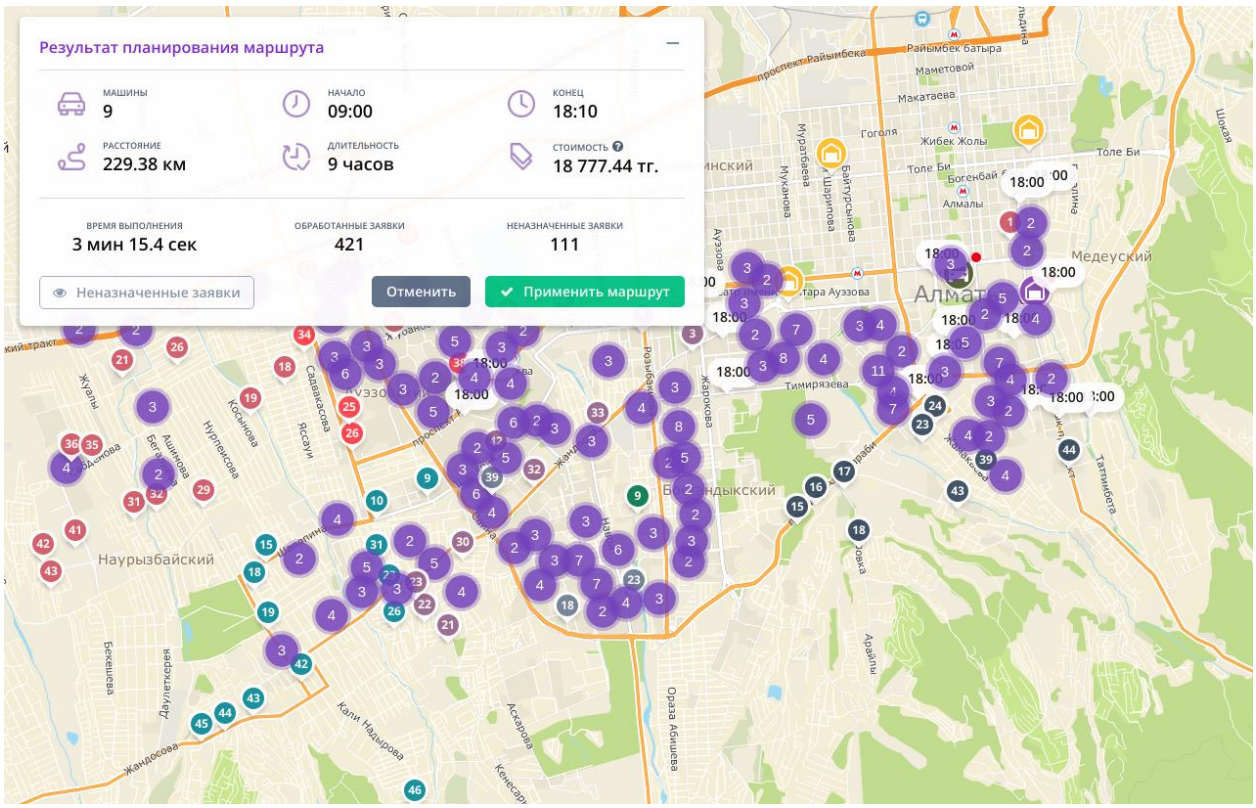

Рисунок 11. Окно результатов планирования в приложении App Relog

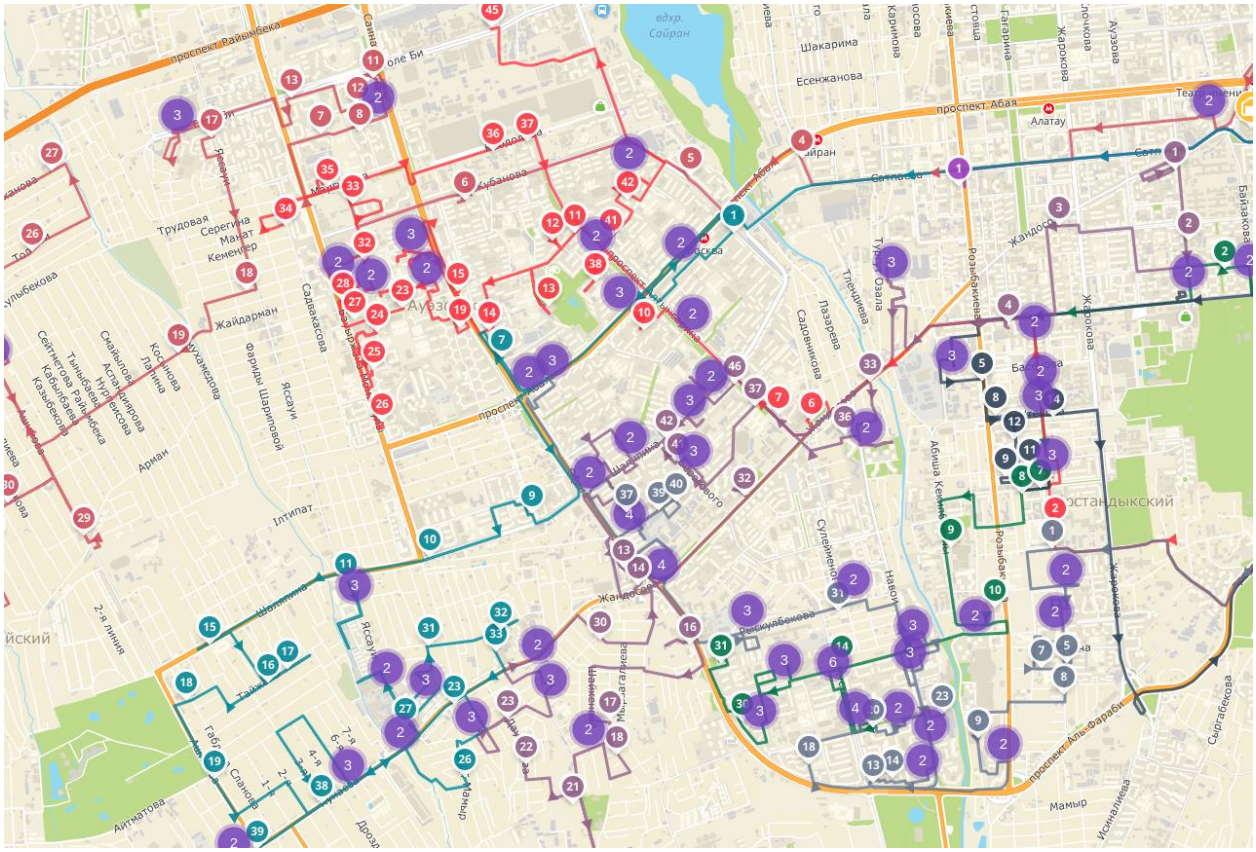

Рисунок 12. Маршрут построенный на основе решения микросервиса Relog Algo

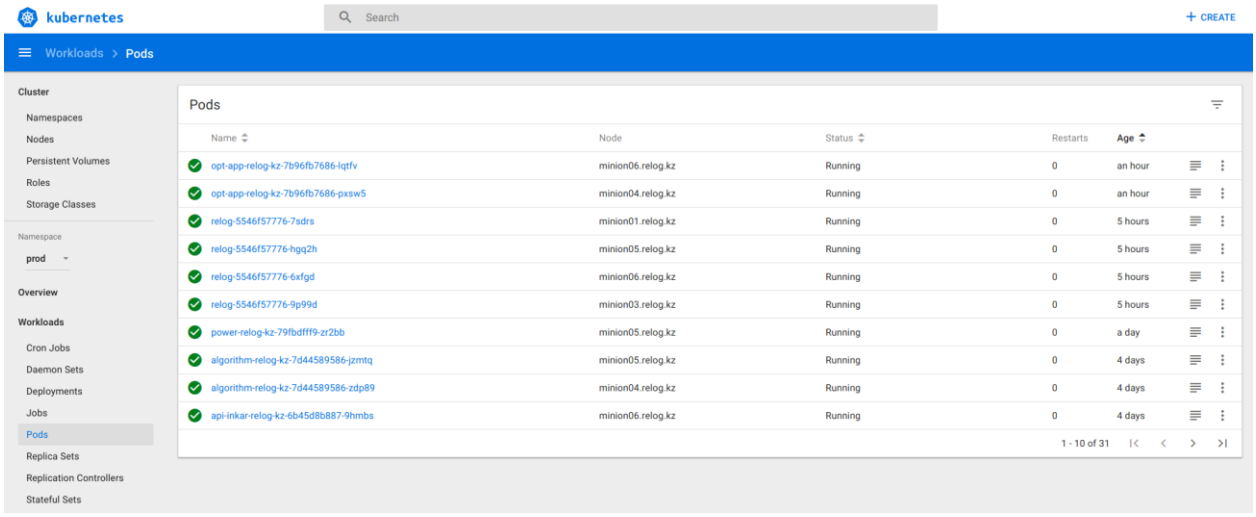

Рисунок 13. Список микросервисов в системе окрестрации Kubernetes

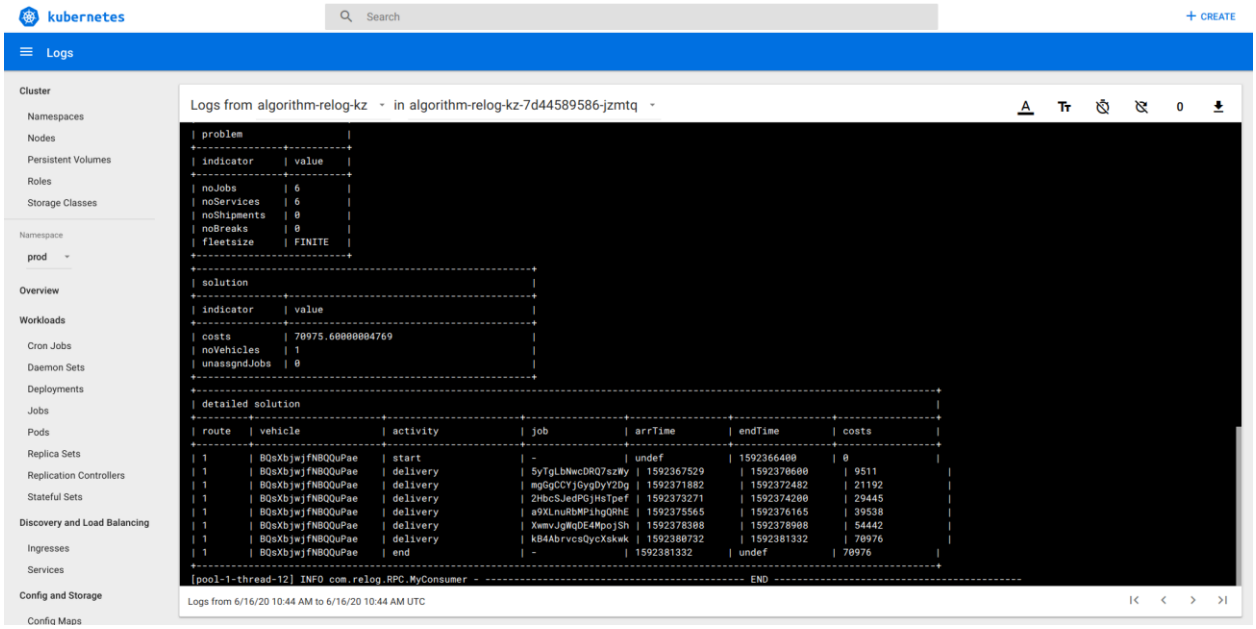

Рисунок 14. Информация о микросервисе в системе окрестрации Kubernetes

ЗАКЛЮЧЕНИЕ

В данной магистерской работе было изучено понятие масштабируемой распределенной веб архитектуры, ее типы, принципы проектирования и инфраструктурные требования. В частности был исследован процесс миграции устаревшего ПО на микросервисную архитектуру при построении масштабируемого веб-приложения на базе системы отпимизации логистики.

Во первых был детально разобран вопрос перехода от монолита к микросервисной архитектуре, проанализированы сложности и специфика данного процесса. В ходе работы были проведены следующие работы:

- Сравнительный анализ различных типов архитектур - монолит, SOA и микросервисы
- Сравнение сильных и слабых сторон монолитной архитектуры и микросервисов
- Сравнительный анализ монолитной архитектуры и микросервисов с учетом основных критериях проектирования системы
- Сравнительный анализ монолитной архитектуры и микросервисов на основе состояния проекта и команды
- Рассмотрены основные методы миграции от монолита к микросервисам с учетом преимуществ и недостатков

Во вторых были изучены современные инструменты для решения проблемы маршрутизации транспортных средств (VRP). В ходе работы были проведены следующие работы:

- Анализ инструментов OR tools, Jsprit, OptsPlanner решение задачи VRP по основным критериям оценки ПО
- Сравнительный инструмента OR tools, Jsprit, OptsPlanner на основании возможностей в решение задачи VRP

В третьих были исследованы концепции построения MSA - контейнеризация, очередь сообщений и система оркестрации и соответствующие инструменты. В ходе работы были проведены следующие работы:

- Сравнительный анализ очереди сообщений (MQ) и REST API по основным критериям оценки ПО
- Сравнительный анализ брокер сообщений Kafka и RabbitMQ с учетом основных критериях проектирования системы
- Анализ систем оркестрации контейнеров Kubernetes, Docker Swarm и Apache Mesos

В заключительной стадии работы был реализован микросервис планирования маршрутов в системе оптимизации транспортной логистики. В ходе работы были проведены следующие работы:

- Поднят микросервис планирования с использованием передовых технологий (RabbitMQ, Docker, Kubernetes), который работает независимо от остальной части системы, что делает его более гибким и доступным
- Полностью разделен процесс развертывания, тестирования и разработки
- Реализована возможность управлять и масштабировать микросервис независимо от других частей системы
- Для планирования маршрутов внедрен инструмент Jsprit, который сделал процесс планирования быстрее и качественнее.

Полученные результаты в ходе данной магистерской работы имеют научно-практическую ценность в процессе миграции от устаревшего приложения к масштабируемой распределенной веб архитектуре, которые могут быть использованы на производстве для решения различных транспортных задач.

Список используемой литературы

[1] R. Chen, S. Li, and Z. Li, “From Monolith to Microservices: A Dataflow-Driven Approach,” in 2017 24th Asia-Pacific Software Engineering Conference (APSEC), 2017, pp. 466–475.

[2] N. Dragoni et al., “Microservices: yesterday, today, and tomorrow,” ArXiv160604036 Cs, Jun. 2016.

[3] S. Newman, Building microservices: designing fine-grained systems. " O’Reilly Media, Inc.", 2015.

[4] R. Chen, S. Li, and Z. Li, “From monolith to microservices: a dataflow-driven approach,” in 2017 24th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2017, pp. 466–475.

[5] D. Taibi, V. Lenarduzzi, and C. Pahl, “Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation,” IEEE Cloud Computing, vol. 4, no. 5, pp. 22–32, 2017.

[6] Apter, Michael J. (1970). The Computer Simulation of Behaviour. London: Hutchinson & Co. p. 83.

[7] Introduction to Algorithms (Cormen, Leiserson, Rivest, and Stein) 2001, Chapter 16 "Greedy Algorithms".

[8] Z. Dehghani, “How to break a Monolith into Microservices.” [Online]. Available: https://martinfowler.com/articles/break-monolith-intomicroservices.html

[9] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, “From monolithic to microservices: an experience report from the banking domain,” Ieee Software, vol. 35, no. 3, pp. 50–55, 2018.

[10] S. Anees, “How to Migrate to Microservices.” [Online]. Available: https://blog.appdynamics.com/product/how-to-migrate-tomicroservices/

[11] M. Mishra, S. Kunde and M. Nambiar, “Cracking the Monolith: Challenges in Data Transitioning to Cloud Native Architectures.” ECSA '18 Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, Madrid, 2018, pp. 1–4.

[12] G. Mazlami, J. Cito and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, 2017, pp. 524–531.

[13] I. Azarny, “CI/CD for Containerized Microservices.” [Online]. Available: https://dzone.com/articles/cicd-for-containerised- microservices

[14] A. Balalaie, A. Heydarnoori and P. Jamshidi, "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture," in IEEE Software, vol. 33, no. 3, pp. 42–52, May-June 2016.

[15] H. Rosendahl, “Containers vs Virtual Machines (virtual security) – A Security Perspective.” [Online]. Available: https://neuvector.com/container-security/containers-vs-virtual-machines-vms/

[16] O. Pozdniakova and D. Mažeika, “Systematic Literature Review of the Cloud-ready Software Architecture.” Baltic J. Modern Computing, vol. 5, pp .124–135, March 2017

[17] S. Anees, “How to Migrate to Microservices.” [Online]. Available: https://blog.appdynamics.com/product/how-to-migrate-tomicroservices/

[18] A. Levcovitz, R. Terra, M. T. Valente, “Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems.” 3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), p. 97–104, 2015

[19] R. Chen, S. Li and Z. Li, "From Monolith to Microservices: A Dataflow-Driven Approach," 2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, 2017, pp. 466–47

[20] G. Mazlami, J. Cito and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures," 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, 2017, pp. 524–531

[21] H. Knoche and W. Hasselbring, "Using Microservices for Legacy Software Modernization," in IEEE Software, vol. 35, no. 3, pp. 44–49, May/June 2018.

[22] C. Fan and S. Ma, "Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report," 2017 IEEE International Conference on AI & Mobile Services (AIMS), Honolulu, HI, 2017, pp. 109–112.

[23] HVATTUM L M, LOKKETANGEN A, LAPORTE G. Solving a dynamic and stochastic vehicle routing problem with a sample scenario hedging heuristic[J]. Transportation Science, 2006,40(4): 421-438.

[24] AZI N, GENDREAU M, POTVIN J-Y. A dynamic vehicle routing problem with multiple delivery routes[J]. Annuals of Operations Research, 2012,199(1): 103-112.

[25] L. Li, and W. Chou, “Design and describe REST API without violating REST: A Petri net based approach,” IEEE International Conference on Web Services, pp. 508-515, July 2011.

[26] V. Mario, G. Oscar, C. Harold, V. Mauricio, S. Lorena, C. Rubby, and G. Santiago. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” In Computing Colombian Conference, 2015 10th, pp. 583-590, November 2015.

[27] V.M. Ionescu, “The analysis of the performance of RabbitMQ and ActiveMQ,” In RoEduNet International Conference-Networking in Education and Research, 2015 14th, pp. 132-137, October 2015.

[28] Avram, Abel Docker: Automated and Consistent Software Deployments (англ.). InfoQ (27 March 2013).

[29] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in 2013 21st Euromicro International Conference on Parallel, D

[30] Docker, Run swarm and kubernetes interchangeably. your choice of swarm or kubernetes for flexible and powerful orchestration options, https:// www. docker. com/ products/ orchestration, [Online; accessed 11-June-2019].