АННОТАЦИЯ

Тема выпускной квалификационной работы бакалавра — «Сравнительный анализ монолитного и микросервисного подходов к разработке web приложений в условиях низкой загрузки».

В данной работе проводится сравнительный анализ монолитного и микросервисного подходов к разработке веб-приложений в условиях низкой загрузки. Для исследования выбраны два примера проектов: монолитная структура, представленная проектом о медицинских перевозках, и микросервисная архитектура, представленная проектом о каталогизации данных производств.

Целью работы является выявление преимуществ и недостатков каждого подхода, их эффективности и применимости в контексте веб-приложений с низкой загрузкой. В работе осуществляется анализ предметной области, изучение теоретических аспектов монолитной и микросервисной архитектур, а также описание процесса разработки двух проектов, на основе которых проводится сравнение ключевых характеристик архитектур, и заключение о перспективах применения каждого подхода.

В результате работы выясняется, что в условиях низкой загрузки монолитная архитектура имеет преимущество ввиду своей простоты, логической целостности и удобства. Исследования представляют собой важный вклад в понимание особенностей различных архитектурных подходов и помогут разработчикам и архитекторам в выборе наиболее подходящего решения при проектировании собственных систем.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Разработка веб-приложений представляет собой процесс создания программных продуктов, которые функционируют на веб-платформе и доступны через интернет-браузеры. История веб-разработки началась в 1990-х годах с появлением первых веб-страниц, которые представляли собой простые статические документы. С течением времени веб-технологии эволюционировали, добавляя интерактивные элементы, базы данных и сложные пользовательские интерфейсы, что привело к возникновению современных динамических веб-приложений.

Важным аспектом разработки веб-приложений является выбор архитектуры, которая определяет структуру и взаимодействие различных компонентов системы. Этот выбор существенно влияет на производительность, масштабируемость, надежность и управляемость приложения. В течение первых 15 лет монолитная архитектура, где все компоненты приложения тесно связаны и работают как единое целое, была практически единственным подходом, применяемым в веб разработке любого масштаба и уровня. Тем не менее, при увеличении количества пользователей, а также при повышении доступных мощностей серверов и уменьшении задержки при передачи информации через интернет, монолитный подход стал приводить к громоздким программным системам, взаимодействие между которыми зачастую полагалось на стандартизованные тяжеловесные протоколы.

В результате в 2010х годах приобрела популярность микросервисная архитектура, которая, напротив, разбивает приложение на множество независимых сервисов, каждый из которых выполняет определенную функцию и может разрабатываться и развертываться отдельно. Такой подход позволил достичь большей гибкости и масштабируемости, заложив прочный фундамент для современных IT-гигантов вроде Google и Amazon.

Но процесс выбора архитектуры для веб-приложения зависит от множества факторов, включая размер и сложность проекта, прогнозируемую нагрузку, требования к масштабируемости и ресурсы команды разработчиков. В

настоящее время как в мире, так и в России, наблюдается широкое распространение обеих архитектур. Монолитные архитектуры по-прежнему популярны для небольших и средних проектов, где важны скорость разработки и простота управления. Микросервисные архитектуры набирают популярность в крупных проектах и компаниях, где требуется высокая гибкость и масштабируемость.

Цель данной работы — провести сравнительный анализ монолитного и микросервисного подходов к разработке веб-приложений в условиях низкой загрузки. Для достижения этой цели работа будет разбита на три основных раздела. В первом разделе будет подробнее рассказано про микросервисную и монолитную архитектуры, конкретные причины, почему выбор архитектуры до сих пор важен и актуален, а также какие исследования уже проведены и почему фокус данного исследования был сделан на низкой загруженности. Во втором разделе будут подробно описаны два проекта, один с монолитной, второй с микросервисной архитектурами. Наконец, в третьем разделе будут приведены несколько критериев, по которым можно проводить сравнивать проекты, и будет проведён сам сравнительный анализ для выявления достоинств и недостатков каждого подхода.

Выводы данной работы будут полезны для разработчиков и менеджеров проектов, стоящих перед выбором архитектуры для своих веб-приложений. Результаты исследования помогут определить, какой из подходов более эффективен в условиях низкой загрузки и какие критерии следует учитывать при принятии решения. Это, в свою очередь, позволит улучшить качество и эффективность разработки веб-приложений в малых и средних проектах.

ОСНОВНЫЕ ПОНЯТИЯ

Монолитная архитектура — это традиционная модель программного обеспечения, которая представляет собой единый модуль, работающий автономно и независимо от других приложений. Такое ПО обладает единой базой кода, в которой объединены все бизнес-задачи в виде одного большого унифицированного блока кода (монолита).

Микросервисная архитектура - модель программного обеспечения, при которой используются небольшие модульные сервисы. Основная концепция архитектуры в том, чтобы разделить сложное приложение на несколько небольших автономных и управляемых компонентов. Каждый микросервис имеет собственный набор кода, базу данных и API для взаимодействия с другими сервисами.

Стенды - собирательное название сред разработки, позволяющих изолировать некоторый этап жизни проекта. Так, основные виды включают тестовый стенд - среда разработки, где разработчики могут проводить любые действия, не влияя на реальную систему, и стенд прода - непосредственно запущенное приложение, с которым взаимодействуют пользователи.

User Story - это способ записи требований к ПО, который помогает команде разработки формализовать требования клиента, не прибегая к излишнему формализму. Представляет собой подробное описание сценариев поведения, которые ожидаются при взаимодействии пользователя с ПО.

Docker - программное обеспечение, предназначенное для автоматизации развёртывания и управления приложениями. Достигается это через создание ”контейнеров” для приложения, в которых хранятся их окружения и зависимости и который может быть развёрнут на любой Linux-системе.

GitHub - крупнейший сервис для хранения исходного кода программного обеспечения, основанный на системе контроля версий Git. Помимо контроля версий и хранения кода позволяет автоматизировать тестирование и деплой с помощью CI/CD.

Java - строго типизированный объектно-ориентированный язык программирования; приложения Java обычно транслируются в специальный байт-код, в результате чего достигается кросскомпиляция. Один из самых популярных языков программирования для разработки Web-приложений (2-е место в рейтингах IEEE Spectrum (2020) и TIOBE (2021)).

PSQL (PostgreSQL) - самая большая свободная объектно-реляционная система управления базами данных, активно поддерживаемая по сегодняшний день.

Vue.js - JavaScript-фреймворк с открытым исходным кодом для создания пользовательских интерфейсов.

Keycloak - продукт с открытым кодом, предназначенный для реализации регистрации и авторизации с возможностью добавления ролевой модели. Keycloak используется для написания минимального количества кода, при этом обеспечивая безопасность при аутентификации.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Описание проблемы

В данном разделе опишем проблемы, решаемые при сравнении монолитного и микросервисного подходов к разработке веб-приложений в условиях низкой загрузки. Три ключевых аспекта, которые следует учитывать при сравнении монолитного и микросервисного подходов, следующие:

- основной фокус ставится на архитектурные особенности и влияние на бизнес-логику;
- сложность и масштабируемость систем зависят от выбранной архитектуры;
- взаимодействие и сотрудничество между техническими специалистами различных областей.

Последний пункт фактически является самым важным для успешного внедрения и эксплуатации выбранного архитектурного подхода. Именно он позволяет при решении задачи максимально эффективно использовать ресурсы и достигать поставленных целей.

Монолитный подход, вопреки распространённому мнению, не является устаревшим или неэффективным. Он также имеет определённые преимущества, такие как упрощенное управление и развертывание, что особенно важно в условиях низкой загрузки. Однако, с увеличением сложности и размеров приложений, монолитная архитектура может становиться менее гибкой и более сложной для масштабирования. Микросервисный подход, в свою очередь, предлагает множество преимуществ, таких как независимое развертывание и возможность использования различных технологий для разных сервисов. Это позволяет легче управлять сложными системами и масштабировать их по мере необходимости. Тем не менее, микросервисная архитектура может быть избыточной и сложной в условиях низкой загрузки, требуя дополнительных ресурсов для управления и оркестрации сервисов.

Разработка качественного веб-приложения зависит от множества внешних и внутренних факторов, таких как бюджет, опыт команды, слаженность команды, квалифицированность мэнеджеров, требования заказчика. Тем не менее, путём выбора подходящей архитектуры возможно устранить большую часть подобных проблем.

1.2 Акутальность

Данные рассуждения уже показывают важность первой части темы данного диплома: ”сравнительный анализ монолитного и микросервисного подходов к разработке web приложений...”. Остаётся вопрос - почему была выбрана именно низкая загрузка? Выбор сравнительного анализа монолитного и микросервисного подходов к разработке веб-приложений в условиях низкой загрузки был обусловлен несколькими важными факторами. Во-первых, в современных условиях многие компании, особенно стартапы и малые предприятия, начинают свою деятельность с небольших масштабов. Для них важна оптимизация ресурсов и минимизация затрат при сохранении гибкости и возможности роста. Поэтому понимание того, какая архитектура наиболее эффективна при низкой загрузке, имеет практическую значимость.

Во-вторых, условия низкой загрузки предоставляют уникальную возможность для анализа эффективности и управляемости архитектурных подходов без влияния факторов, связанных с масштабированием под высокой нагрузкой. В таких условиях можно более детально рассмотреть преимущества и недостатки монолитной и микросервисной архитектур, выявить ключевые аспекты, которые могут повлиять на выбор архитектурного стиля в долгосрочной перспективе. Это также позволяет лучше понять, как каждый подход справляется с изменениями и поддержкой на начальных этапах развития проекта.

Более того, данное исследование фокусируется на сравнении двух конкретных примеров проектов (подробнее в разделе Реализация), которые были реализованы единой командой в примерно одинаковые сроки, по непосредственным требованиям заказчика. В начале обоих проектов мы выбирали

архитектуру, и оба выбора существенно повлияли на характеристики проекта, трудозатраты и скорость разработки. Таким образом, данное исследование применимо и необходимо в современной разработке web приложений.

1.3 Существующие исследования

Для проведения анализа существующих исследований были использованы такие интернет ресурсы, как Google Scholar и Электронная библиотека Физтеха. В результате формируется общая картина: существующие исследования уже обратили внимание на различия между монолитной и микросервисной архитектурами, но в основном они фокусируются на условиях высокой загрузки и масштабируемости. Например, исследование “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation” [1] проводит детальный анализ масштабируемости обеих архитектур. В этом исследовании рассматриваются различные сценарии высокой нагрузки и анализируется, как каждая архитектура справляется с увеличением числа запросов и данных. Результаты показывают, что микросервисная архитектура предлагает значительные преимущества в условиях высокой нагрузки благодаря своей гибкости и способности масштабироваться горизонтально.

Еще одно важное исследование, “A Comparative Review of Microservices and Monolithic Architectures” [2], уделяет внимание производительности при распараллеливании. Исследование оценивает, как обе архитектуры справляются с задачами, требующими параллельного выполнения операций. Результаты показывают, что микросервисы могут более эффективно распределять нагрузку между различными сервисами, что позволяет им лучше справляться с задачами, требующими параллельной обработки. Однако это исследование также концентрируется на условиях, где нагрузка на систему значительно выше, чем в случае низкой загрузки.

Третье исследование, “Migrating from monolithic architecture to microservices: A Rapid Review” [3], фокусируется на процессах миграции от монолитной архитектуры к микросервисам. В этом исследовании

рассматриваются различные стратегии и подходы к миграции, анализируются преимущества и недостатки каждого метода, а также их влияние на общую производительность и управляемость системы. Исследование предоставляет ценные инсайты для организаций, планирующих переход на микросервисную архитектуру, но опять же, оно больше ориентировано на системы, испытывающие значительные нагрузки и требующие повышения масштабируемости.

Несмотря на ценность этих исследований, они в основном сосредоточены на условиях высокой нагрузки и не акцентируют внимание на сценарии, когда система работает при низкой загрузке. В моем исследовании основной акцент делается на сравнении монолитного и микросервисного подходов в условиях низкой загрузки. Это позволяет выявить, как каждая архитектура справляется с задачами в условиях, где производительность и масштабируемость не являются первоочередными факторами, а важнее простота управления и экономия ресурсов.

Мое исследование также основано на реальных примерах двух проектов, разработанных мной и моей командой. Оба проекта имеют примерно одинаковую сложность, что позволяет провести объективный и детализированный анализ. Это отличается от существующих исследований тем, что мы исследуем эффективность и целесообразность каждого подхода в контексте начальных стадий разработки и низкой загрузки, что предоставляет уникальную перспективу и ценные рекомендации для стартапов и малых предприятий, начинающих свою деятельность.

1.4 Особенности каждого подхода

Опишем основные принципы и характеристики монолитной подхода и микросервисной архитектур. Монолитный подход:

- единый исполняемый контейнер: все компоненты приложения размещаются в одном исполняемом контейнере;

– монолитные приложения обычно написаны на одном языке программирования и используют одну базу данных;
– простота развертывания и управления - так как все компоненты находятся в одном приложении, развертывание и управление приложением обычно проще;
– ограничения в масштабировании - монолитные приложения могут столкнуться с проблемами масштабирования при увеличении объема функциональности.

Микросервисный подход:

– разделение на небольшие сервисы: приложение разбивается на независимые компоненты, каждый из которых выполняет определенную функциональность;
– гибкость в выборе технологий: каждый сервис может быть написан на разных языках программирования и использовать различные технологии;
– распределенная архитектура - каждый сервис может иметь свою собственную базу данных и взаимодействует с другими сервисами через API;
– масштабируемость и гибкость - микросервисы позволяют гибко масштабировать отдельные компоненты приложения в зависимости от нагрузки.

Оба подхода имеют свои преимущества и недостатки, и выбор между ними зависит от конкретных требований проекта, его масштаба и потребностей в гибкости и масштабируемости. Преимущества монолитной архитектуры заключаются в простоте разработки (единое приложение облегчает разработку и тестирование), простоте в управлении (единое приложение упрощает развертывание и мониторинг приложения), уменьшении накладных расходов (отсутствие сетевых запросов между компонентами приложения снижает накладные расходы на коммуникацию). Основные недостатки включают в себя сложности в масштабировании (монолитные приложения могут столкнуться с проблемами масштабирования при увеличении объема функциональности), затруднения в обновлении (изменения в одной части приложения могут повлиять на всю систему и требовать полного пересборки и перезапуска) и ограниченную гибкость

	Монолит	Микросервисы
сложность разработки	простая	сложная
сложность управления	простое	сложное
количество накладных расходов	небольшое	большое
масштабирование	сложное	лёгкое
надёжность	хрупкая	надёжная
гибкость	меньше	больше

Таблица 1.1 – Теоретическое сравнение архитектур

(сложнее добавлять новые функции и технологии из-за зависимостей между компонентами).

Микросервисная архитектура обладает практически противоположными характеристиками: гибкость и масштабируемость (каждый сервис может масштабироваться независимо, что обеспечивает гибкость в управлении ресурсами), улучшенная отказоустойчивость (отказ одного сервиса не приводит к полной недоступности приложения), ”технологическая свобода” (различные сервисы могут быть реализованы на разных технологиях, что позволяет использовать наиболее подходящие инструменты для каждой задачи). Из недостатков стоит выделить сложности в развертывании и управлении (управление распределенными системами требует дополнительных усилий и навыков), увеличение сложности тестирования (необходимость тестирования взаимодействия между сервисами увеличивает сложность тестирования, несмотря на упрощение unit тестирование ввиду логической разделённости сервисов) и дополнительные затраты на сетевое взаимодействие (сетевые запросы между сервисами могут вызывать задержки и увеличивать нагрузку на сеть).

Ниже представлена таблица 1.1, подводящая итог рассуждениям выше

1.5 Гипотеза

Рассуждения выше указывают на то, что микросервисная архитектура должна быть удобнее и практичнее в случае высоконагруженных сервисов. В рамках же низкой загруженности микросервисная архитектура не имеет значительных преимуществ. Соответственно, я предполагаю, что исследование покажет преимущество монолитной архитектуры, а микросервисная архитектура имеет применимость только в случае, когда еуже существует значительный опыт разработки микросервисов в первую очередь и есть кодовая база, позволяющая ускорить разработку и облегчить управление проектом.

2 РЕАЛИЗАЦИЯ

В данном разделе будут приведены описания два приложения, разработанные и реализованные под моим руководством. Будет рассказано про основные требования, выдвинутые к данным проектам, их структура, команды для работы с ними, описаны инструменты разработки и сервера, на которых проихводился деплой. Всё это будет использовано в следующих разделах в сравнительном анализе архитектур, чтобы сравнение происходило на прикладных примерах из практики.

2.1 Монолит - Medmobil

2.1.1 Общее описание

Разработка этого проекта началась с нуля. К нашей команде обратился заказчик с предложением разработать сайт для организации медицинских перевозок. Изначально данный проект представлялся в виде разового проекта, хотя после вывода MVP разработка продолжалась в качестве постоянной работы. Было необходимо реализовать сервис, который бы позволял осуществлять:

- просмотр нескольких информационных страниц;
- регистрация и авторизация пользователей, включая ролевую модель - обычные пользователи, владельцы услуг, их диспетчеры и администраторы;
- возможность создания заказа с многочисленными различными параметрами;
- возможность управления ролевой моделью в рамках полномочий роли авторизованного пользователя;
- возможность просмотра, отслеживания и редактирования заказов в зависимости от роли авторизованного пользователя.

Целью было создать аналог сервиса Яндекс.Go, но только для заказа медицинского транспорта для первозки больных в различном состоянии, как заплани-

рованных, так и экстренных. При этом спрос на медицинский транспорт оценивается сверху 100 заказами в день, поэтому данный проект был реализован под низкую загруженность.

2.1.2 План разработки

Первоначальная идея представляла собой следующий план действий:

- разработка теоретической модели базы данных;
- разработка User Stories для основных сценариев на основе представлений заказчика;
- создание базовой части проекта (инициализация проектов PSQL, Java, Vue и их связь);
- реализация MVP (создание моделей в базе данных, разработка бэкенда на Java и фронтенда на Vue);
- аренда сервера и деплой с открытым IP.

Помимо этого было принято решение использовать Github репозиторий для хранения кодовой базы, а также для проведения автоматических тестов и проверки кодстайла. По завершению разработки MVP был сформирован новый план, направленный на выпуск приложения в открытый мир:

- подключение авторизации и регистрации через сторонний сервис (Keycloak);
- снижение задержки между обнаружением ошибки в поведении программе и её ликвидацией;
- снижение ежемесячных трат на поддержание стендов для теста и прода;
- оптимизация скорости работы (в основном направлено на часть фронтенда).

2.1.3 Требования

Требования к проекту менялись в соответствии с планами и уточнялись в процессе разработки. В начале приоритеты были направлены на сокращение

времени и оптимизацию ”фундамента” для будущей разработки. Поэтому, к нашей команде были выдвинуты следующие требования:

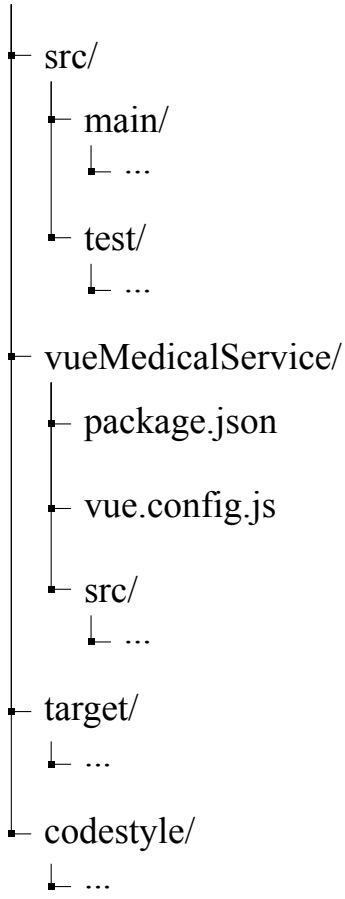
- скорость разработки;
- разработка расширяемой архитектуры;
- применении принципов разработки (DRY, YAGNI, SOLID) с целью создания удобочитаемого, поддерживаемого кода.

После создания кодовой базы и вывода первоначальной версии на стенд прода, требования были изменены в сторону поддержки уже существующей кодовой базы:

- повышение совместимости для упрощения совместимости с внешними сервисами (Keycloak);
- повышение наблюдаемости для увеличения скорости нахождения и исправления ошибок;
- уменьшение потребления ресурсов.

2.1.4 Архитектура

Выбор архитектуры происходил в соответствии с требованиями, представленными выше. Поэтому была выбрана монолитная архитектура (подробнее про причины выбора будет рассказано в разеделе Анализ). Итоговая структура репозитория выглядела так:



Как видно, все файлы лежат в одном репозитории. Папка src/ содержит все файлы, относящиеся к бэкенду - проекту на Java. Папка vueMedicalService содержит все файлы, относящиеся к фронтенду - проекту на Vue.js. Также есть общие файлы, которые относятся к проекту в целом - папка target/ содержит итоговые собранные файлы, codestyle/ содержит скрипты и конфигурационные файлы для проверки стиля кода.

2.2 Микросервисы

2.2.1 Общее описание

Разработка данного проекта началась практически с нуля. Единственное, что было дано нашей команде - это две заготовки под микросервисы, которые не были объединены никоим образом и содержали лишь базовую логику GET/SET для простой таблице в базе данных. К нашей команде обратился заказчик с

предложением разработать сайт для каталогизации и структуризации данных различных производств. Идея заключается в создании единого каталога для всевозможных технологических процессов, материалов и деталей, которые соединяются в единую производственную цепочку в соответствии с параметрами и характеристиками данной сущности.

Было необходимо реализовать сервис, который бы позволял осуществлять:

- просмотр нескольких информационных страниц;
- регистрация и авторизация пользователей, включая ролевую модель - обычные пользователи и администраторы;
- возможность управления ролевой моделью в рамках полномочий роли авторизованного пользователя;
- возможность добавления новых сущностей и параметров (в случае наличия достаточных прав);
- возможность просмотра, поиска и отображения в GUI (в табличном виде, а объектов, имеющих географическое положение, в том числе и на карте) данных о сущностях.

2.2.2 План разработки

План разработки изначально был более детализированный и продуманный на больший срок, поскольку было необходимо разрабатывать платформу для будущей тяжеловесной системы, анализирующей большие данные. По этой же причине уже заказчиком была выбрана микросервисная архитектура. Тем не менее, мы вели разработку в условиях низкой загруженности, поскольку разработкой анализа больших данных должна была заниматься другая команда, после окончания нашей рзработки системы в целом. План разработки:

- разработка теоретической модели базы данных с учётом специфики проекта (неопределённое количество параметров каждой сущности, необходимость рекурсивно вложенных отношений);

– разработка User Stories для основных сценариев на основе представлений заказчика;
– разработка микросервисов, необходимых для поддержания базы данных и операций, необходимых для корректного манипулирования данными
– создание базовой части проекта (инициализация проектов PSQL, Java, Vue);
– разработка и создание сервисов, необходимых для корректной связи микросервисов (Eureka, API Gateway, Config)
– подключение авторизации и регистрации через сторонний сервис (Keycloak);
– реализация MVP (создание моделей в базе данных, разработка бэкенда на Java и фронтенда на Vue);
– аренда сервера и деплой с открытым IP.

Также было принято решение использовать Gitlab репозиторий для хранения кодовой базы, а также для проведения автоматических тестов и проверки кодстайла.

2.2.3 Требования

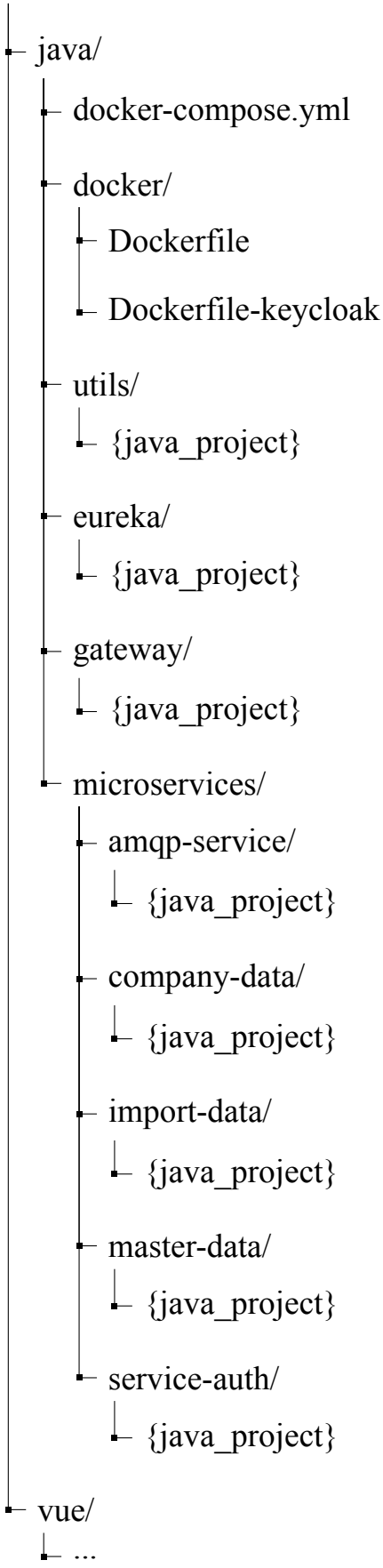
Требования к проекту формировались в соответствии с планами. Были сразу поставлены приоритеты на проектирование системы с ориентиром на последующую доработку проекта в высоконагруженный сервис. Поэтому, к нашей команде были выдвинуты следующие требования:

– разработка расширяемой архитектуры;
– применении принципов разработки (DRY, YAGNI, SOLID) с целью создания удобочитаемого, поддерживаемого кода;
– обеспечение совместимости кода для интеграции с внешними сервисами (Keycloak);
– повышение наблюдаемости для увеличения скорости нахождения и исправления ошибок;
– уменьшение потребления ресурсов;

– возможность масштабирования.

2.2.4 Архитектура

Для данного проекта была выбрана микросервисная архитектура (подробнее про причины выбора будет рассказано в разеделе Анализ). Итоговая структура репозитория выглядела так:



где сокращение [java_project] означает стандартный Java проект: pom.xml и папка src/, в которой хранится вся кодовая база.

Видно, что существует множество различных папок в бэкенде на Java, так как появилось множество отдельных подпроектов, которые отвечают за микросервисы. Также есть общие файлы, которые относятся к проекту в целом - папка utils/ содержит общие классы, docker/ содержит файлы для docker-compose.

2.3 Корректность

Как видно из приведённого выше описания, данные проекты очень похожи друг на друга: оба были целиком созданы мной и моей командой, используется одинаковый стэк. План разработки похож, хотя во втором проекте и присутствует больше шагов ввиду усложнённой структуры проекта с микросервисной архитектурой (подробнее будет в разделе Анализ). Архитектура проектов не имеет принципиальных дополнений по сравнению с базовым проектом Java, но заметно, что во втором проекте больше файлов и папок.

3 СРАВНЕНИЕ

3.1 Виды критериев

В целом, сравнения двух приложений и определения преимущества того или иного подхода (монолитного или микросервисного) можно учитывать следующие параметры:

- Масштабирование
- Производительность
- Отказоустойчивость
- Надёжность
- Переиспользуемость
- Команда
- Скорость разработки
- Совместимость
- Переносимость
- Наблюдаемость
- Потребление ресурсов

Данные параметры были взяты из исследований: ”...возможность быстро обнаружить неполадки...” [4] (т. е. наблюдаемость) и ”...обеспечить отказоустойчивость и балансировку нагрузки...” и ”...требуется больше вычислительных ресурсов для функционирования множества сервисов...” [5] (отказоустойчивость и потребление ресурсов). Также они были взяты из требований (описанных в разделе Реализация) и из пркатической разработки.

Тем не менее, не все данные критерии актуальны в условиях низкой загруженности. В большинстве случаев низкая загрузка возникает или в начале разработки проектов, или если проект был нацелен на небольшое количество пользователей и/или все запросы легковесные и редкие. Рассмотрим с данной точки зрения приведённые выше критерии и выберем подходящие.

а) Масштабирование: Процесс увеличения или уменьшения ресурсов системы для поддержания производительности под изменяющейся нагрузкой. Не подходит, потому что в условиях низкой загрузки нет необходимости увеличивать ресурсы системы для поддержания производительности.

б) Производительность: Мера эффективности системы, отражающая скорость выполнения запросов и обработки данных. Разумеется, данный критерий возможно применять в любых условиях, но для низкой загруженности не столь важна скорость обработки запросов. Не подходит, так как фокус данной работы находится на другом.

в) Отказоустойчивость: Способность системы продолжать функционировать корректно при возникновении сбоев или отказов в её компонентах. Аналогично производительности, отказоустойчивость полезна для любого проекта; тем не менее, данная работа нацелена на анализ условий, специфичных для низкой загруженности. Не подходит, ввиду отсутствия жёстких требований на отказоустойчивость в небольших проектах.

г) Надёжность: Вероятность того, что система будет функционировать безотказно и правильно в течение определённого периода времени. Не подходит по причинам, сходным с отказоустойчивостью.

д) Переиспользуемость: Способность компонентов системы быть использованными в различных контекстах и приложениях без модификаций. Здесь подходят те же аргументы - данная характеристика важнее для высокой производительности, а также её уже обсуждали в других работах, приведённых выше. Не подходит ввиду неисключительности для низкой загруженности.

е) Команда: Группа специалистов, работающих совместно над разработкой, поддержкой и улучшением веб-приложения. Подходит, так как эффективность и взаимодействие команды важны для любого проекта, независимо от уровня загрузки.

ж) Скорость разработки: Время, затрачиваемое на создание, тестирование и внедрение функционала или системы в целом. Подходит, потому что

быстрое создание и внедрение функционала важно на всех этапах проекта, особенно в условиях ограниченных ресурсов и низкой загрузки.

з) Совместимость: Способность системы или её компонентов взаимодействовать и работать с другими системами или компонентами без конфликтов. Подходит, потому что система должна взаимодействовать с другими системами и компонентами корректно, независимо от уровня нагрузки.

и) Переносимость: Способность системы или её компонентов быть перенесёнными и запущенными на различных платформах или в разных средах без значительных изменений. Подходит, так как возможность переноса системы на другие платформы или среды важна для гибкости и адаптации проекта, независимо от текущей загрузки. Более того, в условиях новых проектов важнее уметь быстро менять среду выполнения для экспериментирования или экономии бюджетных средств.

к) Наблюдаемость: Возможность отслеживания состояния системы и её компонентов через сбор и анализ метрик, логов и трассировок. Подходит, поскольку мониторинг состояния системы и её компонентов помогает выявлять и устранять проблемы на ранних стадиях, что важно при любой нагрузке.

л) Потребление ресурсов: Объём вычислительных ресурсов (таких как CPU, память, дисковое пространство и сетевой трафик), используемых системой во время её работы. Подходит, так как эффективность использования ресурсов важна при любой загрузке для оптимизации затрат и производительности.

Теперь, когда мы выделили 6 основных параметров для сравнения, перейдём к самому анализу.

3.2 Команда и скорость разработки

Эффективность команды и скорость разработки играют ключевую роль в успехе любого проекта. Эти факторы особенно важны в условиях низкой

загрузки, где ресурсы ограничены и необходимо быстрое внедрение решений. Рассмотрим эти аспекты на примере моих двух проектов.

В проекте с монолитной архитектурой наша команда разработчиков работала над единой кодовой базой. Это позволило легко управлять и координировать работу, так как все компоненты системы интегрированы в одном месте. Благодаря этому, внедрение новых функций и исправление багов происходило быстрее, так как изменения требовали минимальных изменений в кодовой базе. Команда сосредоточилась на общем контексте проекта, что упрощает коммуникацию и уменьшает риск недопонимания. Однако монолитная архитектура имеет и свои недостатки. С увеличением сложности проекта начали возникать проблемы с масштабируемостью самой команды - управление большими монолитными кодовыми базами стало сложным, методы и классы стали разрастаться, было сложнее следить за тем, чтобы соблюдались принципы SOLID. Всё это стало замедлять разработку. При внесении изменений в одну часть системы возникали непредвиденные последствия в других частях, требующие дополнительного времени на тестирование и отладку.

В проекте с микросервисной архитектурой наша команда была разбита на небольшие группы, каждая из которых отвечала за определенный набор микросервисов. Это позволяило каждому члену команды работать более автономно и параллельно, что, безусловно, ускорило разработку отдельных компонентов. Разделение ответственности также уменьшило зависимость между командами, улучшив гибкость и адаптивность процесса разработки. Также, различные микросервисы использовали различные технологии и инструменты для своих задач - так, один из сервисов отвечал за регистрацию и авторизацию с помощью стороннего сервиса Keycloak, в результате чего потребовались дополнительная библиотека keycloak-admin-client. Благодаря архитектуре микросервисов, остальные проекты не должны были менять свои зависимости для поддержки конкретной версии данной библиотеки. Тем не менее, микросервисная архитектура добавила сложности в координацию и интеграцию работы различных команд. Необходимо было постоянно обеспечивать совместимость и взаимодействие между микросервисами, что требовало дополнительного времени и

усилий. Кроме того, для нашей маленькой команды и небольшого проекта, микросервисы стали избыточными и привели к излишней сложности и увеличению времени разработки из-за необходимости настройки и поддержки инфраструктуры. Так, мы были вынуждены привлечь дополнительного специалиста по docker и архитектора баз данных для помощи в правильном построении и связи баз данных отдельных микросервисов.

Сравнивая два проекта, можно отметить, что в условиях низкой загрузки монолитная архитектура предоставляет значительные преимущества в плане скорости разработки. Единая кодовая база и простота управления позволили команде быстро вносить изменения и внедрять новые функции, что особенно важно для небольших команд, где каждая минута на счету и важно как можно быстрее доставить продукт на рынок.

Микросервисная архитектура, хотя она и может быть более гибкой и адаптивной в долгосрочной перспективе, особенно по мере роста проекта и увеличения команды, в условиях низкой загрузки и ограниченных ресурсов лишь замедляла процесс разработки из-за сложностей координации и управления распределенной системой. Таким образом, по критерию ”Команда и скорость разработки” проект с монолитной архитектурой оказывается предпочтительнее в условиях низкой загрузки.

3.3 Совместимость

Рассмотрим, насколько сильно отличается способность системы или её компонентов взаимодействовать и работать с другими системами или компонентами без конфликтов в зависимости от архитектуры. В веб-разработке совместимость включает взаимодействие между различными сервисами, базами данных, внешними API (например, Keycloak) и другими системами.

В проекте с монолитной архитектурой все компоненты системы находятся в одной кодовой базе и тесно связаны между собой. Все классы были написаны в рамках одного java проекта, с одним общим pom.xml. Совместимость здесь обеспечена на уровне общей кодовой базы и единого окружения выполнения.

Все модули разрабатывались с учетом единой платформы и стандартов, что упростило интеграцию и взаимодействие между ними. Так, например, классам OrderController и PatientController, которые отвечали за принятие запросов касающихся заказов и управления данными пациентов, понадобилось обоим запрашивать данные о пользователе, за которые отвечал UserController. В монолитной архитектуре это решилось простым запросом к общему репозиторию UserRepository. Таким образом, при внесении изменений в один компонент, разработчики могут легко протестировать влияние этих изменений на всю систему. Однако монолитная архитектура имеет свои ограничения в плане совместимости с внешними системами. Интеграция с внешними API и сервисами требовала значительных усилий, так как любые изменения или обновления этих систем могут требовали соответствующих изменений в остальных классах. Также, внедрение новых технологий было затруднено, так как обновление одной библиотеки могло тянуть за собой изменения по всему проекту - что произошло, например, при обновлении библиотеки jakarta.xml.bind-api с версии 3.x до 4.x, когда вся система была вынуждена поддержать эти изменения.

В проекте с микросервисной архитектурой каждый сервис является независимой сущностью, которая взаимодействует с другими сервисами через четко определенные интерфейсы - в нашем случае, через API. Это позволило использовать различные технологии и платформы для каждого микросервиса, что повысило гибкость и адаптивность системы. Например, микросервисы использовали разные библиотеки и даже разные версии базовой библиотеки spring-boot, а взаимодействие между ними осуществлялось через REST. Микросервисная архитектура значительно улучшило совместимость и с внешними системами - каждый микросервис был адаптирован для интеграции со своими внешними API и сервисами без влияния на другие части системы, что позволило быстрее внедрять новые технологии и адаптироваться к изменениям во внешних системах. Конечно, такая архитектура требовала дополнительных усилий для обеспечения совместимости между самими микросервисами, включая управление версиями API и мониторинг взаимодействия между ними, но совместимость была на порядок выше.

Сравнивая два проекта, видно, что микросервисная архитектура предоставляет значительные преимущества в плане совместимости. Независимость микросервисов позволяет легко адаптироваться к изменениям во внешних системах и использовать различные технологии для каждого компонента. Это особенно полезно в условиях, когда требуется интеграция с множеством различных систем и API. Хотя монолитная архитектура и может обеспечить более простое управление совместимостью внутри самой системы, так как все компоненты находятся в одной кодовой базе и работают в едином окружении, но при интеграции с внешними системами и обновлении технологий могут возникать значительные сложности, требующие больших усилий для поддержания совместимости. Следовательно, по критерию ”Совместимость” проект с микросервисной архитектурой оказывается предпочтительнее.

3.4 Переносимость

Перейдём к рассмотрению способности системы или её компонентов быть перенесёнными и запущенными на различных платформах или в разных средах без значительных изменений. В контексте веб-разработки переносимость играет важную роль при развертывании приложений на различных серверных платформах, облачных инфраструктурах или при миграции между ними.

В проекте с монолитной архитектурой вся система была разработана и развернута как единое целое. Это означает, что при переносе приложения на другую платформу необходимо обеспечить совместимость всей кодовой базы и инфраструктуры с новой средой. Монолитные приложения часто зависят от специфических конфигураций и окружений, что может затруднить их переносимость. Так получилось и в случае нашего проекта - в определённый момент потребовалось осуществить перенос приложения с Ubuntu16 на Ubuntu20. Ввиду изменений в операционной системе, библиотеки и конфигурации сервера потребовали значительных изменений в кодовой базе и настройках. Тем не менее, монолитная архитектура имеет преимущества в плане простоты раз-

вертывания. Поскольку вся система развертывается как единое целое, процесс миграции может быть более прямолинейным. Тем не менее, есть очевидная проблема с конкретно нашим решением: ввиду специфики разработки мы не использовали такие инструменты контейнеризации как Docker, чтобы упаковать монолитное приложение и его зависимости в контейнер, из-за чего и возникло большинство проблем.

В проекте с микросервисной архитектурой каждый микросервис является независимой сущностью, которая может быть развернута и запущена на различных платформах и в различных средах. Это существенно улучшило переносимость системы, так как каждый микросервис может быть адаптирован к специфическим требованиям целевой платформы. Например, один из наших микросервисов разворачивался на нашем локальном сервере, а остальные - на серверах Selectel. Каждый микросервис был упакован в собственный контейнер, что позволило легко переносить его между различными средами без изменения кода. Более того, инструменты оркестрации позволили управлять развертыванием и масштабированием микросервисов в различных средах, обеспечив гибкость и адаптивность системы. Однако, управление множеством микросервисов требует дополнительных усилий и ресурсов для обеспечения их взаимодействия и мониторинга, а выгода, получаемая от возможности масштабирования, минимальна в условиях низкой загруженности.

В результате анализа видно, что, несмотря на очевидные значительные преимущества в плане переносимости у микросервисного проекта в конкретно нашем случае, большинство проблем с переносимостью решаются стандартными инструментами контейнеризации. Независимость микросервисов хоть и позволяет легко адаптировать и развертывать их на различных платформах и в различных средах, монолитная архитектура может обеспечить более простое развертывание в тех случаях, когда система разрабатывается и развертывается в статической и неизменной среде. Таким образом, по критерию ”Совместимость” проекты находятся в сравнимом положении.

3.5 Наблюдаемость

Теперь рассмотрим наблюдаемость — возможность отслеживания состояния системы и её компонентов через сбор и анализ метрик, логов и трассировок. Наблюдаемость играет ключевую роль в поддержке и управлении веб-приложениями, обеспечивая своевременное выявление причин проблем и проведение оперативной отладки.

В проекте с монолитной архитектурой вся система работает как единое целое, что упрощает процесс мониторинга и сбора данных. Вся логика приложения и данные находятся в одном месте, что позволяет централизовать логи, метрики и трассировки. Инструменты мониторинга, такие как Jprofiler, легко интегрируются с монолитным приложением, предоставляя всесторонний обзор его состояния. Однако, несмотря на упрощенный процесс мониторинга, в больших и сложных монолитных системах объем логов и метрик может быть значительным, что затрудняет анализ и выявление проблем. Кроме того, любые изменения в мониторинговых инструментах, опять же, могут потребовать изменения в коде всей системы, что усложняет управление. Наконец, ввиду отсутствия физического разделения логически разных систем (что происходит в микросервисах) становится сложнее выделить конкретные точки связей между классами и методами, что приводит к более запутанному процессу отладки.

В проекте с микросервисной архитектурой каждый сервис работает независимо, что увеличило сложность наблюдаемости: каждый микросервис генерировал свои собственные логи, метрики и трассировки, что потребовало разработки системы для агрегации и анализа данных из множества источников, для чего были использованы инструменты, написанные на bash и Python. Кроме этого, поскольку каждый микросервис отслеживается отдельно, это позволяет более точно выявлять и устранять проблемы в конкретных компонентах системы, хотя это и требует дополнительных усилий для настройки и управления системой наблюдаемости, включая координацию между микросервисами и обеспечение совместимости данных.

Сравнив два проекта, видно, что монолитная архитектура выигрывает у микросервисной. Хотя микросервисы и предоставляют более детализированную систему наблюдаемости, более простая и централизованная систе-

му наблюдаемости монолита серьёзно упрощает сбор и анализ данных. По критерию ”Наблюдаемость” проект с монолитной архитектурой оказывается предпочтительнее

3.6 Потребление ресурсов

Последний признак, рассматриваемый в данной работе — это объем вычислительных ресурсов (таких как CPU, память, дисковое пространство и сетевой трафик), используемых системой во время её работы. Приложения могут значительно различаться по эффективности использования ресурсов в зависимости от выбранной архитектуры, даже при прочих равных параметрах.

В проекте с монолитной архитектурой все компоненты приложения работают в рамках одной кодовой базы и исполняются как единое целое. Это может быть более эффективно с точки зрения использования ресурсов, так как все компоненты приложения могут совместно использовать общие ресурсы, такие как память и процессорное время. В монолитной архитектуре отсутствуют и накладные расходы на межсервисные коммуникации, которые могут быть значительными в микросервисных системах. Однако, монолитная архитектура поставляется единым блоком, что означает большое потребление ресурсов на одном сервере. Поскольку вся система исполняется в одном процессе или наборе процессов, масштабирование отдельных компонентов невозможно, что привело к ситуации, когда для масштабирования одного компонента приходилось масштабировать всю систему, что неэффективно с точки зрения использования ресурсов и приводит к неоптимальным тратам бюджета.

В проекте с микросервисной архитектурой каждый микросервис работает независимо и может быть развернут в собственном окружении. Это позволило иметь несколько более слабых серверов, которые уже выбираются и настраиваются отдельно в зависимости от конкретных требований. Например, микросервис company-data, отвечающий за основную обработку информации, был размещён на производительном сервере с мощным CPU и забирал наибольшую нагрузку на себя, тогда как микросервис import-data, занимающийся

хранением и транспортировкой данных, использовал больший объем памяти и дискового пространства. Однако, микросервисная архитектура вводит дополнительные накладные расходы на межсервисные коммуникации. Каждый микросервис должен взаимодействовать с другими микросервисами через сеть, что требует дополнительных ресурсов и снижает общую производительность системы. Также, каждая независимая служба требует собственного набора ресурсов, таких как память и процессорное время, что привело к увеличению общего потребления ресурсов по сравнению с монолитной архитектурой.

Получаем, что, с одной стороны, монолитная архитектура более эффективна с точки зрения общего потребления ресурсов. Совместное использование ресурсов и отсутствие накладных расходов на межсервисные коммуникации позволяют монолитному приложению работать более эффективно в условиях ограниченных ресурсов. С другой стороны, микросервисная архитектура предоставляет гибкость в управлении и масштабировании отдельных компонентов, что позволяет оптимизировать использование ресурсов для каждого микросервиса. Для малонагруженных систем данные архитектурные подходы являются сравнимыми по успешности

ЗАКЛЮЧЕНИЕ

Целью данного исследования был сравнительный анализ монолитного и микросервисного подходов к разработке веб-приложений в условиях низкой загрузки. Исследование направлено на выявление преимуществ и недостатков каждой архитектуры, чтобы предоставить рекомендации по выбору подходящей архитектуры для малых и средних проектов.

В ходе работы было представлено комплексное описание монолитной и микросервисной архитектур, включающее в себя несколько пунктов. Были проанализированы уже существующие исследования с целью понимания текущих знаний по данной теме и подтверждения новизны данной работы. Также были подробно описаны два проекта, разработанные командой под моим руководством, имеющие разную архитектуру, на примере которых и проводился сравнительный анализ. Наконец, были рассмотрены ключевые вопросы, связанные с различными аспектами разработки и эксплуатации веб-приложений, включая такие критерии, как команда и скорость разработки, совместимость, переносимость, наблюдаемость и потребление ресурсов. Эти критерии были выбраны для того, чтобы дать всестороннюю оценку эффективности и применимости каждой архитектуры, сфокусировавшись на условиях низкой загрузки.

Команда и скорость разработки: Монолитная архитектура часто способствует более быстрой разработке, так как вся команда работает над одной кодовой базой. Это упрощает коммуникацию и координацию, что может быть критически важным для малых команд.

Совместимость: Несмотря на то, что каждый микросервис можно адаптировать для взаимодействия с различными системами и сервисами, монолитная архитектура обеспеичвает прямое, непосредственное взаимодействие, что важнее для упрощения разработки.

Переносимость: Микросервисы обеспечивают лучшую переносимость, так как каждый сервис можно развернуть и масштабировать независимо на разных платформах и в различных средах, используя контейнеризацию и оркестрацию. Тем не менее, контейнеризация применима так же и к монолитнной

архитектуре, что может уменьшить риски и повысить удобство управления приложением.

Наблюдаемость: Монолитная архитектура позволяет естественным образом аггрегировать логи, а также проихводить отладку с помощью одного специализированного приложения в едином месте, что упрощает разработку. Тогда как микросервисная архитектура требует дополнительных усилий для настройки и управления мониторингом.

Потребление ресурсов: В условиях низкой загрузки монолитная архитектура часто оказывается более эффективной, так как отсутствуют накладные расходы на межсервисные коммуникации. Однако микросервисы могут быть лучше оптимизированы в долгосрочной перспективе благодаря возможности независимого масштабирования.

Для малых и средних проектов с низкой загрузкой рекомендуется тщательно оценить текущие и будущие потребности проекта. Монолитная архитектура может быть предпочтительнее в случаях, когда важны скорость разработки и простота управления. Микросервисная архитектура может подойти, если проект планируется масштабировать и интегрировать с другими системами в будущем.

Монолитная архитектура оптимальна для проектов с ограниченными ресурсами и небольшими командами разработчиков, где важны скорость разработки и простота управления. Она также подходит для проектов, которые не предполагают значительного роста или масштабирования в ближайшем будущем. Для стартапов и прототипов монолитный подход может быть идеальным, позволяя быстро вывести продукт на рынок. Микросервисная архитектура рекомендуется для проектов, где требуется высокая гибкость и возможность масштабирования отдельных компонентов. Она также подходит для систем, которые требуют частых обновлений и интеграции с внешними сервисами. Проекты, которые предполагают распределенную разработку с участием нескольких команд, также могут выиграть от использования микросервисного подхода.

При выборе архитектуры важно учитывать не только текущие требования проекта, но и его долгосрочные цели. Если проект предполагает быстрый рост и развитие, микросервисы могут быть более подходящими. Для проектов с четко определенными и стабильными требованиями монолитная архитектура может оказаться более эффективной. Важно также учитывать навыки команды и наличие ресурсов для поддержки выбранной архитектуры.

Будущие исследования могут быть направлены на изучение монолитных и микросервисных архитектур в условиях высокой загрузки, чтобы оценить их производительность и масштабируемость в более требовательных сценариях. Также целесообразно исследовать влияние новых технологий и инструментов на выбор архитектуры. Наконец, важно рассмотреть гибридные подходы, которые могут объединять преимущества обеих архитектур, для чего можно разработать третий проект и провести аналогичный анализ.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

[1] Blinowski Grzegorz, Ojdowska Anna, and Przybyłek Adam. Monolithic vs. microservice architecture: A performance and scalability evaluation // IEEE Access. — 2022. — Vol. 10. — P. 20357–20374.

[2] Al-Debagy Omar and Martinek Peter. A comparative review of microservices and monolithic architectures // 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI) / IEEE. — 2018. — P. 000149–000154.

[3] Ponce Francisco, Márquez Gastón, and Astudillo Hernán. Migrating from monolithic architecture to microservices: A Rapid Review // 2019 38th International Conference of the Chilean Computer Science Society (SCCC) / IEEE. — 2019. — P. 1–7.

[4] Шитько Андрей Михайлович. Проектирование микросервисной архитектуры программного обеспечения // Труды БГТУ. Серия 3: Физико-математические науки и информатика. — 2017. — no. 9 (200). — P. 122–125.

[5] Артамонов Юрий Сергеевич and Востокин Сергей Владимирович. Разработка распределенных приложений сбора и анализа данных на базе микросервисной архитектуры // Известия Самарского научного центра Российской академии наук. — 2016. — Vol. 18, no. 4-4. — P. 688–693.