Проблемы перехода от монолитной к микросервисной архитектуре

Маличенко Сергей Владимирович
аспирант МИРЭА
Российского технологического университета,
Россия, г. Москва
E-mail: sm.malichenko@gmail.com

Аннотация. Одним из последних направлений более гибкой установки и выполнения приложений является переход от монолитной к микросервисной архитектуре. В такой архитектуре, где микросервисы могут более свободно обновляться, перемещаться и заменяться, создание ликвидного программного обеспечения также становится проще, поскольку адаптация и развертывание кода проще, чем при использовании монолитной архитектуры, где почти все взаимосвязано. В данной работе мы изучим проблемы такого перехода. Цель состоит в том, чтобы определить причины, по которым компании решают осуществить такой переход, и определить проблемы, с которыми они могут столкнуться в ходе этого перехода. Наш метод — это опрос, основанный на различных публикациях и тематических исследованиях, посвященных этим архитектурным переходам от монолитной архитектуры к микросервисам. Результаты показывают, что типичными причинами перехода к микросервисной архитектуре являются сложность и масштабируемость. С другой стороны, проблемы можно разделить на архитектурные и организационные. Вывод таков: когда компания-разработчик программного обеспечения становится достаточно большой и начинает сталкиваться с проблемами, связанными с размером кодовой базы, именно тогда микросервисы могут стать хорошим способом справиться со сложностью и размером. Несмотря на то, что переход создает свои проблемы, решить их может быть проще, чем проблемы, которые ставит перед компанией монолитная архитектура.

1. Введение

Одним из последних направлений более гибкой установки и исполнения является переход от монолитной архитектуры к микросервисной. Мотивация для этого перехода исходит из того факта, что постоянное поддержание монолитной архитектуры привело к трудностям в том, чтобы идти в ногу с новыми подходами к разработке, такими как DevOps, требующими развертывания несколько раз в день. Напротив, микросервисы предлагают более гибкий вариант, когда отдельные сервисы соответствуют принципу единой ответственности (SRP) [1], и поэтому их можно масштабировать и развертывать независимо друг от друга [2]. Такая архитектура явно поддерживает создание адаптивного программного обеспечения, поскольку оно более свободно обновляется, перемещается и заменяется, чем их традиционные, обычно монолитные аналоги. В этой статье мы изучаем причины, по которым компании решаются на переход от монолитных архитектур к микросервисам, и определяем проблемы, с которыми можно столкнуться при подобном переходе. Исследование основано на различных публикациях и тематических исследованиях, посвященных архитектурным переходам от монолитной архитектуры к микросервисам.

Остальная часть статьи структурирована следующим образом. В разделе 2 обсуждается предыстория. В разделе 3 сравниваются монолитные и микросервисные архитектуры с разных точек зрения. В разделе 4 представлены проблемы, возникающие при переходе от монолита к микросервисам. Ближе к концу статьи в разделе 5 делаются некоторые окончательные выводы.

2. Предыстория

Микросервисы — это небольшие сервисы, соответствующие принципу единой ответственности (SRP) [1]. Каждый сервис ориентирован только на одну функциональность. Такой подход определяет, где проходят границы между различными сервисами. Следовательно, микросервисы по своей природе слабо связаны [4]. Слабое связывание дает разработчикам возможность вносить независимые изменения в сервисы, не затрагивая остальную часть кодовой базы. Поскольку микросервисы не привязаны друг к другу, их можно масштабировать и развертывать независимо [2]. Такая архитектура также является ключевым фактором для создания гибкого программного обеспечения, поскольку она более свободно обновляется, перемещается и заменяется, чем их традиционные, обычно монолитные аналоги.

Все эти качества делают микросервисы желательным вариантом для существующих монолитных приложений. Масштабирование монолитного приложения всегда сложнее, чем масштабирование микросервисов, потому что нужно масштабировать все приложение и развертывать всю кодовую базу, а не масштабировать часть приложения, которая требует больше ресурсов [6]. Текущее развитие облачных сервисов делает автоматическое масштабирование ресурсов очень простым и экономичным. Микросервисы максимально используют это автоматическое масштабирование. При разработке и развертывании больших монолитных приложений компании не могут в полной мере использовать эти функциональные возможности.

По мере того, как приложения увеличиваются в размерах в течение многих лет разработки, их становится все труднее поддерживать и вносить в них изменения [5]. Можно поддерживать и развивать монолитное программное обеспечение, но в конечном итоге становится очевидным, что необходимо внести изменения в архитектуру всего приложения. Такая тенденция впервые была замечена в компаниях с большим трафиком, большим количеством разработчиков и большой кодовой базой. Такие компании, как Amazon, Netflix, LinkedIn, SoundCloud и многие другие перешли на микросервисную архитектуру, потому что их существующее монолитное приложение было слишком сложно поддерживать и развивать.

У монолитных приложений есть свои недостатки, когда кодовая база приложения разрастается, то изменения приходится вносить быстро. Небольшое расширение также невозможно с монолитом, потому что каждый раз нужно развертывать все приложение целиком. Однако, когда команды начинают разрабатывать новое ПО, бизнес требует быстрой разработки новых функций в самом начале, чтобы компания могла выжить. Монолитные приложения упрощают разработку, развертывание и тестирование, когда размер кодовой базы относительно невелик. По этим причинам большинство приложений имеют монолитную архитектуру. Вначале достаточно монолитного подхода, и вполне возможно, что размер кодовой базы и необходимость легкого масштабирования никогда не понадобятся. Это означает, что лучше оставаться с монолитом и избегать технических и организационных проблем, связанных с микросервисной архитектурой. Есть и иные мнения. Можно возразить, что рефакторинг существующего монолита является слишком сложной задачей, и вместо этого организация должна потратить больше времени в начале процесса на оценку требуемой архитектуры и функциональных возможностей. Гораздо проще внедрить случайную тесную связь в монолит, чем в микросервисы. Разрыв этих тесных связей может быть трудным и требует много времени и понимания структуры приложения.

Независимо от того, следует ли начать с микросервисов или монолита, который впоследствии будет преобразован в микросервисы, существует множество технических проблем, которые необходимо решить, чтобы их использовать. Микросервисы добавляют больше дистрибутивов приложения, что добавляет больше точек отказа. Это поднимает множество вопросов, например, как обрабатывать сбои, как службы должны взаимодействовать друг с другом, как обрабатывать

транзакции и т. д. [1]. Если монолитное приложение по какой-то причине перестает работать в продакшене, это очень быстро распознается, так как основной функционал не выполняется. С микрослужбами несколько иначе — если одна служба перестает отвечать на запросы, другие службы продолжают работать, и такие ситуации с ошибками необходимо обрабатывать должным образом. Коммуникация между микросервисами — один из главных вопросов, который нужно решить правильно. Неправильная коммуникация может привести к ситуации, когда микросервисы потеряют свою автономию и, таким образом, основные преимущества всего подхода могут нивелироваться [1]. Кроме того, связь между несколькими микросервисами может привести к проблемам с производительностью, в случае если сервисы слишком детализированы. Кажется, существует общепринятое мнение, что сложность должна заключаться в услугах сервиса, а не в каналах обмена сообщениями [2]. Одной из проблем также является управление оркестровкой микросервисов в производственной среде. К счастью, за последние несколько лет было создано много новых инструментов для поддержки оркестрации, таких как Kubernetes или Mesos.

3. Сравнение монолитной и микросервисной архитектуры

Монолитная архитектура — это стандартный способ начать разработку приложений, поскольку она более проста. Монолитное приложение разрабатывается и развертывается как единое целое, содержащее все необходимые части. Типичное монолитное приложение состоит из уровня пользовательского интерфейса, уровня бизнес-логики и уровня доступа к данным, который взаимодействует с базой данных, как показано на рисунке 1. Монолитная архитектура — хороший способ начать разработку, поскольку она ускоряет первоначальную разработку.

Однако по мере роста кодовой базы возрастают проблемы монолитной архитектуры. Новые функции и модификации старых функций сложнее реализовать, потому что разработчик должен найти правильное место для применения этих изменений. Требуется много времени, чтобы ознакомиться с большой кодовой базой. Это означает, что новым разработчикам требуется время, чтобы освоиться, поскольку они чувствуют себя потерянными и не могут найти правильное решение для применения изменений. При рефакторинге большой монолитной кодовой базы изменения могут отражать многие части программного обеспечения. Это может привести к тому, что разработчики будут с осторожностью выполнять большие задачи по рефакторингу, потому что их изменения влияют на множество мест, и проверка того, что все по-прежнему работает, является большой задачей. Это может даже привести к ситуациям, когда рефакторинг игнорируется, потому что это слишком рискованно. Что приведет к плохому коду. Поскольку разработчики не знакомы с кодовой базой, весьма вероятно, что уровень дублирования кода повысится, поскольку практически невозможно найти существующий код, который уже делал бы то же самое. Кроме того, весьма вероятно, что модульность кодовой базы снижается по мере роста кодовой базы, поскольку нет жестких границ модулей.

Рисунок 1. Типичное монолитное приложение, состоящее из трех слоев.

Есть и другие причины для разделения монолита, помимо размера кодовой базы. Одна из причин — командная структура [1]. Если команды расположены в разных географических областях и их связь очень медленная, имеет смысл взять на себя ответственность за разные части кода. Это упрощает разработку, так как теперь у команд в разных географических регионах нет конфликтов по поводу изменения частей программного обеспечения. Команда, владеющая службой, может решать, что происходит внутри этой службы, а другие команды должны заботиться только об интерфейсе службы. При таком подходе общение не должно быть таким быстрым и детализированным, как раньше. Кроме того, если команда владеет сервисом, более вероятно, что код останется чище, а технические проблемы будут решены быстрее, поскольку больше некого винить в качестве кода. Это разделение также означает, что, если команда хочет опубликовать новую версию службы в рабочей среде, это будет намного быстрее и проще, потому что команде не нужно так много общаться и координировать изменения с другими командами.

В целом можно сказать, что если к организации предъявляются следующие требования, то монолитный подход может быть правильным выбором: количество команд невелико, кодовая база относительно невелика и останется такой в ближайшие годы, команды находятся близко друг к другу географически, и их общение не затруднено. Кроме того, большое значение имеет сложность предметной области, и непросто сказать, когда использовать монолитный подход вместо микросервисов или наоборот. Можно иметь монолит, который имеет хорошую модульность и чистый код, но требуется больше работы, чтобы сохранить эту модульность при работе с монолитом вместо микросервисов, поскольку микросервисы обеспечивают модульность по своей природе и затрудняют нарушение модульности.

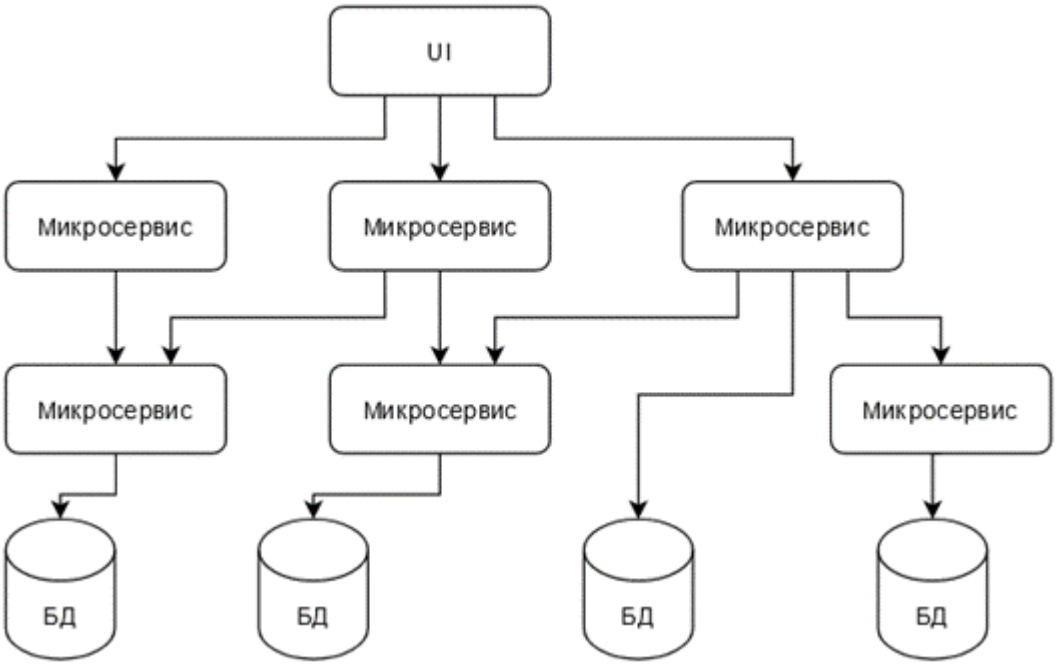

Рисунок 2. Пример микросервисной архитектуры.

Типичное монолитное приложение можно увидеть на рисунке 1. Многоуровневая архитектура с тремя уровнями очень распространена, особенно в корпоративных приложениях. Как мы видим, обычно существует только одна база данных, а это означает, что, если некоторые данные требуют лучшего масштабирования, такая архитектура не поддерживает это. Это ограничивает выбор, который команды могут сделать для поддержки бизнес-требований и требований масштабирования. Например, большая часть данных прекрасно впишется в систему управления реляционной базой данных, но некоторые части данных приложения требуют большей масштабируемости и производительности, которые, например, может обеспечить Cassandra. Одна база данных также означает, что изменения схемы должны координироваться между несколькими командами, что замедляет разработку.

На рисунке 2 на можно увидеть, что с микросервисами можно выбрать механизм базы данных для каждого микросервиса. Такой шаблон называется базой данных на службу. Это дает командам больше свободы в выборе инструментов. Проектирование и масштабирование базы данных упрощается, когда база данных состоит из меньшего количества таблиц, а микрослужба имеет полный контроль над данными и схемой базы данных. Некоторые микросервисы могут быть вообще без базы данных, если они, например, пишут на диск. На рис. 2 также видно, что микросервисы обычно взаимодействуют друг с другом, и пользовательский интерфейс может запрашивать данные из нескольких сервисов.

Таблица 1 содержит сравнение этих двух архитектурных стилей. Как мы видим, у обоих есть плюсы и минусы. В заключение по таблице можно заметить, что стиль микросервисной архитектуры становится привлекательным, когда мы работаем с большой кодовой базой. В небольших проектах технические проблемы, возникающие при использовании микросервисов, могут не окупиться. Кроме того, если команде не хватает навыков DevOps, возможно, в начале будет лучше придерживаться монолита.

Таблица 1. Сравнение монолита и микросервисов

Категория	Монолит	Микросервисы

Готовность	Быстро, потом медленнее, по мере роста кодовой базы.	Вначале медленнее из-за технических проблем и инфраструктуры, но со временем разработка ускоряется
Рефакторинг	Чрезмерно усложнена	Проще и безопаснее, потому что изменения содержатся внутри микросервиса.
Развертывание	Весь монолит должен быть развернут целиком.	Можно развертывать небольшими частями, только по одному сервису за раз.
Язык кодирования	Трудно изменить. Так как кодовая база большая. Требуется перезапись всего приложения.	Язык и инструменты можно выбрать для каждой службы. Сервисы небольшие, поэтому изменить их легко.
Масштабирование	Масштабирование означает развертывание всего монолита.	Масштабирование может быть выполнено для каждой службы.
Навыки DevOps	Не требует больших знаний, так как количество технологий ограничено.	Требуются уверенные знания DevOps для поддержки большой инфраструктуры.
Понятность	Низкая	Легко понять, поскольку кодовая база является строго модульной, а сервисы используют SRP.
Производительность	Никаких накладных расходов на связь. Стек технологий может не поддерживать производительность.	Связь добавляет накладные расходы. Возможный прирост производительности из-за выбора технологии.

4. проблемы внедрения микросервисной архитектуры

Проблемы с внедрением микросервисной архитектуры можно разделить на две части. технические проблемы и организационные проблемы [1]. И то, и другое одинаково важно для правильного понимания. Проблемы немного отличаются, если разработка приложения будет начинаться с нуля, по сравнению с преобразованием большой существующей кодовой базы из монолита в микросервисы. В этой статье мы сосредоточимся на проблемах, связанных с рефакторингом существующего монолитного приложения в сторону микросервисной архитектуры. Большинство из этих проблем необходимо решить и при запуске микросервисов с самого начала. Самые большие различия заключаются в том, что нет необходимости в больших организационных изменениях и рефакторинге, но выбор сервисов и их бизнес-требований может быть сложнее, если команды решат начать с микросервисной архитектуры с самого начала.

4.1 Технические проблемы

Существуют различные технические проблемы, которые необходимо решить, прежде чем можно будет использовать микросервисную архитектуру. Когда отправной точкой является монолитное приложение, организация, скорее всего, уже знакома с предметной областью и имеет представление о том, где можно найти стыки приложения [1]. Самая большая проблема в этих

случаях состоит в том, чтобы разделить эти службы. Рефакторинг сервисов из монолитной архитектуры может занять много времени и усилий. Вот почему рефакторинг в сторону микросервисов должен выполняться небольшими частями. Кроме того, при реализации новых функций их не следует добавлять к монолиту, даже если это может быть быстрее. Вместо этого организациям следует расширить предложение микросервисов и добавить новые микросервисы для замены старого монолитного кода. Применяя этот механизм, организация постепенно переводит большую часть кодовой базы в сторону микросервисов. Чрезвычайно важно соблюдать осторожность при выполнении этого рефакторинга, потому что существует возможность внесения новых ошибок в существующие функции. Вот почему перед началом этого процесса необходимо хорошее тестовое покрытие.

Тестирование можно рассматривать как средство реализации всего проекта рефакторинга. Если большая часть тестирования выполняется вручную, то может быть хорошей идеей сначала получить автоматическое тестовое покрытие и отложить рефакторинг в сторону микросервисов. Хорошее автоматическое тестовое покрытие помогает рефакторингу, а также дает возможность получить больше от микросервисов. Микросервисы можно выпускать часто только в том случае, если можно проверить, что программное обеспечение делает то, что ему нужно [1]. Существует несколько стратегий автоматического тестирования, которые разработчики могут применять к своему приложению в зависимости от его потребностей. Непрерывная интеграция и непрерывная доставка идут рука об руку с микросервисами. Без этих двух практик очень сложно управлять несколькими службами, их развертыванием и проверкой действий службы.

После того, как автоматическое тестовое покрытие создано, становится возможным начать думать о других проблемах. Первое, что нужно сделать, это определить микросервисы и зоны их ответственности. Важно, чтобы декомпозиция сервисов была правильной [1]. Это важно, потому что вносить много изменений во все сервисы дорого. Вместо этого легко изменить функциональность внутри одной службы, но, когда изменения затрагивают несколько служб и их интерфейсов, задача усложняется и требует больше времени. Именно здесь полезен предыдущий опыт работы с монолитом и методами проектирования его компонентов, поскольку разработчики уже должны иметь хорошее представление о бизнес-концепциях приложения. Вероятно, лучше всего начать с самых простых и очевидных сервисов, а когда у организации будет больше знаний об архитектуре микросервисов, сервисы могут стать более детализированными. Существующие микросервисы можно разделить на более мелкие сервисы, когда есть лучшее понимание состава сервисов.

При дроблении сервисов следует обращать внимание на то, чтобы сервисы не становились слишком простыми. Микросервисы могут привести к снижению производительности, особенно если связь осуществляется по сети [1]. Например, если связь осуществляется с использованием REST через HTTP, каждый межсервисный вызов добавляет дополнительную нагрузку из-за задержки в сети, а также из-за сортировки данных. Если сервисы слишком детализированы, между ними будет большой трафик, и поскольку каждый вызов увеличивает нагрузку, в результате система может работать недостаточно хорошо.

Одной из самых больших проблем является интеграция между различными микросервисами [2]. Не рекомендуется привязывать интеграцию между сервисами к какой-то конкретной технологии, потому что команды могут использовать разные языки программирования при реализации сервисов. Вместо этого лучше использовать технологию, которая не требует определенного языка программирования. Есть также множество других проблем, связанных с интеграцией микросервисов. Интерфейс микросервиса должен быть простым в использовании и иметь хорошую обратную совместимость, поэтому при введении новых функций клиенты,

использующие сервис, не должны обязательно обновляться. Как и любой хороший интерфейс, он также должен скрывать детали реализации внутри.

Использование микросервисов в производственной среде создает новые проблемы, которые необходимо решить. В рабочей среде могут работать сотни различных служб, и многие службы могут иметь несколько запущенных экземпляров, чтобы соответствовать масштабу, который требуется от приложения. Такое большое количество микросервисных организаций, работающих в производственной среде, означает, что должны быть инструменты для автоматического развертывания, масштабирования и управления этими сервисами. Процесс развертывания вручную невозможен, если развертывание в рабочей среде выполняется несколько раз в день. Docker и аналогичные технологии упрощают разработку и развертывание микросервисов. Docker делает микросервисы легко переносимыми и изолированными. Нет конфликтов зависимостей или необходимости настраивать каждую среду. С помощью Docker разработчики могут легко имитировать производственную среду в своей локальной среде разработки. Если принято решение использовать Docker в производственной среде, существует несколько инструментов для масштабирования, развертывания и управления этими контейнерами. Такие инструменты, как Kubernetes, упрощают решение этих задач. Kubernetes предоставляет множество функций, таких как горизонтальное масштабирование, обнаружение сервисов, балансировка нагрузки и так далее.

В дополнение к проблемам с инфраструктурой существуют также такие проблемы, как ведение журнала и мониторинг, которые требуют большего внимания при работе с микросервисами, чем при работе с монолитным приложением. В случае сбоев необходимо хорошее логирование. Эти журналы, также, должны быть легко доступны для поиска, и все службы должны собирать журналы в одном месте, чтобы упростить поиск проблем. Когда в продакшене работает только одно монолитное программное обеспечение, отслеживать его намного проще. Монолитное приложение можно масштабировать до нескольких узлов, но по-прежнему требуется отслеживать меньше узлов или контейнеров, чем при запуске микрослужб в рабочей среде. Это означает, что, когда нужно отслеживать больше, также должны быть хорошие автоматизированные инструменты, которые уведомляют людей, которым необходимо действовать, в случае сбоя микросервиса. Поскольку существует больше движущихся частей, становится более вероятным, что служба выйдет из строя или возникнут другие проблемы, такие как, например, высокая задержка отклика службы. Пользователи могут не заметить, что один микросервис не работает, и может показаться, что все работает нормально. При запуске монолита в продакшене пользователи сразу заметят, что весь сервис не функционирует должным образом.

В организации из нескольких микросервисов, следует также учитывать возможность того, что сервис может не отвечать на запросы. Дизайн микросервисов должен быть отказоустойчивым. В распределенной системе с большим количеством сервисов неизбежно, что в какой-то момент сервис может оказаться под большой нагрузкой и не сможет своевременно ответить. Здесь схема автоматического выключателя становится необходимой. Шаблон прерывателя цепи быстро обрабатывает сбои и может обеспечить откат, который возвращает данные по умолчанию вместо ожидания ответа от зависимого сервиса. Автоматический выключатель отслеживает сбои, и когда сбоев достаточно, последующие вызовы зависимых сервисов не будут выполняться, а вместо этого будет возвращена ошибка. Это означает, что вместо добавления дополнительной нагрузки к зависимости путем выполнения новых вызовов пользователю немедленно возвращается ошибка, которая дает зависимости время для восстановления после нагрузки. Кроме того, если это возможно, может быть предоставлен резервный метод. Например, когда продуктовой службе не удается получить персонализированные рекомендации по продуктам, она может вернуться к рекомендациям, которые привязаны к этому продукту по умолчанию, или вместо этого просто ничего не возвращать в качестве рекомендаций, и тогда пользовательский интерфейс может

обработать этот случай. Такой подход означает, что пользователь может даже не заметить, что микросервис, обслуживающий рекомендации, не работает. Есть несколько готовых решений, которые можно использовать в микросервисах. Самый известный из них — Hystrix. Hystrix — это библиотека, обеспечивающая задержку и отказоустойчивость распределенных систем. Использовать Hystrix довольно просто, это позволяет разработчикам легко делать обработку зависимостей о обеспечить приемлемую отказоустойчивость.

Управление данными является важной частью любого приложения. Есть много важных вопросов, таких как использование реляционной базы данных или NoSQL, какой поставщик базы данных лучше всего подходит для использования приложения и какую схему должна иметь база данных. Микросервисы предоставляют свободу использования нескольких механизмов баз данных. Этот шаблон, называемый базой данных для каждого сервиса, имеет свои проблемы. Несколько разных баз данных означают, что управлять ими сложнее, и у организации может быть недостаточно знаний о базе данных.

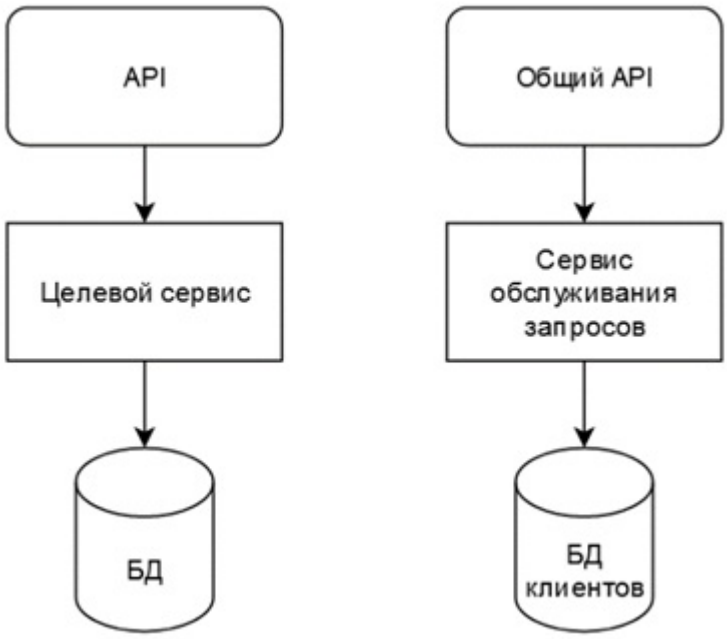

Рисунок 3. Иллюстрация использования базы данных каждым сервисом.

Раньше, если монолитное приложение использовало традиционную реляционную базу данных, то использование транзакций ACID (атомарность, непротиворечивость, изоляция, устойчивость) было простым. Теперь, когда есть несколько сервисов, каждый из которых имеет свою собственную базу данных, транзакции сложнее обрабатывать, и требуется больше времени для обработки транзакций. Вместо транзакций микросервисы могут договариваться об окончательной согласованности данных. Это означает, что изменения, сделанные другими службами, могут быть сохранены не сразу, но в конечном итоге они будут сохранены, когда служба обработает сообщение. Если раньше пользователю приходилось ждать завершения всей транзакции, то теперь он может быть не в состоянии немедленно изучить данные, которые создаст служба зависимостей. На рисунка 3 показан подход «одна база данных на службу». В этом случае служба владеет данными, и, если, например, службе заказов нужно что-либо знать о счетах, она должна пройти через API шлюз. Это приводит к потере связанности услуг. Если команде, управляющей службой выставления счетов, необходимо изменить схему базы данных счетов, они могут сделать это, не меняя никаких других служб, кроме службы счетов, при условии, что API остается прежним.

Также возможно использовать одну единую базу данных для всех сервисов. Этот подход

показан на рис. 4. Одна база данных для всех служб, однако, проблематична, поскольку теперь схема базы данных тесно связана. Одна база данных также означает, что службы имеют доступ к данным, которые должны быть доступны только через вызовы других служб. Это может привести к потере модульности, так как очень легко запросить данные из другой таблицы напрямую вместо того, чтобы вызывать службу, которая должна возвращать эти данные. На рисунке 4 мы видим, что служба заказов также имеет доступ к схеме счетов. Это позволяет разработчикам службы заказов легко получать данные из счетов, не используя API счетов. Это приводит к жесткой связи во время разработки: если команда, разрабатывающая службу выставления счетов, хочет изменить схему, теперь им приходится координировать эти усилия с несколькими другими командами. Общая база данных снижает многие положительные стороны микросервисов, и использование одной общей базы данных не рекомендуется [1]. Вместо этого при рефакторинге в сторону микросервисной архитектуры монолитная база данных также должна быть разделена на несколько баз данных, доступ к которым может получить только служба, которая обрабатывает этот бизнес-контекст.

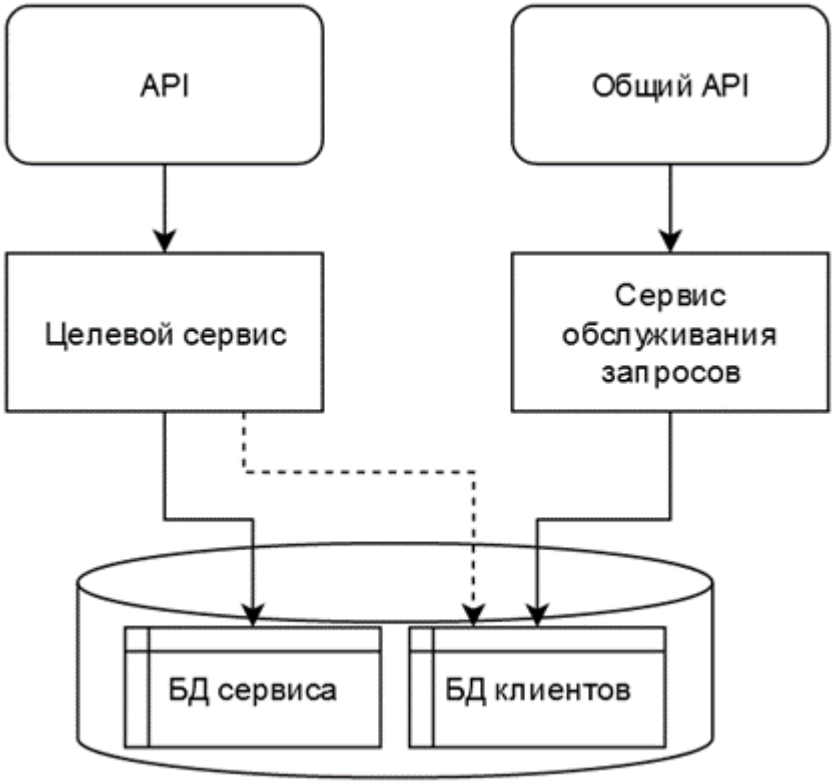

Рисунок 4. Иллюстрация взаимодействия нескольких сервисов с одной базой данных.

4.2 Организационные проблемы

Помимо технических проблем, связанных с микрослужбами, существуют организационные проблемы, которые необходимо решить при переходе от монолитной архитектуры к архитектуре микрослужб. Даже если организация решает все технические задачи, структура и навыки организации также должны поддерживать новую архитектуру [1].

Одной из проблем является структура организации. Чтобы разработать хорошее приложение, компания должна привести свою структуру в соответствие со структурой архитектуры приложения. Если раньше с монолитным приложением в компании были большие команды, у которых были четкие роли, такие как обеспечение качества, разработка и администрирование базы данных, то такая организационная структура не работает с микросервисами. Закон Конвея гласит, что организация, проектирующая систему, создаст систему, структура которой является копией структуры организации. Если структура организации монолитна, то микросервисный подход не работает. Организация должна разделить эти большие команды на более мелкие, которые могут

работать автономно. Таким образом, структура архитектуры соответствует структуре организации и не противоречит друг другу.

На рисунке 5 показана типичная организация разработки при работе над монолитными проектами, состоящая из команд, имеющих очень специализированные области деятельности. Команды очень хороши в своих специализированных областях, но при предоставлении бизнес-функций им необходимо сотрудничать друг с другом. Такая структура приводит к более медленным циклам разработки. На рисунке 6 показана организация, построенная вокруг микросервисов и служб. Когда команды организованы таким образом, у них больше автономии в отношении своих релизов.

Рисунок 5. Организация команд при разработке монолитных приложений.

Теперь нет процесса передачи третьей стороне, и командам не нужно ждать, пока другие команды завершат свои изменения. Если есть новые бизнес-требования для выставления счетов, команда, отвечающая за обслуживание счетов, может развернуть свои изменения в рабочей среде, когда они будут готовы. Такая структура снижает потребность в детальном общении, поскольку команды могут работать отдельно. Команды имеют полный контроль над своим обслуживанием и графиком развертывания, что дает им право собственности на продукт. Помимо своих технических навыков, они также будут развивать навыки, связанные с конкретным продуктом, что означает, например, что команда бухгалтерских услуг будет иметь лучшие деловые возможности в этой области.

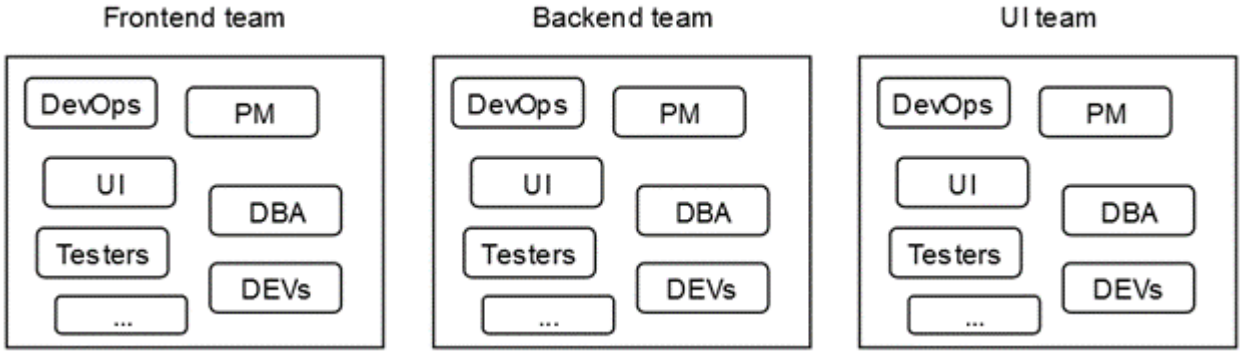

Рисунок 6. Организация команд при разработке микросервисных приложений.

Когда организация принимает стиль микросервисной архитектуры, у команд должно быть больше свободы и ответственности, но меньше производственных процессов. Это означает, что команды могут развернуть свой сервис в рабочей среде, когда им это нужно вместо того, чтобы ждать одобрения от кого-то другого. Команды владеют своей кодовой базой и несут ответственность за функциональность службы. Они больше не могут винить кого-то другого в своих неудачах, и, если есть проблемы в производстве, команда, ответственная за этот сервис,

должна их исправить. Владение кодом приносит разработчикам удовлетворение за внешний вид, улучшения и обязывает разработчиков к долгосрочному участию вместо того, чтобы всегда думать, что проблемы в кодовой базе. Владение также дает командам свободу развивать сервис по своему усмотрению. Возможно, некоторые ограничения все же имеют смысл, но с микросервисами ограничений гораздо меньше, чем с монолитом. Такое изменение очень драматично. Может быть страшно вдруг взять на себя полную ответственность за код, если до архитектурного рефакторинга у людей был шанс спрятаться от ответственности. Требуется время, чтобы команды приняли эту новую ответственность и увидели в ней положительные стороны. Хорошей золотой серединой было бы иметь в начале операционную команду, которая по-прежнему несет основную ответственность за производство, чтобы у команд было время принять тот факт, что они владеют кодовой базой [1]. Когда командам удобно владеть кодовой базой, они могут постепенно также взять на себя ответственность за производство своего сервиса.

Когда команды берут на себя полную ответственность за свои услуги, им могут потребоваться новые навыки для развертывания и устранения проблем в рабочей среде. Это означает принятие менталитета DevOps. DevOps можно описать как «набор практик, предназначенных для сокращения времени между фиксацией изменения в системе и внедрением изменения в обычное производство при обеспечении высокого качества». Поскольку цикл быстрого изменения является одним из основных моментов микросервисов, развертывание должно быть быстрым и плавным. Такой процесс развертывания называется непрерывной доставкой. Он направлен на сокращение цикла выпуска приложения за счет совместной работы разработчиков и операторов. В монолитной организации разработчик просто коммитит код, а за развертывание отвечает специальная группа. Теперь каждая команда должна иметь возможность делать развертывания своего сервиса, а также, скорее всего, всего приложения целиком. Людей с соответствующими навыками может не хватить, поэтому членам команды необходимо освоить новые навыки и улучшить процесс развертывания. Когда операторы и разработчики работают вместе в одной команде, у них схожие цели, но обучение и адаптация требуют времени и терпения.

5. Заключение

Основываясь на различных проблемах, с которыми сталкиваются организации при переходе от монолитной архитектуры к микросервисной архитектуре, можно сделать вывод, что этот переход непрост и потребует много времени и усилий со стороны различных частей организации. Создание распределенной системы создает новые проблемы, которые необходимо решить. Несмотря на то, что микросервисы можно считать новой архитектурой программного обеспечения, они все еще в некоторой степени освоены в том смысле, что инструменты для микросервисов довольно хороши, и большинство проблем можно решить, применяя инструменты с открытым исходным кодом, созданные компаниями, которые уже совершили переход от монолита к микросервисам. Однако эти инструменты не решают проблемы рефакторинга и устранения тесной связи кодовой базы. Основное внимание, конечно, будет уделено технической стороне проблем, но организациям не следует забывать о законе Конвея. Структура организации должна быть похожа на их архитектуру. Итак, чтобы добиться успеха с микросервисами, необходимо решить как технические, так и организационные проблемы.

Рефакторинг в микросервисы — это большой процесс, который может занять много времени, и этот процесс требует участия всех частей организации. Этот переход по-прежнему выполним, как показали предыдущие примеры. Однако организация, рассматривающая этот переход, должна оценить стоимость и выгоду от перехода и подумать о своей проблемной базе. Архитектура микросервисов — это не план спасения, который работает для каждой организации, и в некоторых случаях проблемы перевешивают преимущества. Однако для кого-то это единственный способ

продолжать быструю разработку и быстро доставлять программное обеспечение своим клиентам.

Использованная литература

1. Sam Newman Building Microservices, Designing Fine-Grained Systems. — 1-е изд. — 2015: O'Reilly Media Inc.

2. Microservices a definition of this new term // MartinFowler URL: https://martinfowler.com/articles/microservices.html (дата обращения: 05.05.2022).

3. Microservices — Pattern: Microservice Architecture // Microservice Architecture URL: https://microservices.io/patterns/microservices.html (дата обращения: 04.05.2022).

4. Johannes Thones, Microservices IEEE Software // 2015. — С. 116-116.

5. M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas and S. Gil, Evaluating th

6. I Love APIs 2015: Microservices at Amazon // Chrismunns URL: https://chrismunns.com/talks.html (дата